Cours IN201

Année 2010-2011

Les Systèmes d'Exploitation

Responsable : Bertrand Collin.
Auteurs : Bertrand Collin, Marc Baudoin, Manuel Bouyer, Jérôme Gueydan, Thomas Degris, Frédéric Loyer, Damien Mercier.

École nationale supérieure
de **techniques avancées**

ENSTA
ParisTech

Avertissement

Les auteurs du document

Ce document est l'œuvre itérative des nombreux enseignants ayant participé au cours « Systèmes d'exploitation » de l'ENSTA ParisTech depuis sa création : même si chacun des chapitres a été initialement écrit par un seul enseignant, ceux-ci ont été ensuite relus, corrigés, complétés, remaniés chaque année et il est difficile aujourd'hui d'attribuer avec justesse la paternité exacte des différentes contributions.

Les auteurs des documents originaux sont cités ci-dessous, de même que les enseignants qui ont participé au cours au fil des ans, en espérant que personne n'ait été oublié.

Les chapitres allant de l'introduction à 5 ont été écrits par Jérôme Gueydan et mis à jour par Bertrand Collin. Le chapitre 6 s'inspire très fortement de textes écrits par Frédéric Loyer et Pierre Fiorini et a été remis à jour par Bertrand Collin. Le chapitre 7 a été écrit par Manuel Bouyer et Frédéric Loyer et très légèrement mis à jour par Bertrand Collin puis corrigé pour la partie concernant l'`UEFI`. Les chapitres 8 et 9 ont été rédigés par Bertrand Collin et relus par Marc Baudoin.

Les chapitres 10, 11, 12 et 13 sont l'œuvre de Manuel Bouyer, Jérôme Gueydan et Damien Mercier.

Les chapitres 14, 15 et 16 ont été écrits par Marc Baudoin et Damien Mercier.

Le chapitre 17 a été écrit par Bertrand Collin et patiemment relu et corrigé par Marc Baudoin et Manuel Bouyer.

Les chapitres 18 et 19 ont été rédigés par Thomas Degris.

Les exercices et corrigés proposés tout au long de ce document ont été conjointement écrits par Marc Baudoin, Manuel Bouyer, Bertrand Collin, Thomas Degris, Jérôme Gueydan et Damien Mercier.

Les projets décrits dans les chapitres 20, 21, 22, 23, 24 et 25 ont été respectivement proposés par Nicolas Israël, Olivier Perret, Damien Mercier et Bertrand Collin.

L'aide-mémoire du langage C proposé dans le chapitre 26 a été rédigé par Damien Mercier.

Les personnes suivantes ont par ailleurs participé à la constitution de ce document à l'occasion de leur participation au cours « Systèmes d'exploitation » : Robert N'Guyen, Régis Cridlig, Antoine de Maricourt, Matthieu Vallet.

Anglicisme et autres barbarismes

Il est très difficile d'expliquer le fonctionnement de l'informatique, des systèmes d'exploitation et des ordinateurs sans employer de mots anglais ou, pire, de mots francisés à partir d'une racine anglaise (comme *rebooter*, par exemple, souvent employé en lieu et place de réinitialiser). Nous avons essayé tout au long de ce document de cours d'utiliser systématiquement des mots français en indiquant en italique le mot anglais habituellement employé.

Néanmoins, certains mots anglais n'ont pas de traduction satisfaisante (comme *bug*, par exemple) et nous avons parfois préféré utiliser le mot anglais original plutôt que l'une des traductions officielles (bogue, en l'occurrence). Cependant, afin de ne pas tomber dans le travers fustigé au paragraphe précédent, cette utilisation des angliscismes a été strictement limitée au terme initial et des traductions françaises ont été employées pour les termes dérivés (par exemple, débogueur et débogage).

Enfin, certaines expressions anglaises sont parfois employées sans faire référence à leur sens originel (comme *fork*, par exemple). Dans ce cas, la traduction française donne une expression correcte, mais plus personne ne fait le lien entre cette expression et le sujet évoqué. L'essentiel étant que nous nous comprenions tous, nous avons dans ce cas-là utilisé le mot anglais.

Mises à jour

Les dernières mises à jour concernant les sujets présentés en cours, les énoncés et les exercices des travaux pratiques ou l'énoncé des projets figurent sur la page www consacrée à ce cours [1]. Tous les programmes présentés en exemple dans ce document sont disponibles via cette page www ainsi que la majorité des corrections des exercices de travaux pratiques.

Remerciements

Les auteurs remercient tous les (re)lecteurs bénévoles qui ont eu le courage d'indiquer les erreurs, les imprécisions et les fautes d'orthographe, ainsi que toutes les personnes qui, par le biais de discussions fructueuses, de prêts d'ouvrages ou d'explications acharnées, ont permis la réalisation de ce manuscrit.

En particulier, les auteurs remercient :

1. Cette page est accessible à partir de l'url `http://www.ensta.fr/~bcollin/`

– Thierry Bernard et Jacques Le Coupanec pour les nombreuses explications sur l'architecture des ordinateurs et la programmation de la MMU, ainsi que pour la relecture attentive du chapitre 1 ;
– Eric Benso, Christophe Debaert et Cindy Blondeel pour la relecture minutieuse des chapitres allant de l'introduction à 7 ;
– Régis Cridlig pour les nombreuses précisions apportées aux chapitres allant de l'introduction à 5 ;
– Béatrice Renaud pour la lecture attentive des chapitres allant de l'introduction à 7 ainsi que pour la correction des nombreuses fautes d'orthographe ;
– Lamine Fall et Denis Vervisch pour leur relecture patiente de l'ensemble du document.

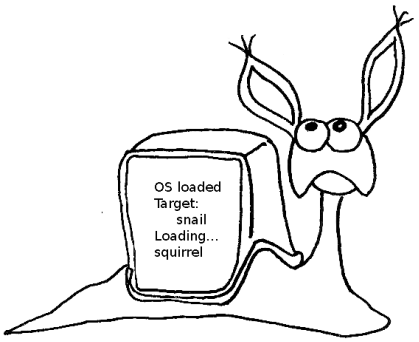
Glossaire

De nombreux acronymes ou abréviations sont utilisés dans ce manuscrit. Ils sont généralement explicités dans le texte, mais un résumé de ces termes employés paraît toutefois utile.

ADSL	*Asymmetric Digital Subscriber Line*
ALU	*Arithmetic and Logic Unit*
APIC	*Advanced Programmable Interrupt Controller*
ATA	*Advanced Technology Attachment*
ATAPI	*ATA with Packet Interface*
BSB	*Back Side Bus*
FSB	*Front Side Bus*
LAPIC	*Local Advanced Programmable Interrupt Controller*
CISC	*Complex Instruction Set Computer*
CPU	*Central Process Unit*
DDR	*Double Data Rate*
DRAM	*Dynamic Random Access Memory*
DRM	*Digital Right Management*
DVD	*Digital Versatil Disk*
SRAM	*Static Random Access Memory*
GPT	*Globally Unique Identifier Partition Table*
IPI	*Inter Processor Interrupts*
IMR	*Interrupt Mask Register*
IRQ	*Interruption ReQuest*
IRR	*Interrupt Request Register*
ISR	*In-Service Register*
MMU	*Memory Management Unit*
MPI	*Message Passing Interface*
PCIe	*Peripheral Component Interconnect Express*
PIC	*Programmable Interrupt Controler*
POSIX	*Portable Operating System Interface [for Unix]*
PTE	*Page Table Entry*

PVM	*Parallel Virtual Machine*
RISC	*Reduced Instruction Set Computer*
SCSI	*Small Computer System Interface*
SDRAM	*Synchronous Dynamic Random Access Memory*
SSII	*Société de Services en Ingénierie et Informatique ou Société de Services et d'Ingénierie en Informatique*
TLB	*Translocation Lookaside Buffer*
UEFI	*Unified Extensible Firmware Interface*
USB	*Universal Serial Bus*

Introduction



Pourquoi un système d'exploitation ?

Le système d'exploitation est l'élément essentiel qui relie la machine, composée d'éléments physiques comme le microprocesseur, le disque dur ou les barrettes mémoire, et l'utilisateur qui souhaite effectuer des calculs. Sans système d'exploitation, chaque utilisateur serait obligé de connaître le fonctionnement exact de la machine sur laquelle il travaille et il serait, par exemple, amené à programmer directement un contrôleur de périphérique USB pour pouvoir enregistrer ses données sur une clé USB. Sans le contrôle du système d'exploitation, les utilisateurs pourraient aussi détruire irrémédiablement certaines données stockées sur un ordinateur, voire détruire certains périphériques – comme le disque dur – en programmant des opérations illicites.

Les systèmes d'exploitation jouent donc un rôle majeur dans l'utilisation d'un ordinateur et si l'outil informatique s'est répandu dans le monde entier, c'est certes grâce à l'abaissement des prix d'achat et grâce à l'augmentation de la puissance des ordinateurs, mais c'est surtout grâce aux progrès réalisés lors des cinquante dernières années dans la programmation des systèmes d'exploitation : avec une machine de puissance équivalente, la moindre opération sur un ordinateur qui aujourd'hui nous paraît triviale était alors proprement impossible à réaliser !

Ce document présente les systèmes d'exploitation ainsi que leurs composants principaux et leurs structures. Toutes les notions essentielles à la compréhension du fonctionnement d'un ordinateur avec son système d'exploitation sont ici abordées, mais un lecteur désireux d'approfondir un sujet particulier doit se reporter à un ouvrage spécialisé, par exemple un des ouvrages cités dans la bibliographie. Les notions présentées ici ne font en général référence à aucun système d'exploitation en particulier. Néanmoins, nous avons choisi d'illustrer les fonctionnalités avancées des systèmes d'exploitation par des exemples provenant essentiellement des systèmes d'exploitation de type Unix. Nous justifierons plus loin ce choix et un lecteur critique pourra considérer que ce choix est a priori aussi valable qu'un autre.

Nous abordons aussi dans ce document certains sujets qui ne relèvent pas directement des systèmes d'exploitation, comme l'architecture des ordinateurs ou la compilation de programmes, mais dont l'étude permet d'expliquer plus facilement le rôle d'un système d'exploitation et de montrer l'étendue des services qu'il rend.

Pourquoi étudier les systèmes d'exploitation ?

Avant de se lancer à corps perdu dans l'étude des systèmes d'exploitation, il est raisonnable de se demander ce que cette étude peut nous apporter. Précisons tout d'abord que ce document est à l'origine le support écrit d'un cours proposé en deuxième année du cycle d'ingénieur de l'ENSTA ParisTech [2]. Les raisons évoquées ci-dessous s'adressent donc à tous les étudiants non spécialisés en informatique qui exerceront rapidement des métiers d'ingénieur à responsabilité.

Tous les ingénieurs issus de grandes écoles généralistes auront à utiliser l'outil informatique dans leur métier. Bien entendu, suivant le métier exercé ou suivant l'évolution de la carrière de chacun, cette utilisation sera plus ou moins fréquente et certains ne feront que tapoter de temps en temps sur leur clavier alors que d'autres passeront des heures à se battre avec (contre ?) la machine.

Quel que soit le profil choisi, tous les ingénieurs de l'ENSTA ParisTech se retrouveront dans une des trois [3] catégories suivantes.

L'utilisateur : comme son nom l'indique, la seule préoccupation de l'utilisateur est d'utiliser sa machine. Son désir le plus vif est que celle-ci se mette à fonctionner normalement quand il l'allume et que ses logiciels favoris lui permettent de travailler correctement.

Le décideur : il prend les décisions vitales concernant les choix stratégiques et commerciaux de l'informatique d'entreprise. C'est lui qui décidera par exemple s'il vaut mieux acheter un gros ordinateur relié à 50 terminaux ou s'il est préférable d'acheter 50 micro-ordinateurs

2. Voir `http://www.ensta.fr/~bcollin`.

3. Nous pourrions ajouter d'autres catégories afin de distinguer certaines professions comme par exemple le conseil en informatique.

connectés en réseau. Souvent le décideur se fonde sur les besoins exprimés par les utilisateurs pour prendre sa décision.

Le programmeur : il cherche à tirer le meilleur parti de la machine qu'il programme tout en perdant le moins de temps possible en développements. Le programmeur cherche aussi à préserver ses programmes du temps qui passe et tente de les réutiliser de machine en machine afin de ne pas tout recommencer à chaque fois.

Beaucoup d'élèves ingénieurs estiment spontanément que seule la 3e catégorie doit s'intéresser aux cours d'informatique et, notamment, aux systèmes d'exploitation. En fait, ceci est faux et c'est même une erreur grave.

L'utilisateur et les systèmes d'exploitation

L'utilisateur doit connaître le fonctionnement et les capacités des systèmes d'exploitation car c'est pour lui l'unique façon de savoir ce qu'il est en droit d'attendre d'un ordinateur. En pratique, lorsqu'un utilisateur exprime ses besoins, il fait souvent référence à des logiciels (ou à des machines) qu'il utilise déjà ou dont il a entendu parler. Il est très rare qu'un utilisateur exprime ses besoins en dehors de tout contexte commercial, c'est-à-dire sans citer d'exemples de logiciel ou de marque.

Ce phénomène a un effet pervers : il n'est pas rare qu'un utilisateur passe sous silence un besoin informatique soit parce qu'il pense que ce n'est pas possible, soit parce qu'il ne connaît pas d'exemples de logiciel répondant à ce besoin. Par exemple, bon nombre d'utilisateurs de PC dans les entreprises ne savent pas qu'un ordinateur peut fonctionner pendant de longs mois sans qu'il y ait besoin de le réamorcer (de le « rebooter ») et, leur environnement professionnel ou commercial ne proposant aucun système stable, ce besoin est rarement exprimé. Inversement, il est fort probable que si un système stable leur était proposé, il aurait beaucoup de succès [4].

Connaître le fonctionnement des systèmes d'exploitation et des ordinateurs permet donc à l'utilisateur d'exprimer ses besoins informatiques réels et non un sous-ensemble réduit de ces besoins. Par ailleurs, connaître le fonctionnement de la machine permet généralement d'adopter la bonne attitude quand celle-ci ne fonctionne pas exactement comme on le désire et cela évite souvent d'entreprendre des actions quasi-mystiques, voire totalement ésotériques, comme il arrive d'en voir dans certains cas...

Bien entendu, pour l'utilisateur, seuls les principes de fonctionnement comptent et il n'a pas besoin de connaître les détails concernant tel ou tel système d'exploitation. Il est néanmoins capital d'étudier un système d'exploitation complexe et efficace qui offre le plus de services possibles (comme Unix par exemple) et l'utilisateur ne peut donc se contenter d'étudier le système d'exploitation qu'il utilise habituellement [5].

4. La propagation rapide de Windows 2000, nettement plus stable que les précédentes moutures, et le succès mitigé de Windows XP, qui n'apporte rien de mieux dans ce domaine, tend à confirmer cette analyse.

5. Sauf si c'est Unix, bien entendu !

Le décideur et les systèmes d'exploitation

Le décideur doit s'enquérir des besoins des utilisateurs et prendre la décision d'achat informatique (par exemple) adéquate. La connaissance des systèmes d'exploitation est ici capitale pour – au moins – deux raisons : la réponse aux besoins des utilisateurs et la pérennité des choix informatiques.

Pour répondre aux besoins des utilisateurs, le décideur doit connaître les possibilités des machines et l'ensemble des logiciels disponibles (systèmes d'exploitation compris). Il doit aussi connaître les différentes possibilités matérielles, les réseaux, etc. Par ailleurs, il peut (doit ?) aider les utilisateurs à mieux définir leurs besoins en faisant abstraction des références commerciales habituelles. Cette étape souvent négligée est capitale, car si les utilisateurs n'expriment pas correctement leurs besoins informatiques, le décideur ne pourra pas prendre une décision efficace.

En supposant que notre décideur ait réussi à définir les besoins des utilisateurs dont il est responsable, il doit maintenant prendre la bonne décision. Le matériel informatique coûte beaucoup moins cher qu'auparavant et il est donc possible d'en changer souvent. Cette possibilité est trompeuse car elle laisse supposer que les choix stratégiques se font désormais à court terme, ce qui est totalement faux. Ainsi, si une entreprise décide de changer de type d'ordinateur pour passer par exemple de PC à Mac, il est probable que la plupart des développements qu'elle a effectués et la plupart des données qu'elle a stockées seront perdues dans l'opération. Dans le meilleur cas, cela nécessitera une phase de traduction d'un format à un autre, phase qui revient très cher.

Outre les coûts mentionnés ci-dessus, il faut aussi prendre en compte les coûts de formation des utilisateurs : rares sont les utilisateurs qui sont capables de s'autoformer à partir des documentations des systèmes d'exploitation ou des logiciels nouvellement installés et il est donc nécessaire d'organiser des sessions de formation, le plus souvent sous-traitées à des sociétés de formation et généralement très coûteuses (d'un à plusieurs milliers d'euros par jour et par personne). Ces sessions de formation peuvent aussi engendrer une interruption du service puisque les personnels de l'entreprise se retrouvent indisponibles pendant plusieurs jours (voire plusieurs semaines en temps cumulé).

En conséquence, même si le matériel est fréquemment renouvelé, la politique informatique est maintenue à long terme. Le décideur doit donc s'assurer que les choix qu'il fait ne mettent pas son entreprise en danger [6] et qu'ils lui permettront de s'adapter aux différents changements dans les années à venir.

Par ailleurs, il ne faut pas que les choix du décideur lient le sort de son entreprise à un fournisseur particulier, comme, par exemple, un fabricant d'ordinateurs : les fournisseurs savent pertinemment qu'il est difficile pour une entreprise de changer de type de machine et ils agissent alors en situation de quasi-monopole. Ils jouent sur le fait que cela revient moins cher à l'entreprise de payer très cher ce matériel plutôt que de changer de fournisseur.

6. La sécurité des solutions mises en place devrait à ce titre être une préoccupation majeure.

Une bonne solution pour assurer cette indépendance et pour répondre aux besoins de pérennité est d'utiliser des systèmes d'exploitation toujours plus ouverts, toujours plus efficaces, qui évoluent rapidement et qui sont portés sur de nombreuses architectures différentes.

Ce même raisonnement doit être tenu pour les applications sur serveurs (messagerie, gestionnaire de bases de données, serveurs web, etc.), pour les applications dites « métier » (outil de gestion financière, outil de gestion des ressources humaines, logiciel de publication assistée par ordinateur, etc.) et pour les applications de bureautique (traitement de textes, tableur, outil de présentation, etc.).

Le programmeur et les systèmes d'exploitation

La première préoccupation du programmeur est de ne pas refaire ce qui a déjà été fait, c'est-à-dire de ne pas perdre de temps à (ré)écrire du code qui existe déjà. La connaissance des systèmes d'exploitation est bien entendu extrêmement importante dans ce cas-là et ceux-ci se présentent alors comme des bibliothèques de services disponibles.

Cette vision des systèmes d'exploitation, même si elle a longtemps perduré, est cependant primitive et tout bon programmeur doit prendre en compte deux autres problèmes : le portage de ses programmes et le choix du système d'exploitation à utiliser. Le portage d'un programme est l'ensemble des actions permettant d'adapter un programme développé sur une machine donnée avec un système d'exploitation donné à une autre machine dotée éventuellement d'un autre système d'exploitation. Si le portage s'effectue immédiatement (ou au moins simplement), le programme est dit portable (les personnes peu soucieuses de la langue française parlent parfois de « portabilité »).

Le portage d'applications est un des principaux problèmes des entreprises actuelles, notamment des SSII, et les coûts de portage d'une application n'ayant pas été correctement conçue dès le départ sont généralement énormes.

Un des intérêts des systèmes d'exploitation est justement de rendre les programmes plus facilement portables, en s'intercalant entre la réalité physique des machines et les programmes des utilisateurs. Par exemple, en écrivant des programmes en langage C, le programmeur n'a pas besoin de se soucier de la machine sur laquelle il travaille et il pourra probablement porter son programme sur d'autres machines. Notons que ce n'est pas une garantie car beaucoup de constructeurs de machines proposent des bibliothèques de fonctions ou des facilités propres à leurs machines (afin de « fidéliser » les programmeurs, voir discussion sur le décideur).

Outre les spécificités des constructeurs, le programmeur est confronté aux différences des systèmes d'exploitation qui ne proposent pas tous la même interface. Un programme en C fonctionnant sous Unix, par exemple, a peu de chance de fonctionner sous Windows (et réciproquement). Un effort de standardisation a été réalisé pour faciliter les portages et une norme a été établie (Posix). En attendant que tous les systèmes

d'exploitation proposent la même interface [7], le choix d'un système d'exploitation est donc crucial et, en particulier, il est préférable de choisir le système qui assure le portage le plus facile sur le maximum de machines et de systèmes d'exploitation.

Notons que le choix d'un système d'exploitation ne doit pas être confondu avec le choix de la machine [8] sur laquelle ce système fonctionne et, même si la plupart des constructeurs de machine tentent d'imposer leur propre système en prétendant qu'aucun autre système ne fonctionne sur les machines en question, le système d'exploitation est un programme comme un autre : à ce titre, il se doit d'être portable et ce serait une aberration de choisir un système d'exploitation non portable pour développer des applications portables...

Plan du document

Ce document se compose de 26 chapitres répartis en 4 parties :
- une partie de cours général (chapitre 2 à 9) ;
- une partie traitant particulièrement de la programmation système en `C` sous Unix (chapitre 10 à 19) ;
- une partie proposant des projets informatiques (chapitre 20 à 25) ;
- une partie consistant en une aide-mémoire du langage `C` (chapitre 26).

Première partie : les systèmes d'exploitation

Cette partie est générale à tous les systèmes d'exploitation et, même si certains exemples particuliers sont choisis pour illustrer le propos, les principes s'appliquent à l'essentiel des systèmes d'exploitation.

De nombreux rappels sur l'architecture des ordinateurs sont effectués dans le chapitre « Rappels sur l'architecture des ordinateurs » et l'accent est particulièrement mis sur tous les composants des ordinateurs sans lesquels il serait très difficile de développer un système d'exploitation efficace. Une section de ce chapitre aborde un sujet un peu différent qui, cependant, a trait à la fois aux systèmes d'exploitation et à l'architecture des ordinateurs : la mesure des performances des ordinateurs.

Le chapitre « Qu'est-ce qu'un système d'exploitation ? » est consacré aux principales définitions des systèmes d'exploitation et à leur structure. Nous y annonçons en particulier quelles sont les tâches qu'un système d'exploitation doit assurer et nous esquissons les choix à faire pour assurer un fonctionnement efficace d'un ordinateur.

L'historique des systèmes d'exploitation est dressé au chapitre « Évolution des systèmes d'exploitation », ainsi que celui des ordinateurs : ordinateurs et systèmes d'exploitation se sont développés ensemble et il n'est pas possible de présenter l'un sans l'autre. Cet historique permet de constater que les principes des systèmes d'exploitation

7. En supposant que cela arrive un jour...
8. Ce choix est aussi très important.

n'ont pas été établis en une fois et que ce domaine a beaucoup évolué et évolue encore. À la fin de ce chapitre, nous passons en revue les systèmes d'exploitation actuels.

Le chapitre « Compilation et édition de liens » présente rapidement les principes de la compilation de programmes. Même si ce sujet ne fait pas partie de l'étude des systèmes d'exploitation, il est préférable de l'aborder afin de faciliter l'explication de la gestion des processus et de la gestion de la mémoire. Ces explications seront respectivement abordées aux chapitres « La gestion des processus » et « La gestion de la mémoire ».

Le chapitre « Le système de fichiers » sera consacré au système de fichiers qui, avec la gestion des processus et la gestion de la mémoire, est un élément essentiel des systèmes d'exploitation modernes. Comme les lecteurs sont généralement très familiers avec cet élément qui représente finalement l'abstraction ultime d'un ensemble de données numériques, nous avons repoussé son explication vers la fin de ce document.

Les chapitres « Les architectures multi-processeurs et les *threads* » et « La virtualisation des systèmes d'exploitation » ont été introduits récemment. Avec l'arrivée massive des processeurs multi-cœurs dans les stations de travail et les serveurs, il était indispensable de décliner les différents concepts vus précédemment dans le cadre d'une architecture parallèle. Toutefois il faut garder à l'esprit que le chapitre Les architectures multi-processeurs et les *threads* ne dispense en aucun cas de suivre assidument le cours IN203 « Parallélisme » car il ne donne qu'une partie des notions indispensables à la compréhension complète des *threads*. Le chapitre « La virtualisation des systèmes d'exploitation » se veut avant tout une introduction à la notion de virtualisation d'un système d'exploitation. Il permettra aux lecteurs de découvrir les fondements mais aussi les différentes méthodes employées pour virtualiser un serveur ainsi que les avantages et inconvénients de chacune de ces techniques. Enfin il se veut aussi un moyen de découvrir la terminologie employée afin de saisir rapidement, lorsqu'une décision s'impose, les offres proposées sur le marché actuel.

Deuxième partie : la programmation système en C sous Unix

La deuxième partie concerne uniquement le système d'exploitation Unix. La plupart des concepts abordés dans la première partie sont ici mis en œuvre par le biais de travaux pratiques de programmation système en langage `C`.

Les exercices proposés sont en général fournis avec leurs corrections et ils permettent de passer en revue l'essentiel des appels système Posix fournis par le système d'exploitation Unix.

Cette partie est composée des chapitres suivants : gestion des processus, création de processus, recouvrement de processus, entrées / sorties et système de fichiers, utilisation des signaux, communication entre processus par tuyau, gestion des entrées / sorties synchrones, communication entre machines par *socket* et, enfin, utilisation d'un débogueur.

Troisième partie : sujets des projets

La programmation de plusieurs projets est proposée dans cette partie. Les sujets sont assez ouverts et ils permettent à chacun de développer sa propre stratégie pour répondre au problème.

Quelle que soit la voie choisie, la résolution de ce problème sera fondée sur les principes détaillés dans les deux parties précédentes et, en particulier, le développeur des projets devra affronter, sous une forme légèrement différente, certains problèmes qui sont habituellement résolus par les systèmes d'exploitation.

Quatrième partie : aide-mémoire du langage `C`

Cet aide-mémoire n'est ni un manuel pour apprendre le langage `C`, ni un manuel de programmation. Il contient simplement de façon synthétique mais précise toutes les informations permettant de faciliter la résolution des exercices proposés en travaux pratiques.

En particulier, cet aide-mémoire décrit une partie des fonctions de la bibliothèque standard d'entrées / sorties du langage `C`. Signalons au lecteur que s'attaquer à la programmation système en `C` sous Unix sans maîtriser le langage `C` ou sans savoir ce dont il est capable relève du défi ! Dans tous les cas, une (re)lecture de cet aide-mémoire ne peut faire de mal à personne.

Les sujets qui ne sont pas traités dans ce document

Il n'est pas possible d'aborder tous les sujets concernant les systèmes d'exploitation dans un document synthétique et encore moins dans un cours de 2e année du cycle d'ingénieur. Les sujets suivants, bien qu'importants, ne seront donc pas traités dans ce document :

- les systèmes d'exploitation temps réel ;
- les systèmes embarqués ;
- les problèmes liés au parallélisme et aux systèmes distribués (ils sont abordés de manière superficielle au sein du chapitre « Les architectures multi-processeurs et les *threads* » ;
- l'utilisation de réseaux locaux [9] ou internationaux ;
- la sécurité des systèmes d'information (ce sujet sera abordé durant le dernier cours d'un point de vue macroscopique).

9. Une introduction à ce problème est néanmoins proposée dans le chapitre 16.

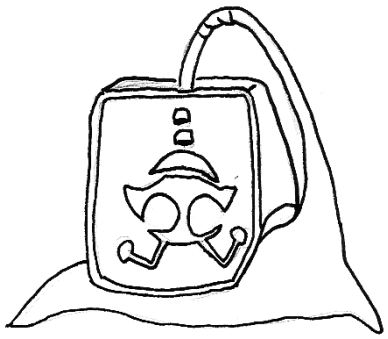
Table des matières

Première partie

Les systèmes d'exploitation

1

Rappels sur l'architecture des ordinateurs



Nous nous limitons dans ce chapitre à fournir les éléments permettant de comprendre le fonctionnement d'un ordinateur et le rôle d'un système d'exploitation. Certains aspects de l'architecture des ordinateurs sont volontairement simplifiés et il est préférable pour toute précision sur le fonctionnement d'un ordinateur de se reporter à un cours spécialisé.

Mots clés de ce chapitre : microprocesseur, processeur, mémoire, entrées/sorties, bus, registres, unité arithmétique et logique (ALU, Arithmetic and Logic Unit), unité de gestion de mémoire (MMU, Memory Management Unit), cache, coprocesseur, horloge, périphériques, contrôleur de périphériques, interruptions, jeu d'instructions, adresse physique, adresse virtuelle, page, segment.

1.1 Représentation symbolique et minimaliste d'un ordinateur

Un ordinateur peut être représenté symboliquement de la façon suivante (voir figure 1.1) : il se compose d'un microprocesseur (souvent abrégé en processeur) qui effectue les calculs, d'une unité de mémoire volatile qui permet de stocker les données et d'un certain nombre de périphériques qui permettent d'effectuer des entrées / sorties sur des

supports non volatiles comme un disque dur, une bande magnétique ou un CD-ROM. Ces composants dialoguent entre eux par l'intermédiaire d'un bus de communication.

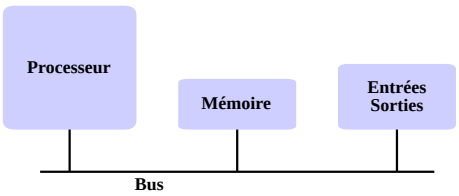


FIGURE 1.1 – *Représentation symbolique et minimaliste d'un ordinateur.*

En réalité, la structure d'un ordinateur est bien entendu plus complexe, mais le fonctionnement peut se résumer à cette représentation symbolique.

1.2 Représentation fonctionnelle d'un ordinateur

Afin de bien comprendre les conséquences de l'architecture des ordinateurs sur le développement des systèmes d'exploitation, nous allons adopter une représentation légèrement plus compliquée : l'architecture élémentaire d'un ordinateur représentée sur la figure 1.2. Elle se compose d'un microprocesseur, d'une horloge, de contrôleurs de périphériques, d'une unité de mémoire principale, d'une unité de mémoire cache et de plusieurs bus de communication.

Le microprocesseur

Le microprocesseur est lui-même composé de nombreux éléments. Nous citerons ici les principaux.

- **Les registres :** ce sont des éléments de mémorisation temporaire. Ils sont utilisés directement par les instructions (voir section 1.4) et par les compilateurs pour le calcul des valeurs des expressions et pour la manipulation des variables.
- **L'unité arithmétique et logique :** généralement nommée ALU (Arithmetic and Logic Unit), elle effectue toutes les opérations élémentaires de calcul.
- **L'unité de gestion de la mémoire :** généralement nommée MMU (Memory Management Unit), elle contrôle les opérations sur la mémoire et permet d'utiliser des adresses virtuelles pour accéder à une donnée en mémoire (voir section 1.5).
- **Le cache primaire :** cette unité de mémorisation temporaire fonctionne sur le même principe que le cache secondaire (voir section 1.2).

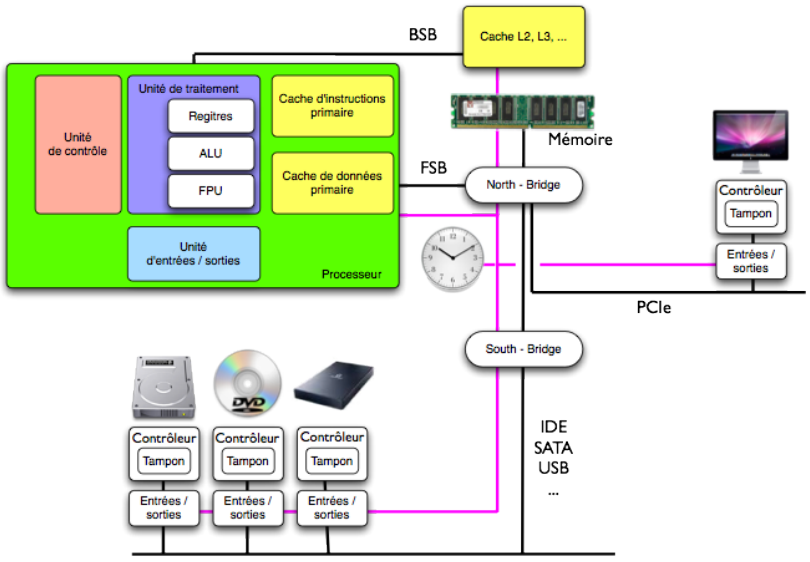


FIGURE 1.2 – *Représentation fonctionnelle d'un ordinateur.*

- Le coprocesseur : le microprocesseur peut contenir plusieurs coprocesseurs comme, par exemple, le coprocesseur « flottant » qui effectue tous les calculs en nombre non entiers (i.e. de type *float*). Autrefois optionnel sur les ordinateurs, il est aujourd'hui systématiquement présent.

- Le bus interne : il permet aux différents éléments du processeur de communiquer entre eux.

Une partie importante des processeurs dont nous n'avons pas parlé ici est celle concernant le contrôle des opérations sur les éléments le constituant et le décodage des instructions.

L'horloge

Le processeur est un composant synchrone, c'est-à-dire qu'il a besoin d'une horloge pour contrôler la cadence des opérations qu'il exécute. L'horloge contraint donc la vitesse de fonctionnement du processeur et il est fréquent que le processeur fonctionne moins vite qu'il ne le pourrait. En se fondant sur cette remarque, certains fabricants peu scrupuleux augmentent leurs bénéfices en associant des processeurs prévus pour fonctionner à (par exemple) 750 MHz avec des horloges à 1 GHz. Cette manipulation (appelée *overclocking*) réduit souvent de beaucoup la durée de vie du processeur car

la dissipation de chaleur qui en résulte est bien plus importante. D'autre part, après chaque « coup » d'horloge, les transitions qui affectent chaque ligne dans le processeur doivent se stabiliser. Changer la fréquence d'horloge peut mettre en péril la stabilité du processeur et conduire à des dysfonctionnements.

Sur certains PC anciens, il n'était pas rare de trouver un bouton « turbo » permettant d'augmenter la cadence de l'horloge. Néanmoins, ces PC fonctionnaient sur le principe inverse : la vitesse de l'horloge était en fait réduite pour des raisons de compatibilité avec certains périphériques obsolètes et l'augmentation ramenait l'horloge à la vitesse normale.

Ce même principe est aujourd'hui appliqué de façon automatique sur les processeurs des ordinateurs portables de dernière génération qui, en cas d'inactivité, abaissent leur vitesse de fonctionnement afin d'économiser de l'énergie et dissiper moins de chaleur.

La vitesse de transfert des bus de communication et de la plupart des périphériques est aussi régie par une horloge, mais il est fréquent que deux horloges différentes soient utilisées (ce qui permet aux vendeurs peu scrupuleux de tromper le client naïf – voir section 1.2).

Il est enfin important de comparer ce qui est comparable ! La vitesse de l'horloge ne doit servir à comparer que des processeurs de la même famille. L'annonce faite par Apple de commercialiser le Power Mac 8100, l'ordinateur possédant la plus haute fréquence d'horloge (110 MHz comparés aux 100 MHz des processeurs Intel), était assez douteuse puisque le PowerPC 601 et le Pentium étaient structurés différemment et n'utilisaient pas les mêmes jeux d'instructions.

> De nos jours, les processeurs multiplient généralement la vitesse de l'horloge par un facteur de 10 à 16 pour fonctionner plus rapidement en interne. C'est-à-dire que, par exemple, pour une horloge à 150 MHz, le processeur fonctionnera en interne à 1500 ou 2400 MHz. Ceci n'est efficace que si le processeur n'a pas besoin de communiquer avec le cache secondaire, la mémoire principale ou les entrées / sorties qui, eux, sont cadencés à des vitesses moindres (voir aussi section 1.2).

La mémoire principale

La mémoire principale (autrefois appelée « mémoire vive ») est généralement constituée de barrettes de type DRAM (Dynamic Random Access Memory). Ces mémoires doivent être rafraîchies régulièrement c'est-à-dire qu'il faut sans cesse lire puis réécrire les données qu'elles contiennent [1]. Par ailleurs, il est indispensable de réécrire systématiquement la donnée que l'on vient de lire. Ce travail de rafraîchissement est pris en charge de manière transparente par le contrôleur de mémoire.

1. Ce phénomène est lié à l'utilisation de pico-condensateurs dont les courants de fuite provoquent la perte de charge.

Il existe des mémoires de type SRAM (Static Random Access Memory) qui ne nécessitent pas de rafraîchissement. Malheureusement, elles sont beaucoup plus chères...

Les nouvelles technologies chassant les anciennes, on trouve maintenant de la SDRAM (Synchronous Dynamic Random Access Memory). Les mémoires de type DRAM nécessitaient un temps d'attente pour être certain que les données reflétaient bien le changement introduit. Ce n'est plus le cas avec les SDRAM qui attendent les fronts d'horloge pour prendre en compte les signaux d'entrée. Citons également les DDR SDRAM (Double Data Rate) qui fonctionnent de manière synchrone sur le front montant et le front descendant de l'horloge. Les DDR actuelles (DDR-600) peuvent assurer un débit de 4,8 Gio/s.

La mémoire est divisée en zones élémentaires qui représentent la plus petite quantité qui peut être stockée. Généralement, cette quantité est de 1 octet et la mémoire est alors divisée en un grand nombre d'octets contigus [2] que l'on numérote dans un ordre croissant. Pour accéder à une donnée, on parle alors de l'adresse de cette donnée en mémoire, c'est-à-dire du numéro de l'octet dans lequel est stockée la donnée ou le début de la donnée si celle-ci occupe plusieurs octets.

Une donnée stockée en mémoire peut donc être lue à condition de connaître son adresse et sa taille, c'est-à-dire le nombre d'octets à lire. Notons à ce propos que, suivant les modèles de processeur, les données dont la taille dépasse l'octet ne sont pas toujours rangées de la même façon en mémoire (pour une même adresse) [3].

La mémoire cache

La mémoire cache [4] (parfois appelé simplement « le cache ») est une zone de mémoire de petite taille, plus rapide que la mémoire principale et située plus près du processeur. Cette mémoire cache est utilisée comme intermédiaire pour tous les accès à la mémoire principale : chaque fois que le processeur désire accéder à une donnée stockée en mémoire principale, il lit en fait celle-ci dans la mémoire cache, éventuellement après que cette donnée a été recopiée de la mémoire principale vers le cache s'il s'avère qu'elle n'était pas présente dans le cache au moment de la lecture.

La mémoire cache étant de taille réduite, il n'est pas possible de conserver ainsi toutes les données auxquelles accède le processeur et, généralement, les données les plus anciennes sont alors écrasées par les données qui doivent être recopiées. Cette stratégie est efficace si le processeur lit plusieurs fois la même donnée dans un laps de temps très court ou s'il utilise peu de données :

2. La mémoire étant composée de plusieurs barrettes, ces octets ne sont pas véritablement contigus.

3. La norme IEEE 754 (`http://floating-point-gui.de` pour aller plus loin) donne la représentation suivante pour un nombre réel à virgule flottante. Un nombre réel est enregistré dans un mot de 32 bits (soit 4 octets). Le bit de poids fort donne le signe s du réel. Les 8 bits qui suivent donnent l'exposant x et les 23 bits de poids faible donnent la partie significative m (parfois appelée mantisse). On obtient la valeur du réel par $r = (-1)^s \times (1 + m \times 2^{-23} \times 2^{x-127})$

4. Le terme exact est « mémoire tampon », mais le terme « cache » est si couramment utilisé que certains informaticiens ne comprennent plus le terme de « mémoire tampon ».

– lors du premier accès à la donnée, celle-ci ne se trouve pas dans la mémoire cache et elle est alors copiée de la mémoire principale vers la mémoire cache ;
– lors des accès suivants, la donnée est déjà présente dans la mémoire cache et il n'est pas nécessaire d'accéder à la mémoire principale (qui est beaucoup plus lente).

Notons que cette stratégie peut se révéler très efficace, notamment lorsque l'ordinateur est utilisé pour des calculs scientifiques : la multiplication de deux matrices est un exemple typique de calcul où il est nécessaire d'accéder plusieurs fois aux mêmes données. Inversement, cette stratégie est parfois très mauvaise, en particulier lors d'accès séquentiels à des données, comme par exemple le parcours de tableaux. Même si aujourd'hui les compilateurs ont fait de nombreux progrès quant à l'utilisation du cache, tout bon programmeur doit faire attention lors de la conception de programmes à ne pas utiliser l'écriture la plus défavorable à la politique de cache (par exemple, en cas d'utilisation de tableaux à plusieurs dimensions)...

En pratique, le processeur ne peut pas savoir à l'avance si la donnée qu'il cherche se trouve ou ne se trouve pas dans la mémoire cache et la copie de la donnée à partir de la mémoire principale ne peut donc pas être effectuée de façon préventive. Une stratégie de tentative / échec est alors employée et nous verrons que cette stratégie intervient à de nombreux endroits dans la conception des ordinateurs et des systèmes d'exploitation :

– le processeur lit la donnée dans le cache ;
– si la donnée n'est pas présente, un défaut [5] de cache a lieu (*cache fault* [6] en anglais) ;
– ce défaut de cache déclenche alors une action prédéterminée, en l'occurrence la copie de la donnée de la mémoire principale vers la mémoire cache ;
– la donnée peut alors être lue par le processeur.

L'échange de données entre la mémoire cache et la mémoire principale est similaire à l'échange de données entre la mémoire principale et la mémoire secondaire (généralement le disque dur) qui sera abordé dans le chapitre sur la gestion de la mémoire.

Le même principe est aussi employé entre le cache primaire, logé directement au sein du processeur, et le cache secondaire, logé à l'extérieur. La plupart des processeurs possèdent même plusieurs zones de mémoire cache interne, parfois appelées cache de niveau 1, cache de niveau 2, etc. La plupart des périphériques et des contrôleurs de périphériques possèdent aussi de la mémoire cache pour accélérer les transferts et ce principe est donc très générique.

Pourquoi utiliser cinq zones différentes de mémoire (registres, cache primaire, cache secondaire, mémoire principale et mémoire secondaire) ? On pourrait en effet imaginer que le processeur qui a besoin d'une donnée aille directement la chercher dans

5. Croire qu'un défaut de cache traduit un mauvais fonctionnement de l'ordinateur ou du système d'exploitation serait donc une grossière erreur. Nous verrons plus loin d'autres défauts qui apparaissent fréquemment, comme les défauts de page.

6. Le terme *fault* est souvent traduit par « faute ». Le terme « défaut » est néanmoins plus approprié.

la mémoire principale sans passer par le cache. Il y a en fait deux raisons intimement liées qui justifient ce choix : le temps d'accès à une donnée et le prix de la mémoire. Il est très difficile de construire des mémoires qui soient à la fois rapides et de grande taille. Par ailleurs, le prix de l'octet de mémoire dépend très fortement et de façon non linéaire de sa rapidité. Se limiter à une seule zone de mémoire revient donc à faire un choix définitif : des accès rapides à une tout petite quantité de mémoire ou des accès lents à une grande quantité de mémoire.

L'idée consiste donc à utiliser des zones mémoires très rapides pour stocker quelques données utilisées très souvent et des zones mémoires moins rapides pour les autres. On parle alors de hiérarchie des mémoires. Le tableau 1.1 résume les quantités de mémoire généralement utilisées et les temps d'accès aux données stockées dans ces mémoires. Outre le temps d'accès nominal à ces zones mémoire, il faut aussi prendre en compte les temps d'accès aux différents intermédiaires. Par exemple, le processeur accède directement au cache primaire, mais doit passer par le bus externe et le contrôleur de cache pour accéder au cache secondaire.

Nom	registres	L1	L2	RAM	Swap
Taille	1 Ko	64 Ko	1 Mo	2 Go	4 Go
Temps d'accès	0.25 ns	1 ns	2 ns	5 à 70 ns	20 ms
Débit	32 à 64 Go/s	32 Go/s	16 Go/s	12 Go/s	300 Mo/s

TABLE 1.1 – *Quantités et temps d'accès typiques des mémoires utilisées sur un ordinateur. La mémoire cache L1 est situé à l'intérieur du microprocesseur et utilise la même horloge que le processeur, ce qui en fait une mémoire à l'accès très rapide. La mémoire cache L2 peut être située à l'intérieur ou partagée entre les deux cœurs (dans le cas d'un* double cœur*). Elle est légèrement moins rapide. En fait ces deux accès dépendent fortement de l'emplacement et du mode d'accès ; si le processeur utilise un bus système pour accéder au cache L2, naturellement les performances sont à revoir à la baisse.*

Les contrôleurs de périphériques

En réalité, le processeur ne communique pas directement avec les périphériques : ceux-ci sont reliés à des contrôleurs de périphériques et c'est avec eux que le processeur dialogue. Un contrôleur peut gérer plusieurs périphériques et le processeur n'a pas besoin de connaître précisément ces périphériques : s'il souhaite, par exemple, écrire une donnée sur le disque dur, il le demande au contrôleur de disque dur et c'est ce dernier qui se débrouille pour effectivement satisfaire la demande du processeur. Le processeur transmet alors la donnée à écrire au contrôleur, le contrôleur la stocke dans une mémoire tampon qu'il possède et la transmettra au disque dur.

Ce relais de l'information par des contrôleurs permet déjà de s'abstraire des spécificités des périphériques : une partie de ces spécificités n'est connue que du contrôleur. Cela permet par exemple de développer des périphériques très différents les uns des autres sans qu'il y ait de problème majeur pour les insérer dans un ordinateur. Cela permet aussi de renouveler les périphériques d'un ordinateur sans avoir à changer quoi que ce soit (si ce n'est le périphérique en question, bien entendu !).

Parmi les contrôleurs utilisés dans les ordinateurs, nous citerons le contrôleur `SCSI` (Small Computer System Interface) sur lequel il est possible de connecter indifféremment des disques durs, des lecteurs de `CDROM`, des lecteurs de bandes magnétiques et bien d'autres périphériques encore. Pour le processeur, et donc pour tous les programmes d'un ordinateur, l'accès à un disque dur `SCSI` est parfaitement identique à l'accès à un lecteur de `CDROM` `SCSI` : seule la spécificité du contrôleur `SCSI` importe. Les périphériques `SCSI` sont cependant un peu plus chers que les périphériques `IDE` (appelés aussi périphériques `ATA` ou `ATAPI` ou encore `SATA`) et ces derniers, bien que moins « universels » et souvent moins performants sont donc plus répandus. La généralisation des périphériques `USB` ou `Firewire` va peut-être permettre de trouver un juste compromis.

Quels sont les périphériques d'un ordinateur ? En fait, il n'y a pas de limite stricte et on peut considérer que tous les composants d'un ordinateur qui ne sont pas regroupés dans le processeur sont des périphériques. À ce titre, la mémoire principale est un périphérique comme les autres et elle est gérée par un contrôleur. Pour éclaircir un peu les idées, voici la liste des périphériques qu'un ordinateur moderne [7] doit posséder :

- **Un clavier :** difficile de faire quoi que ce soit sans clavier ;
- **Une souris :** la souris est apparue assez récemment et paraît aujourd'hui absolument indispensable. Elle a peu évolué et se présente parfois sous forme de boule, de levier (*joystick*) ou de surface sensitive (*touchpad*) ;
- **Un écran :** comme pour la souris, il est difficile aujourd'hui d'imaginer un ordinateur sans écran, mais il faut savoir que l'apparition des écrans est aussi assez tardive. Ils évoluent lentement en proposant de plus grandes tailles (17, 19 ou 24 pouces), de meilleures résolutions (1600×1200 voire 1920×1080) et toujours plus de couleurs (16 millions). La prochaine évolution devrait consister en la disparition complètes des tubes cathodiques au profit des écrans plats ainsi qu'à la fusion entre les écrans de télévision et les écrans d'ordinateur. De nombreux écrans équipant soit des portables soit des ordinateurs fixes font maintenant état de leur capacité à afficher des `DVD Blu-Ray` dans une résolution HD ;
- **De la mémoire :** le prix des barrettes de mémoire a soudainement chuté en 1996, passant de 150 à 20 € les 4 Mo ! En octobre 1999, une

7. Tous les chiffres présentés ici doivent être revus à la baisse pour les ordinateurs portables qui sont confrontés à des problèmes d'encombrement et de consommation électrique.

barrette de 32 Mo coûtait environ 40 € et, aujourd'hui, le prix est presque de 0,05 € pour 1 Mo (sauf pour les mémoires de dernière génération, toujours plus chères). La mémoire est donc devenue une denrée beaucoup moins rare et il est désormais fréquent de voir des ordinateurs personnels munis de 4 Go de mémoire et certaines stations de travail spécialisées (notamment celles qui accueillent des serveurs virtuels) vont jusqu'à 64 Go !

- Un disque dur : un ordinateur possède souvent un, voire plusieurs, disques durs. La tendance actuelle est de l'ordre de 350 Go pour un ordinateur portable et d'au moins 500 Go pour une station de travail personnelle. Leurs capacités croissent d'année en année et, aujourd'hui, il est impensable [8] d'acheter un disque dur de moins de 150 Go et quasiment impossible de trouver des disques durs neufs de moins de 20 Go.

- Un lecteur de disquette : les disquettes (1,44 Mo) sont en train de disparaître au profit de nouveaux supports comme les clés USB (de 256 Mo à 32 Go). Le principe reste cependant le même.

- Un lecteur de CD-ROM ou de DVD : les CD-ROM vont à court terme être supplantés par les DVD qui peuvent contenir beaucoup plus de données (jusqu'à 17 Go pour un DVD double couche et double face contre 650 Mo pour un CD-ROM). L'existence de graveurs de DVD à bas prix (de l'ordre de 70 € suivant le type et la vitesse de gravure) et l'existence de DVD devraient rapidement sonner le glas du CD-ROM.

- Un lecteur de bandes magnétiques : essentiellement utilisées pour effectuer des sauvegardes, les bandes magnétiques souffrent d'un défaut majeur, à savoir l'accès séquentiel et lent aux données. Elles restent cependant un support très fiable, notamment dans le temps.

- Une interface réseau : pour pouvoir se connecter à un réseau local (lui-même éventuellement relié à un réseau international tel que l'Internet).

- Un modem : pour effectuer des transferts de données numériques via une ligne téléphonique et, par exemple, se connecter à un réseau distant. L'avènement de l'ADSL a rendu ce périphérique moins fréquent et, même si les modems existent toujours, ils sont maintenant le plus souvent externes aux ordinateurs.

- Une carte son : indispensable pour transformer en signal analogique (audible via des haut-parleurs) les sons stockés sous forme numérique.

8. Hormis pour des raisons de compatibilité avec des ordinateurs anciens ou avec des systèmes d'exploitation qui n'ont pas été prévus pour fonctionner avec de telles quantités de disque dur.

Les bus de communication

Un ordinateur possède généralement de nombreux bus de communication spécialisés que l'on symbolise souvent comme étant un seul bus. Il est traditionnel d'en citer au moins trois : le bus de données sur lequel sont transférées les données à traiter, le bus d'adresses qui sert uniquement à transférer les adresses des données en mémoire et le bus de commandes qui permet de transférer des commandes entre le processeur et les périphériques. Par exemple, lorsque le processeur désire lire une information dans la mémoire principale, il envoie l'adresse de cette information sur le bus d'adresses ; le bus de commande indique au contrôleur de la mémoire principale qu'il doit donner l'information au processeur ; le contrôleur renvoie alors l'information demandée sur le bus de données.

La taille des bus de communication est un facteur important dans la rapidité des ordinateurs. Elle se mesure en bits et représente la taille de la plus grande donnée qui peut être transmise en une seule fois par ce bus. Supposons que nous disposons d'un ordinateur muni d'un bus de 8 bits ; si nous voulons transmettre un entier composé de 32 bits (c'est souvent le cas en C, par exemple), il faudra le découper en 4 portions de 8 bits et le transmettre en 4 fois, d'où une perte de temps. Cette perte de temps n'est bien entendu pas très grande en elle-même, mais multipliée par un grand nombre de données à transmettre, cela peut devenir très pénalisant.

Ainsi, si nous disposons d'un bus de 8 bits capable d'effectuer 33 millions d'opérations par seconde (on dit souvent : un bus de 8 bits à 33 MHz) et si nous ne traitons que des données de 32 bits, le bus pourra transmettre au mieux [9] 8,25 millions de données par seconde.

Notons que les constructeurs restent souvent discrets sur les performances des bus des ordinateurs qu'ils produisent car cela leur permet de vendre plus facilement leur dernier modèle : un processeur à 500 MHz peut effectivement effectuer 500 millions d'opérations par seconde à condition qu'il n'ait pas besoin de communiquer avec les éléments situés à l'extérieur du processeur, c'est-à-dire en particulier avec les différents éléments de stockage (cache secondaire, mémoire principale, mémoire secondaire). Réciproquement, un processeur à 500 MHz qui aurait besoin de lire ou d'écrire en permanence des données en mémoire sera limité par la vitesse du bus, par exemple 100 MHz, et passera donc les quatre cinquièmes de son temps à attendre que les données soient disponibles. On peut alors se demander s'il est vraiment intéressant d'acheter le tout dernier processeur qui fonctionne non pas à 2 GHz mais à 2,5 GHz, ce qui lui permettra de perdre non pas les neuf dixièmes de son temps mais les quatorze quinzièmes...

Notons qu'en pratique il est très rare qu'un processeur ait besoin de lire ou d'écrire des données en permanence et l'utilisation d'un processeur plus rapide conduit donc généralement à un gain de temps (voir discussion de la section 1.6 sur ce sujet et sur la façon dont on peut apprécier un gain de temps). Néanmoins, ce gain n'est pas

9. Le découpage de la donnée en 4 lots prend un certain temps...

proportionnel au gain réalisé en vitesse de processeur et, formulé différemment, un ordinateur muni d'un processeur fonctionnant à 1 GHz ne sera pas deux fois plus rapide qu'un ordinateur muni d'un processeur fonctionnant à 500 MHz.

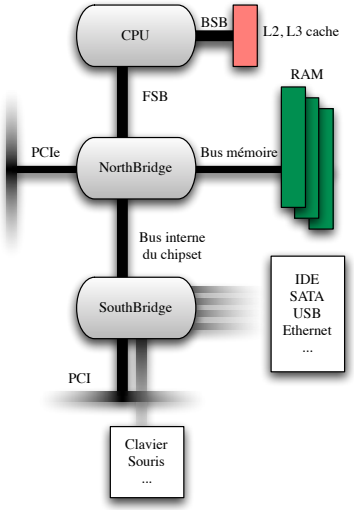


FIGURE 1.3 – *Les composants d'un ordinateur ne sont pas tous reliés entre eux par le même bus. Afin de réduire les latences, différents bus sont utilisés et deux micro-composants (le NorthBridge et le SouthBridge) contrôlent la circulation des données d'un composant à un autre.*

Le bus système, ou bus processeur, ou encore bus mémoire, souvent connu sous le nom de FSB (*Front Side Bus*) permet de connecter le processeur à la mémoire de type RAM ainsi qu'à d'autres composants (voir fig. 1.3). Les communications sur ce bus système sont gérées par un composant particulier de la carte mère : le NorthBridge. C'est ce composant qui conditionne souvent le chipset de la carte mère. Les communications avec des périphériques plus lents tels que les disques durs par exemple sont gérées par un autre composant : le SouthBridge.

Le bus de cache ou BSB (*Back Side Bus*) réalise la connexion entre le processeur et la mémoire cache secondaire (cache L2 et cache L3) quand celle-ci n'est pas directement intégrée au processeur.

Le bus PCIe (*Peripheral Component Interconnect Express*) permet la connexion de périphériques variés comme une carte video,... Cette nouvelle technologie, compatible avec l'ancien bus PCI, adopte un modèle de transmission par commutation de paquets (très proche du modèle réseau TCP/IP) sur une ligne série à la différence de l'ancien protocole parallèle dans lequel un composant monopolisait le bus pendant sa transaction.

Cette ligne série peut d'ailleurs être décomposée en plusieurs voies afin de paralléliser les transferts.

Le processeur et les bus de communication étant cadencés par l'horloge, la vitesse des processeurs est généralement un multiple de la vitesse du bus. Actuellement, les progrès réalisés pour les bus de communication sont beaucoup plus lents que ceux réalisés pour les processeurs et les bus de communication fonctionnent généralement à 600 ou 1060 MHz. Dans un passé proche, certains modèles de PC faisaient directement référence à ce phénomène en indiquant le facteur multiplicatif. Par exemple, on parlait de « DX2 66 » (bus à 33 MHz et processeur à 66 MHz). Notons que cette habitude a rapidement dégénéré car les « DX4 100 », par exemple, étaient en fait des « DX3 100 »... Il est généralement intelligent de choisir un ordinateur dont le FSB et le CPU sont dans une juste proportion...

Les interruptions

Les différents composants d'un ordinateur communiquent *via* des interruptions qui, comme leur nom l'indique, peuvent provoquer l'interruption de la tâche qu'effectue le processeur. Ce dernier peut alors décider de donner suite à l'interruption en déclenchant une routine d'interruption appropriée ou il peut décider d'ignorer l'interruption. Généralement, les routines ainsi exécutées font partie du système d'exploitation et les interruptions facilitent donc l'intervention de celui-ci.

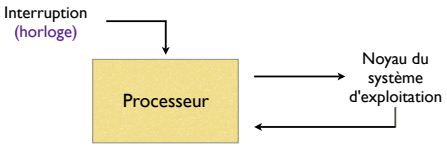


FIGURE 1.4 – *Le système d'exploitation reprend régulièrement la main grâce aux interruptions : à chaque « tic », l'horloge déclenche une interruption qui sera traitée par le processeur* via *l'exécution d'une routine particulière, située dans une zone mémoire particulière. Il suffit donc de placer le système d'exploitation dans cette zone mémoire pour garantir l'intervention régulière de celui-ci.*

Le terme d'interruption est à rapprocher du mot interrupteur, car c'est par le biais d'une liaison électrique que la communication s'établit : lorsqu'un périphérique, par exemple, veut déclencher une interruption du processeur, il applique une tension sur une ligne électrique les reliant (voir figure 1.5).

Les interruptions sont utilisées pour informer de l'arrivée d'événements importants, comme par exemple lors d'accès mémoire impossibles ou lorsqu'un périphérique a terminé le travail qui lui était demandé. Généralement, on distingue les interruptions matérielles, effectivement produites par le matériel, et les interruptions logicielles

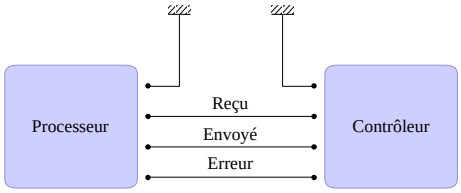


FIGURE 1.5 – *Les composants des ordinateurs communiquent en appliquant des tensions sur des lignes électriques*

que le processeur s'applique à lui-même. Ces interruptions logicielles jouent un rôle important dans la conception des systèmes d'exploitation.

Les interruptions matérielles ne sont habituellement pas transmises directement au processeur et elles transitent par divers contrôleurs. Il se peut alors que l'un de ces contrôleurs décide de ne pas transmettre certaines demandes d'interruptions (souvent appelées IRQ pour *Interruption ReQuest*) et, donc, qu'une demande d'interruption déclenchée par un périphérique ne provoque pas l'interruption de la tâche en cours. Le terme d'interruption est donc assez mal choisi, ce qui explique l'apparition d'autres termes : exceptions, trappes (*trap* en anglais). Certains contrôleurs d'interruption autorisent les utilisateurs à masquer des interruptions, ce qui représente une sorte de « programmation bas-niveau » du matériel.

Précisons, enfin, que rien n'assure que la transmission d'une interruption est synchrone...

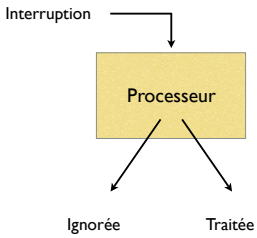


FIGURE 1.6 – *Une interruption peut être ignorée par le processeur (ou par un des contrôleurs transmettant celle-ci) et n'interrompt donc pas toujours un traitement en cours.*

La gestion des interruptions est assez complexe et nécessite en particulier de prendre quelques précautions : si une interruption interrompt une tâche et que le processeur décide de déclencher une routine d'interruption, il est absolument nécessaire de sauvegarder l'état de la tâche interrompue, avec en particulier le contenu des

registres, afin qu'elle puisse reprendre dès la fin de la routine d'interruption.

Un autre problème se pose pour les interruptions : que se passe-t-il si une interruption intervient alors que le processeur est déjà en train d'exécuter une routine d'interruption ? Généralement, des niveaux de priorité différents ont été affectés aux interruptions et le choix dans de telles situations dépend de la priorité des interruptions traitées.

Cette tâche est en général du ressort du `PIC` [10] (« *Programmable Interrupt Controler* ») qui contrôle le niveau de priorité des interruptions (« *Interrupt Priority Level* »).

Les `PICs` fonctionnent sur la base d'un ensemble de registres : l'`IRR` (« *Interrupt Request Register* ») qui spécifie quelles interruptions sont en attente d'acquittement, l'`ISR` (« *In-Service Register* ») qui contient les interruptions acquittées mais en attente d'un signal de Fin d'Interruption (`EOI` soit « *End Of Interrupt* ») et enfin l'`IMR` (« *Interrupt Mask Register* ») qui tient la liste des interruptions ignorées et donc non acquittées.

On distingue même sur les architectures à plusieurs processeurs deux gestionnaires de politique de priorité [11] : le traditionnel `APIC` et le `LAPIC` (pour « *Local* » `APIC`) dont le rôle est de traiter les interruptions entre un processeur et un autre (« *Inter-Processor Interrupts* ») et naturellement de s'occuper des interruptions externes signalées au processeur auquel il est attaché.

1.3 Mode noyau *versus* mode utilisateur

Les processeurs récents ont au moins deux modes de fonctionnement : un mode noyau (ou système ou superviseur) et un mode utilisateur [12]. Ces deux modes sont utilisés pour permettre au système d'exploitation de contrôler les accès aux ressources de la machine. En effet, lorsque le processeur est en mode noyau, il peut exécuter toutes les instructions disponibles (voir section suivante) et donc modifier ce qu'il veut. En revanche, lorsque le processeur est en mode utilisateur, certaines instructions lui sont interdites.

Le système d'exploitation (au moins une partie) d'un ordinateur est généralement exécuté en mode noyau alors que les autres programmes fonctionnant sur la machine sont exécutés en mode utilisateur. Ce principe contraint ces programmes à faire appel au système d'exploitation pour certaines opérations (en particulier pour toutes les opérations de modification des données) et représente donc une protection fondamentale, située au cœur du système, puisque tous les accès aux données sont nécessairement contrôlés par le système d'exploitation [13]. Nous verrons dans la suite du document

10. On le dénomme aussi «*Advanced Programmable Interrupt Controler*» pour `APIC`.
11. Il est important de noter que ces gestionnaires faisant partie de la carte mère ne sont pas exempts de bugs. C'est pourquoi on retrouve parfois, pour contourner les bugs matériels les fameuses options de démarrage `noapic,nolapic` sous Linux.
12. La presse informatique désigne parfois ces modes par « ring 0 » et « ring 3 ».
13. Voir section 2.1 à ce sujet.

comment ce principe est utilisé et comment il paraît difficile d'espérer développer un système d'exploitation efficace contre les intrusions sans adopter une telle démarche [14].

L'utilisation de ces modes est un bel exemple du développement complémentaire des systèmes d'exploitation et des machines. En effet, la distinction noyau versus utilisateur provient de considérations liées aux systèmes d'exploitation et le fait qu'elle soit assurée par les couches matérielles de la machine simplifie la tâche du système. Réciproquement, une fois cette distinction implantée, elle a permis de développer des couches matérielles plus élaborées et plus efficaces.

1.4 Le jeu d'instructions

Le jeu d'instructions est l'ultime frontière entre le logiciel et le matériel. C'est la partie de la machine vue par le programmeur et c'est le langage dans lequel les compilateurs doivent transformer les codes sources de plus haut niveau.

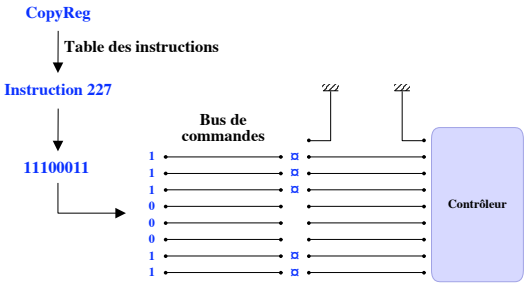


FIGURE 1.7 – *Le jeu d'instruction est la couche ultime entre le logiciel et le matériel. La traduction de chaque instruction en action « électrique » s'effectue par une simple table de correspondance et une instruction n'est qu'une représentation symbolique de l'application de tensions sur des contacteurs.*

Les jeux d'instructions dépendent du processeur et varient énormément d'un type d'ordinateur à un autre. En particulier, le nombre d'instructions disponibles et le travail effectué par ces instructions sont difficilement comparables.

Les instructions sont généralement classées en 5 catégories :

- arithmétique et logique (addition, soustraction, et, ou) ;
- flottant (opérations flottantes).
- transfert de données (de la mémoire principale vers les registres et réciproquement) ;
- contrôle (appel de procédure, branchement, saut) ;

14. Précisons qu'il existe des systèmes d'exploitation qui n'utilisent que le mode noyau du processeur.

– système (trappe, appel au système d'exploitation) ;

Les instructions système sont typiquement les instructions interdites en mode utilisateur. En particulier, citons l'instruction permettant le changement de contexte (*context switching*, voir chapitre 5 sur les processus et sur l'ordonnancement), l'instruction permettant de manipuler la pile (voir chapitre 4 sur la compilation et les processus) et l'instruction permettant de passer du mode utilisateur au mode noyau (et réciproquement).

On peut alors se demander comment un ordinateur peut passer du mode utilisateur au mode noyau, puisqu'il doit déjà être en mode noyau pour exécuter l'instruction permettant ce passage... En fait, il existe une instruction spéciale qui déclenche une interruption logicielle et, donc, l'exécution de la routine d'interruption correspondante. Cette routine fait généralement partie du système d'exploitation, c'est-à-dire qu'elle s'exécute en mode noyau, et elle permet donc de contrôler que rien d'illégal n'est demandé. Lorsque la routine est terminée, le processeur repasse en mode utilisateur.

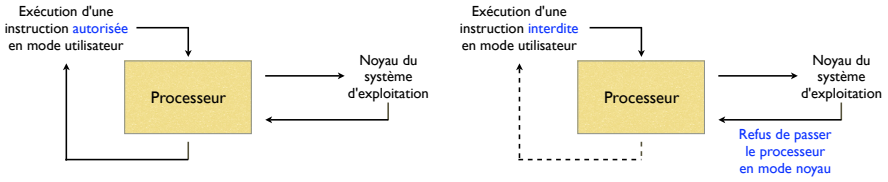


FIGURE 1.8 – *Le passage du mode utilisateur au mode noyau fait nécessairement appel au système d'exploitation : les appels système provoquent une interruption logicielle qui déclenche une routine d'interruption correspondant au système d'exploitation. Le principe est similaire à celui mis en œuvre avec les interruptions de l'horloge.*

Notons que rien n'oblige à utiliser une routine relevant du système d'exploitation et que, d'ailleurs, certains systèmes laissent l'utilisateur exécuter de cette façon ses propres routines : c'est ce qu'on appelle le déroutement de l'interruption.

Notons aussi que l'utilisation de cette instruction spéciale simplifie grandement la tâche des systèmes d'exploitation et que, comme nous le verrons tout au long de ce cours, il est difficile de concevoir un système d'exploitation sans utiliser les couches matérielles des ordinateurs.

Une instruction se déroule en plusieurs étapes, effectuées par des composants différents. Elle comporte au moins cinq étapes :

– la recherche de l'instruction en mémoire (*fetch*) ;
– le décodage de l'instruction (*decode*) ;
– la lecture des opérandes (*load*) ;
– l'exécution proprement dite (*execute*) ;
– le stockage du résultat (*store*).

Une instruction peut être décomposée en plusieurs opérandes. Le nombre de ces opérandes dépend du travail effectué par cette instruction, c'est-à-dire essentiellement de la catégorie à laquelle appartient l'instruction (voir ci-dessus). Par exemple, une instruction d'addition a besoin d'au moins trois opérandes : les deux nombres à additionner et l'emplacement où le résultat sera stocké [15].

En pratique, l'instruction n'utilise pas directement les deux nombres à additionner mais les registres ou les adresses de la zone de mémoire principale où sont stockés ces deux nombres. On distingue d'ailleurs plusieurs stratégies pour les jeux d'instructions selon qu'ils effectuent les calculs directement à partir de données stockées en mémoire principale, à partir de données stockées dans les registres ou les deux à la fois. L'exemple suivant montre comment est effectuée l'opération consistant à lire les valeurs des données B et C stockées en mémoire principale, à additionner ces deux valeurs et à stocker le résultat dans A. C'est ce que symboliquement nous écririons :

$$A = B + C$$

- Jeu d'instructions de type mémoire-mémoire :
  1. Addition et stockage direct (`Add B C A`) ;
- Jeu d'instructions de type registre-registre :
  1. Chargement de B dans le registre R1 (`Load R1 B`) ;
  2. Chargement de C dans le registre R2 (`Load R2 C`) ;
  3. Addition et stockage du résultat dans le registre R3 (`Add R1 R2 R3`) ;
  4. Stockage du résultat dans A (`Store R3 A`).

Le principe qui consiste à décomposer une instruction complexe, comme l'addition du modèle mémoire-mémoire, en une série d'instructions élémentaires, comme l'addition du modèle registre-registre (aussi appelé chargement-rangement), a donné naissance à des jeux d'instructions appelés RISC (Reduced Instruction Set Computer). Les autres jeux d'instructions sont alors nommés CISC (Complex Instruction Set Computer).

RISC est souvent traduit en français par « jeu réduit d'instructions ». Cette traduction reflète effectivement les conséquences historiques de l'apparition des RISC, mais ne correspond plus à la réalité : il y a aujourd'hui plus d'instructions dans un jeu d'instructions RISC que dans un CISC. En revanche, les instructions RISC sont toujours beaucoup simples que les instructions CISC et comportent moins de modes d'adressage. La bonne traduction est donc « jeu d'instructions réduites » pour RISC et « jeu d'instructions complexes » pour CISC.

Quel est l'intérêt des RISC ? Tout d'abord, l'utilisation d'instructions réduites permet de faire en sorte que toutes les instructions aient la même taille, ce qui facilite

15. Certains processeurs stockent le résultat au même endroit que l'un des deux nombres à ajouter et n'ont donc besoin que de deux opérandes.

grandement l'opération de décodage. Ensuite, cela favorise la mise en place des *pipeline*, c'est-à-dire des recouvrements des étapes d'exécution d'une intruction. Ces cinq étapes (*fetch, decode, load, execute et store*) peuvent être assurées par des composants différents et il est alors possible, lorsque deux instructions se suivent et lorsqu'elles ont la même taille, d'exécuter l'étape de *fetch* de la seconde instruction alors que la première en est à l'étape de *decode* [16]. Cela conduit au schéma de la figure 1.9.

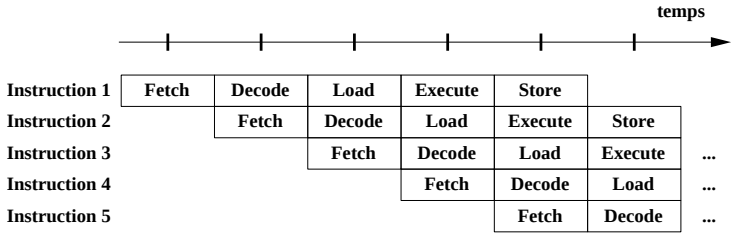


FIGURE 1.9 – *Le principe du* pipeline

En fait, une étape est exécutée à chaque cycle d'horloge et, donc, pour n instructions et un *pipeline* à cinq étages, il faudra $n+4$ cycles d'horloge pour toutes les exécuter, soit en moyenne $1+\frac{4}{n}$ cycle par instruction. Ainsi, pour une machine RISC, une instruction est en moyenne exécutée à chaque cycle d'horloge. Une autre façon de voir les choses est représentée sur la figure 1.10.

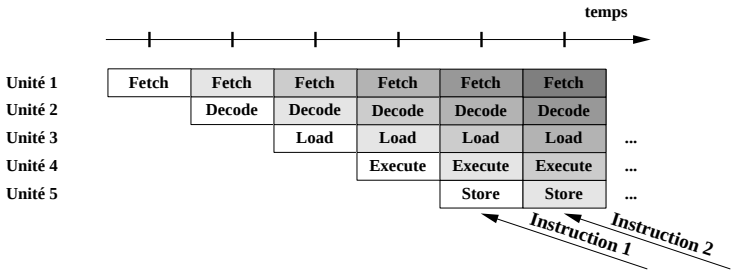


FIGURE 1.10 – *Le principe du* pipeline *vu de façon différente.*

16. Il existe des machines CISC avec *pipeline* mais leur mise en œuvre est plus complexe car il est difficile de prévoir la fin d'une instruction.

1.5 Rôle de l'unité de gestion de mémoire

La protection de la mémoire est une des tâches principales des systèmes d'exploitation modernes. Cette protection est très difficile à réaliser sans l'intervention des couches matérielles de la machine et certains composants, comme la MMU (*Memory Management Unit*), jouent un rôle capital dans cette protection.

Inversement, il est difficile de parler de la protection de la mémoire sans faire référence à des services particuliers du système d'exploitation. Ce chapitre étant consacré uniquement au matériel, nous resterons le plus vague possible en ce qui concerne le rôle réel de la MMU et un lecteur curieux pourra trouver des explications plus concrètes dans le chapitre 6 consacré à la gestion de la mémoire.

Adresses physiques

Nous avons défini à la section 1.2 le terme « adresse » comme une indexation des octets disponibles dans la mémoire principale de l'ordinateur. Nous appellerons désormais ce type d'adresse des « adresses physiques » dans la mesure où elles font directement référence à la réalité physique de la machine. Une adresse physique est donc un nombre qu'il suffit de passer au contrôleur de mémoire pour désigner une donnée stockée en mémoire.

Le nombre d'adresses physiques différentes dépend de la taille des adresses : une adresse de 8 bits permet de coder $2^8 = 256$ zones d'un octet, soit 256 octets ; une adresse de 32 bits permet de coder $2^{32} = 2^2 \times 2^{10} \times 2^{10} \times 2^{10} = 4$ Go. Pendant longtemps, la taille des bus d'adresses ne permettait pas de passer une adresse en une seule fois et diverses méthodes étaient utilisées pour recomposer l'adresse à partir de plusieurs octets. Aujourd'hui les bus d'adresses ont une taille de 32 ou 64 bits, ce qui permet de passer au moins une adresse à chaque fois.

Protection de la mémoire

Il est très difficile de protéger des zones de la mémoire en travaillant directement sur les adresses physiques des données contenues dans ces zones [17]. Supposons en effet que nous souhaitons écrire un système d'exploitation qui protège les données utilisées par différentes tâches. Tout d'abord, il est clair qu'il est plus facile de protéger ces données si elles sont rassemblées en lots cohérents (toutes les données de la tâche *A*, puis toutes les données de la tâche *B*, etc.) que si elles sont entrelacées (une donnée de la tâche *A*, une donnée de la tâche *B*, une donnée de la tâche *A*, etc.).

Ensuite, si les données sont entrelacées, le système devra pour chaque donnée marquer le propriétaire de la donnée et les droits d'accès à cette donnée (voir figure 1.11). Par exemple, il faudra préciser que certaines données du système d'exploitation sont inaccessibles pour les autres tâches. Imaginons qu'au moins 2 bits soient nécessaires

17. Voir aussi le chapitre sur la gestion de la mémoire.

pour marquer chaque donnée. Cela signifie donc que pour 8 bits alloués en mémoire, 2 bits supplémentaires devront être utilisés. Par ailleurs, il faut aussi marquer ces 2 bits supplémentaires... Si on suppose que ces marquages sont regroupés quatre par quatre, cela veut dire que pour 3 octets alloués en mémoire, le système d'exploitation a besoin de 4 octets.

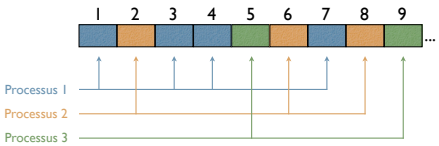


FIGURE 1.11 – *La protection de données entrelacées n'est guère efficace*

Cette politique de protection n'est donc pas réaliste, puisqu'elle entraîne un gâchis considérable de la place mémoire. Comme nous le disions plus haut, la mémoire a longtemps été une denrée rare et tous les développeurs de systèmes d'exploitation ont rapidement cherché une solution.

Adresses virtuelles

En pratique, les ordinateurs n'ont jamais 4 Go de mémoire principale (adresses sur 32 bits) et une grande partie des adresses ne sert à rien. Une idée intéressante est d'utiliser tout l'espace d'adressage pour effectuer la gestion de la mémoire, puis de transformer au dernier moment l'adresse utilisée en une adresse physique.

Par exemple, utilisons les 2 premiers bits d'adresse pour réserver des zones de mémoire au système. Ces 2 bits désignent 4 zones de 1 Go et nous allons réserver la zone 3 (11 en binaire) pour le système. Ainsi, toutes les données concernant le système d'exploitation auront une adresse de 32 bits débutant par 11 et il suffira donc de tester ces 2 premiers bits pour savoir si une donnée appartient au système ou pas.

L'avantage de ce système est évident ! Mais la correspondance entre les adresses ainsi composées et les adresses physiques n'est plus assurée : les adresses de 32 bits commençant par 11 correspondent à des données stockées après les 3 premiers Go de mémoire... De telles adresses n'ayant pas de correspondance physique s'appellent des adresses virtuelles.

Une adresse virtuelle est donc une représentation particulière de la zone occupée par un octet qui nécessite une transformation adaptée pour correspondre à une adresse physique. Cette transformation s'appelle la traduction d'adresses ou, par abus de langage, translation d'adresses.

Traduction d'adresses et TLB

La conversion des adresses virtuelles en adresses physiques connues par les couches matérielles a lieu constamment pendant l'exécution des différents programmes utilisés sur l'ordinateur. Cette conversion doit donc être très rapide car elle conditionne la vitesse d'exécution des programmes. C'est pourquoi cette traduction est directement assurée par un composant électronique logé dans le processeur : la MMU [18].

Le système d'exploitation n'intervient donc pas au moment de la traduction d'adresses. Néanmoins, la MMU est programmée par le système d'exploitation et c'est ce dernier, notamment, qui lui indique comment sont composées les adresses et qui met à jour la table de traduction.

Ce procédé est encore un exemple montrant à quel point la conception des systèmes d'exploitation et la conception des architectures des ordinateurs sont liées.

Il est à noter que, outre la MMU, d'autres composants matériels interviennent dans la traduction d'adresse, comme par exemple des mémoires tampon de traduction anticipé (*Translocation Lookaside Buffer* ou TLB) et un lecteur désireux d'en savoir plus pourra se reporter à des ouvrages spécialisés sur l'architecture des ordinateurs (voir bibliographie de ce document, par exemple).

1.6 Performances des ordinateurs

Cette section sur les performances n'est pas directement liée à l'étude des architectures des machines, ni à l'étude des systèmes d'exploitation. Néanmoins, les systèmes d'exploitation interviennent aussi dans la mesure des performances des ordinateurs et nous (r)établirons ici quelques vérités importantes pour le choix d'un ordinateur.

Le fond du problème

Les constructeurs d'ordinateurs s'affrontent généralement sur la place publique en annonçant des performances toujours plus élevées. Outre le fait que la notion de performance masque les problèmes et les innovations des architectures des ordinateurs, elle est très ambiguë et chaque constructeur joue sur cette ambiguïté pour proposer des tests de rapidité (*benchmarks* en anglais, parfois abrégés en *benchs*) qui avantagent leurs choix d'architecture ou leurs spécificités matérielles.

Nous allons dans cette section essayer de faire le point sur les mesures proposées par les constructeurs et, comme d'habitude, essayer de séparer le bon grain de l'ivraie. Comme pour les sections précédentes, un lecteur désireux d'approfondir ces questions de mesure de performances doit se reporter à des ouvrages spécialisés.

Tout d'abord, il est nécessaire de bien définir ce que nous voulons mesurer. Deux approches sont possibles : celle de l'utilisateur ou celle du centre de calcul.

18. La MMU n'a pas toujours été logée directement dans le processeur, mais c'est désormais le cas.

L'utilisateur d'un ordinateur est intéressé par le temps que mettra un programme pour s'exécuter sur son ordinateur. Pour lui, ce temps n'a cependant pas de sens a priori car il est difficile d'estimer la durée qu'une exécution devrait avoir et il est d'ailleurs très rare de voir un utilisateur mesurer précisément le temps d'exécution d'un programme. En revanche, les différences entre ordinateurs sont parfaitement perceptibles pour un utilisateur car il possède une notion très précise du temps de réponse usuel de son ordinateur (on parle parfois de réactivité) : s'il utilise un tout nouvel ordinateur et si son programme s'exécute plus rapidement sur cet ordinateur, l'utilisateur se rend compte immédiatement de l'accélération ! C'est généralement cette perception de l'accélération de l'exécution d'un programme qui permet de dire qu'un ordinateur est « plus puissant » qu'un autre.

Le centre de calcul n'est en revanche pas intéressé par les capacités isolées des ordinateurs qu'il possède. Il a une vision globale des performances : sur l'ensemble des ordinateurs du centre et pour un temps donné, combien de programmes ont pu être exécutés ? Cette vision des choses comporte (au moins !) deux mérites : d'une part, seule une moyenne des performances des ordinateurs est prise en compte, d'autre part, l'enchaînement des exécutions est aussi mesuré. La moyenne des performances est probablement une mesure plus juste car il est facile d'imaginer qu'un ordinateur puisse être très rapide pour certaines tâches et très lent pour d'autres. Mesurer ses performances pour une seule tâche reviendrait donc à avantager un type de programme. L'enchaînement des exécutions est aussi très important à mesurer : on pourrait imaginer un ordinateur qui utiliserait de façon très efficace les couches matérielles dont il dispose, mais qui, en contrepartie, rendrait les phases de chargement des programmes ou de lecture des données très longues et très pénibles. Les systèmes d'exploitation (chargés de ces deux services) interviennent donc aussi dans cette partie de l'informatique et il paraît douteux de vouloir mesurer les performances d'un ordinateur sans préciser le système d'exploitation utilisé pour les tests de rapidité.

Remarquons qu'un utilisateur peut être considéré comme un centre de calcul ne disposant que d'un ordinateur : il est très rare de nos jours qu'un utilisateur n'exécute qu'un programme à la fois, même s'il existe encore des système d'exploitation primitifs n'offrant que cette possibilité ! Nous ne ferons donc plus dans le reste du texte de différence entre l'utilisateur et le centre de calcul. Remarquons aussi que les principes énoncés ci-dessus s'appliquent uniquement à des programmes qui ont une durée d'exécution limitée, comme par exemple des programmes effectuant des calculs scientifiques. Ainsi, par exemple, il paraît difficile de mesurer le temps d'exécution d'un éditeur de texte ou d'un programme de jeu !

Nous pouvons donc au regard de ces quelques remarques distinguer plusieurs mesures possibles.

La mesure des performances d'un processeur : cette mesure intéresse a priori le constructeur d'ordinateurs car elle lui permet de choisir un processeur adapté aux besoins de ses clients. Notez que c'est malheureusement souvent cette mesure qui sert d'argument publicitaire

pour la vente d'ordinateurs...

La mesure des performances d'une architecture d'ordinateur : cette mesure intéresse assez peu de personnes et elle représente en fait la capacité du constructeur d'ordinateurs à intégrer efficacement les composants d'un ordinateur autour du processeur. Cette mesure devrait tenir compte des performances du bus de communication, des performances des zones de mémoire, du disque dur, etc. Elle donne en fait la mesure de la performance maximale d'un ordinateur, quels que soient les moyens utilisés pour la mesurer.

La mesure des performances d'un ordinateur : cette mesure n'a aucun sens !

La mesure des performances d'un couple (ordinateur, OS) : cette mesure intéresse en premier lieu l'utilisateur car elle représente directement la performance qu'il peut attendre de sa machine et car elle correspond à la sensation de rapidité (ou de lenteur) qu'a l'utilisateur devant son ordinateur. Insistons sur le fait que le système d'exploitation intervient dans cette mesure et que, donc, de mauvais résultats concernant cette mesure ne signifient pas nécessairement que l'ordinateur ou les programmes utilisés sont mauvais.

Il n'est pas rare de voir des constructeurs proposer des ordinateurs ayant d'excellentes mesures des performances de leur architecture, mais incapables de fournir le système d'exploitation et le compilateur qui sauront tirer partie de cette machine. Cela revient à équiper une voiture d'un moteur de formule 1 en oubliant de fournir la boîte de vitesses : certes le moteur tournera très vite et sera très performant, mais même à fond en première, il est probable que ça n'aille pas très vite !

Temps utilisateur *versus* temps système

Si nous chronométrons l'exécution d'un programme, nous mesurons en fait plusieurs choses à la fois : le temps que l'ordinateur a effectivement mis pour effectuer les calculs demandés, le temps utilisé par le système d'exploitation pour rendre les services nécessaires à l'exécution du programme et le temps perdu à attendre des entrées / sorties comme par exemple des lectures sur le disque dur ou des affichages à l'écran. Comme évoqué plus haut, ce qui compte pour l'utilisateur, c'est la somme des temps pris par ces trois entités, soit effectivement le temps global.

Les choses se compliquent cependant lorsque plusieurs programmes s'exécutent en concurrence sur une même machine : comment mesurer le temps utilisé par un seul des programmes ? Outre ce problème, il faut aussi pouvoir distinguer dans le temps utilisé par le système d'exploitation le temps effectivement employé pour répondre à des besoins du programme dont on cherche à mesurer l'exécution. Ces considérations ont entériné la distinction entre le temps utilisateur, c'est-à-dire le temps effectivement

utilisé pour les calculs, et le temps système, c'est-à-dire le temps employé par le système d'exploitation pour servir le programme testé.

Malheureusement, cette distinction sous-entend que le temps utilisateur ne dépend pas du système d'exploitation, ce qui est en général complètement faux : le programme qui s'exécute est obtenu par compilation d'un code source écrit dans un langage de haut niveau (comme le C) et l'efficacité de ce programme dépend non seulement de la façon dont il a été compilé, mais aussi de la façon dont les appels système ont été programmés (par le système d'exploitation). Disposer du temps utilisateur d'un programme permet néanmoins aux développeurs d'améliorer le code source de ce programme. Cependant, cela ne veut pas dire qu'il n'est pas possible de diminuer le temps système en modifiant les sources...

Par ailleurs, et nous insistons sur ce fait, ce qui intéresse l'utilisateur, c'est la durée d'exécution d'un programme comprenant donc le temps utilisateur, le temps système et le temps utilisés pour effectuer des entrées / sorties. Les constructeurs d'ordinateur et les vendeurs de logiciel ont cependant vu dans cette distinction le moyen de s'affranchir des performances des systèmes d'exploitation et ont donc proposé des mesures s'appuyant uniquement sur le temps utilisateur, ce qui n'a pas de sens. Pour reprendre l'analogie avec les voitures, supposons que nous voulons mesurer le temps que met une voiture pour atteindre la vitesse de 100 km/h à partir d'un départ arrêté. Mesurer le temps utilisateur revient à mesurer le temps total de l'action auquel nous soustrairons le temps de réaction du conducteur (entre le moment où le départ a été donné et le moment où le conducteur a accéléré), le temps des changements de vitesse et le temps pendant lequel les pneus ont patiné. Notons que cette mesure favorise étrangement les voitures qui ont de mauvais conducteurs, de mauvais pneus et de mauvaises boîtes de vitesses...

Les unités de mesure

Nous allons ici décrire quelques unités utilisées par les constructeurs pour mesurer les performances des ordinateurs. Comme nous l'avons déjà mis en avant, ces mesure ne reflètent pas la puissance d'un ordinateur et de son système d'exploitation car elles se fondent généralement sur le temps utilisateur.

L'unité de mesure la plus courante a longtemps été le MIPS [19] (millions d'instructions par seconde). Cette mesure varie généralement de pair avec la puissance des ordinateurs, en tout cas avec la puissance ressentie par les utilisateur : plus une machine est puissante, plus elle affiche de MIPS. Ceci explique probablement pourquoi cette mesure est si populaire... Néanmoins, elle n'a aucun sens et il est facile de trouver des contre-exemples flagrants. Sans entrer dans les détails, remarquons deux faits majeurs :

1. pour une même opération, le nombre d'instructions nécessaires varie d'une machine à l'autre ;

19. À ne pas confondre avec le fabricant de microprocesseurs MIPS, racheté par la société Silicon Graphics Incorporated (SGI).

2. certaines instructions complexes demandent beaucoup plus de temps que d'autres instructions simples.

Par exemple, les instructions faisant appel au coprocesseur sont généralement très coûteuses en temps. Pour effectuer un calcul flottant, il est cependant bien clair qu'il vaut mieux faire un appel au coprocesseur plutôt que d'exécuter des dizaines d'instructions entières. Donc l'unité MIPS est totalement inadaptée, en particulier pour les calculs utilisant les flottants. On trouve souvent des traduction de MIPS qui reflètent ce phénomène : « Meaningless Indication of Processor Speed » ou « Meaningless Indoctrination by Pushy Salespersons ».

Afin de prendre en compte les calculs utilisant des flottants, le FLOPS (Flotting point Operation Per Second) a vu le jour. On parlait au début de MégaFLOPS, puis de GigaFLOPS et certaines machines affichent désormais des performances en Téra-FLOPS. Cette unité est plus astucieuse que le MIPS : d'une part, elle est spécialisée dans un type d'applications (les calculs flottants) et, d'autre part, elle ne compte plus le nombre d'instructions, mais le nombre d'opérations par seconde ce qui lui permet de s'abstraire un tant soit peu des couches matérielles. Les critiques faites sur les MIPS restent néanmoins valables pour les FLOPS et, en particulier, certaines opérations flottantes, comme les additions, sont beaucoup plus rapides que d'autres, comme les divisions.

Deux programmes particuliers sont beaucoup utilisés par les fabricants de PC pour mesurer les performances de leurs machines : le Dhrystone et le Whetstone. Le programme Dhrystone n'utilise pas d'opérations flottantes et cette mesure est donc peu utile. Le programme Whetstone est écrit en FORTRAN et est compilé sans optimisation. Il est alors raisonnable de se poser la question suivante : les valeurs seraient-elles du même ordre avec des programmes écrits en C ou en un autre langage de haut niveau ? Il est probable que non.

Les résultats du Dhrystone et du Whetstone se mesurent en MIPS, mais ces deux noms sont souvent directement utilisés comme une unité.

Notons que ces deux unités furent surtout utilisées par les fabricants de PC parce que le système d'exploitation utilisé (`MS-DOS`) était si rudimentaire (mono-tâche et mono-utilisateur) qu'il supprimait *ipso facto* les problèmes de répartition entre temps utilisateur et temps système. Néanmoins, ces mesure tiennent compte des entrées / sorties, ce qui est déjà un bel effort.

Notons aussi que l'utilisation du Dhrystone est un bon exemple de démarche commerciale ingénieuse : pendant longtemps les processeurs Intel obtenaient de mauvais résultats pour les calculs flottants alors qu'ils excellaient dans les calculs entiers. Les constructeurs de processeurs pour station de travail (comme SUN, DEC, MIPS, Motorola) affichaient eux des performances opposées : excellentes pour les calculs flottants, mais médiocres pour les calculs entiers. Promouvoir l'unité Dhrystone revenait donc à éradiquer la concurrence...

Le tableau 1.2 indique les résultats obtenus par des processeurs assez anciens mais il permet d'illustrer simplement cette section. On peut ainsi remarquer que le

Modèle	Cadence (MHz)	Mémoire (Mo)	OS	D (MIPS)	W (MIPS)	M (S)
Sun Sparc 10	100	48	Solaris	156	122	11,0
Sun Sparc 20	75	32	Solaris	172	137	11,0
MIPS R5000	200	64	IRIX 5.3	214	110	8,6
Pentium 120	120	32	Solaris	120	60	10,2
Pentium 120	120	24	NT	186	70	10,0
Sun Ultra 170	167	128	Solaris	284	152	5,4
Sun Ultra 170	167	128	Solaris	332	188	2,9

TABLE 1.2 – *Quelques performances de stations Unix et NT. Les trois dernières colonnes référencent les résultats des tests Dhrystone, Whetstone et Maxwell.*

Pentium 120 obtient d'excellentes notes sous Windows NT au test Dhrystone, mais des notes dans la moyenne pour le test Maxwell. De même, on peut voir que le MIPS R5000 obtient suivant les test de très bonnes notes (Dhrystone et Maxwell) ou de très mauvaises (Whetstone). Enfin, les deux dernières lignes du tableau ont été réalisées grâce à deux compilateurs différents et on peut constater que le résultat s'en ressent.

De nombreuse unités sur le même principe apparaissent régulièrement. Voici par exemple trois unités utilisées par un magazine sur les PC : les Winstone, les Winmark ou les CPUmark. A priori, leur seul intérêt est de perdre le client en proposant systématiquement une unité bien adaptée à la machine à vendre. La mode actuelle est de promouvoir les unités de calcul fondées sur le rendu d'images ou de séquences 3D utilisant un ou plusieurs cœurs. Il est toutefois évident pour le lecteur que le meilleur processeur affublé d'une quantité faible de mémoire ou d'un bus système à la vitesse réduite se comportera comme une voiture de course sur laquelle auraient été greffé un train de pneus de mobylette ainsi que les freins d'un vélo. Comment dès lors accepter de participer à ces concours que l'on voit fleurir sur la toile [20].

La voie de la sagesse ?

Une unité plus intéressante a vu récemment le jour : le SPEC (System Performance Evaluation Cooperative). Cette unité est soutenue par un certain nombre de constructeurs et prend en compte l'intervention du système d'exploitation et du compilateur dans la mesure des performances. Sa philosophie est la suivante :

> La méthodologie basique du SPEC est de fournir aux personnes désireuses de réaliser des tests de performances une suite standardisée de codes

20. Les noms, et souvent l'orthographe, sont exemplaires de cours de maternelle : *Venez faire chauffer votre CPU multicore : Cinebench R10* – Comme dans tout bench, le but est d'atteindre le plus gros score possible, toutefois, sachez que je ne prendrai en compte que le score cpu. Bien entendu, les processeurs seront classé par nombre de core, un dual core ne pouvant évidement pas rivaliser contre un quad core pour le temps de rendu (sauf à lui coller un fréquence de brute).

> sources fondés sur des applications existantes et portées sur une large variété de plateformes. La personne en charge des mesures prend ce code source, le compile pour son système et peut optimiser le système testé pour obtenir les meilleurs résultats. Cette utilisation de code connu et ouvert réduit grandement le problème des comparaisons entre oranges et pommes [21].

Les codes sources mentionnés sont par exemple : `perl`, `gcc`, `bzip2`, `h264ref`,...

Il apparaît souvent sous deux formes, les SPECint et les SPECflop, faisant ainsi référence à des programmes utilisant ou pas les calculs flottants. Cette unité ayant évolué au fil des ans, il est aussi habituel de spécifier la version utilisée en accolant l'année de création : par exemple, des SPECint95.

Un grand pas a été franchi avec cette unité car, d'une part, le rôle du système d'exploitation est reconnu et, d'autre part, les programmes utilisés sont proches des préoccupations des utilisateurs. Cette unité n'est néanmoins pas parfaite et, en particulier, elle s'accommodait très mal des ordinateur multi-processeurs. Ce n'est plus le cas dans la mesure où elle différencie même les résultats selon les catégories cœurs par processeurs et nombre de processeurs.

Conclusion sur les performances

Nous tirerons deux conclusions de cette courte étude des mesures des performances des ordinateurs.

Tout d'abord, ces mesures ne riment à rien et il est très clair que certains constructeurs n'hésitent pas à modifier l'architecture de leurs machines pour que celles-ci obtiennent de meilleurs résultats. En quelque sorte, ils optimisent l'architecture de la machine en tentant de maximiser la mesure obtenue. Cette course à la mesure des performances va à l'encontre du progrès informatique car les programmes de test ne reflètent généralement pas les préoccupations des utilisateurs et les constructeurs optimisent donc leurs machines pour des tâches qui n'intéressent personne.

Ensuite, si un ingénieur a besoin d'exécuter un programme donné en un temps donné et s'il décide d'acheter un ordinateur adapté à ce besoin, il est illusoire de se fonder sur les performances affichées par les constructeurs pour choisir ce nouvel ordinateur. La bonne démarche consiste à tester la machine avec le programme en question ! Contrairement à ce que l'on pourrait croire, les constructeurs prêtent volontiers leurs machines à des industriels ou à des laboratoires de recherche pour que ces derniers les testent et il serait stupide de ne pas profiter de telles occasions.

1.7 Conclusion : que retenir de ce chapitre ?

Ce chapitre assez long a permis de faire le point sur l'architecture des ordinateurs et sur ses relations étroites avec les systèmes d'exploitation. Il est important de retenir

21. apples-to-oranges comparizons – NDLT.

les points suivants.

Sur l'architecture des ordinateurs :

- les composants des ordinateurs communiquent par des interruptions qui représentent une forme de programmation bas-niveau ;
- les processeurs ont deux modes de fonctionnement (mode noyau et mode utilisateur) qui permettent aux systèmes d'exploitation de protéger fondamentalement les ressources de la machine (notamment les données des utilisateurs) ;
- plusieurs zones de mémoire (mémoire secondaire, mémoire principale, cache externe, cache interne, registres) de caractéristiques très différentes sont utilisées par les ordinateurs pour effectuer leur travail et ces zones sont organisées sous la forme d'une hiérarchie de mémoire.

Sur la facilité de portage des développements :

- le jeu d'instructions, spécifique à chaque processeur, est la couche logicielle ultime ;
- tout programme écrit grâce au jeu d'instruction est spécifique de l'ordinateur pour lequel il a été écrit (processeur et périphérique) et ne peut donc pas être réutilisé sur d'autres ordinateurs.

Sur les relations entre l'architecture des ordinateurs et les systèmes d'exploitation :

- les systèmes d'exploitation et les ordinateurs ont évolué ensemble (rôle de la MMU, par exemple), chacun permettant à l'autre de progresser ;
- la protection des ressources n'a de sens que si elle est effectuée au plus profond de l'ordinateur, c'est-à-dire si elle est effectuée par le matériel.

2

Qu'est-ce qu'un système d'exploitation ?

Nous avons vu dans le chapitre précédent que la programmation d'un ordinateur peut être très complexe si elle s'effectue directement à partir des couches matérielles. L'utilisation d'un langage de type assembleur permet de s'affranchir du jeu d'instructions, mais la programmation reste néanmoins très délicate. Par ailleurs, tous les développements faits en assembleur ne sont valables que pour une seule machine.

Un autre inconvénient de la programmation de bas niveau est qu'elle permet de tout faire sur un ordinateur, y compris des opérations illicites pour les périphériques qui peuvent entraîner leur dégradation ou tout simplement la perte irrémédiable de données importantes.

Le système d'exploitation d'un ordinateur est un programme qui assure toutes les tâches relevant des ces deux aspects, c'est-à-dire que, d'une part, il facilite l'accès aux ressources de la machine tout en assurant une certaine portabilité des développements et, d'autre part, il contrôle que tous ces accès sont licites en protégeant ses propres données, celles des utilisateurs et celles de toutes les tâches s'exécutant sur la machine.

Mots clés de ce chapitre : machine virtuelle, ressources, gestionnaire de ressources, CPU, appel système, système monolithique, système en couches, noyau, micro-noyau, services.

2.1 Définitions et conséquences

Les différents ouvrages traitant du sujet ne sont pas d'accord sur la définition à donner d'un système d'exploitation. Nous ne donnerons donc ici des définitions formelles que pour éclaircir les idées des lecteurs, mais ces définitions importent peu en elles-mêmes.

En revanche, il est important de bien saisir les deux approches différentes des systèmes d'exploitation, quelle que soit la définition qu'on en donne : d'une part, faciliter l'accès à la machine et, d'autre part, contrôler cet accès.

Le système d'exploitation est une machine virtuelle

Le système d'exploitation est une machine virtuelle plus simple à programmer que la machine réelle. Il offre une interface de programmation à l'utilisateur qui n'a donc pas besoin de connaître le fonctionnement réel de la machine : l'utilisateur demande au système d'effectuer certaines tâches et le système se charge ensuite de dialoguer avec la machine pour réaliser les tâches qui lui sont demandées.

Dans cette optique, la machine réelle est en fait cachée sous le système d'exploitation et ce dernier permet donc un dialogue abstrait entre la machine et l'utilisateur. Par exemple, l'utilisateur se contente de dire « je veux écrire mes données dans un fichier sur le disque dur » sans se préoccuper de l'endroit exact où se trouvent ces données, ni de le façon dont il faut s'y prendre pour les copier sur le disque dur.

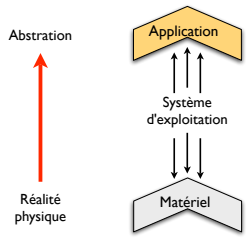


FIGURE 2.1 – *Le système d'exploitation permet un dialogue abstrait entre la machine et l'utilisateur.*

Notons qu'à cet effet le système d'exploitation se doit de proposer une interface permettant ce dialogue abstrait et c'est pour cela que des concepts dénués de toute réalité physique sont généralement utilisés lorsqu'on parle des systèmes d'exploitation. Citons par exemple les variables qui permettent de faire référence à des valeurs stockées quelque part en mémoire, les fichiers qui représentent l'abstraction du stockage de données ou les processus qui sont l'abstraction d'un programme en cours d'exécution.

Une partie de ce document est d'ailleurs dédiée à l'explication de ces concepts.

Pour reprendre l'analogie automobile que nous avons déjà utilisée dans le chapitre précédent, nous dirions que le système d'exploitation en tant que machine virtuelle est ce qui relie les éléments mécaniques de la voiture au conducteur. Il propose une interface plus ou moins standard (le volant, les pédales d'accélérateur, de frein et d'embrayage, etc.) qui permet au conducteur de dialoguer avec le moteur et d'obtenir un travail de ce dernier, sans pour autant se soucier du fonctionnement réel de ce moteur.

Le système d'exploitation est un gestionnaire de ressources

Les ressources d'une machine sont tous les composants qui sont utilisés pour effectuer un travail et qu'un utilisateur de la machine pourrait s'approprier. À ce titre, tous les périphériques comme la mémoire ou les disques durs sont des ressources. Les registres du processeur ou le temps passé par le processeur à faire des calculs [1] sont aussi des ressources.

Le système d'exploitation est un gestionnaire de ressources, c'est-à-dire qu'il contrôle l'accès à toutes les ressources de la machine, l'attribution de ces ressources aux différents utilisateurs de la machine et la libération de ces ressources quand elles ne sont plus utilisées.

Ce contrôle est capital lorsque le système permet l'exécution de plusieurs programmes en même temps [2] ou l'utilisation de la machine par plusieurs utilisateurs à la fois. En particulier, il doit veiller à ce qu'un utilisateur ne puisse pas effacer les fichiers d'un autre utilisateur ou à ce qu'un programme en cours d'exécution ne détruise pas les données d'un autre programme stockées en mémoire.

Un autre aspect capital du contrôle des ressources est la gestion des conflits qui peuvent se produire quand plusieurs programmes souhaitent accéder en même temps aux même données. Supposons par exemple qu'un utilisateur exécute deux programmes, chacun d'eux écrivant dans le même fichier. Il y a de grandes chances que, si aucune précaution n'est prise, le résultat contenu dans le fichier ne soit pas celui escompté.

Si nous nous référons à notre analogie automobile, le système d'exploitation en tant que gestionnaire de ressources est représenté par le limiteur de vitesse qui oblige le conducteur à conduire sagement ou par le système d'arrêt automatique en cas d'endormissement qui empêche le conducteur endormi d'entrer en collision avec un mur.

Commodité versus efficacité

Les deux définitions ci-dessus adoptent deux points de vue diamétralement opposé et le grand dilemme dans le développement des systèmes d'exploitation a été le choix entre la commodité et l'efficacité.

1. On parle généralement de temps CPU (*Central Processing Unit*).
2. Nous détaillerons plus loin ce qu'il faut comprendre exactement par ces termes.

Certes il est agréable de disposer d'un ordinateur commode, mais il ne faut pas oublier que les ordinateurs sont de plus en plus fréquemment utilisés pour effectuer des tâches critiques. Ainsi, autant il peut être acceptable qu'un ordinateur individuel « plante » de temps en temps, autant il est rassurant de savoir que les ordinateurs contrôlant les avions ou les opérations de chirurgie assistée sont d'une stabilité à toute épreuve, quitte à ce que cela soit au détriment d'un commodité d'utilisation.

Unix et Windows sont deux archétypes de système ayant choisi chacun un des aspect au détriment de l'autre (au moins à l'origine) :

- sous Unix, tout est interdit par défaut et il faut explicitement autoriser les différentes actions (se connecter, avoir accès à un fichier, etc.) ; ceci permet aux systèmes Unix d'être très stable mais rend ces systèmes d'un abord difficile ;
- sous Windows, tout est autorisé par défaut et il faut explicitement interdire ce qui doit l'être ; ceci permet un accès facile aux systèmes Windows mais rend très difficile leur sécurisation et leur stabilisation.

Il convient donc, d'une part, de garder ce dilemme à l'esprit pour effectuer des choix raisonnables en fonction des circonstances, d'autre part, d'œuvrer pour le développement de systèmes d'exploitation qui soient à la fois commodes et efficaces.

Mode utilisateur versus mode noyau

Nous avons aussi vu dans le chapitre précédent que les processeurs fonctionnent dans deux modes différents : le mode noyau où toutes les instructions sont autorisées et le mode utilisateur où certaines instructions sont interdites.

Le système d'exploitation peut utiliser cette propriété pour faciliter le contrôle qu'il exerce : il s'exécute en mode noyau alors que tous les autres programmes sont exécutés en mode utilisateur. Ces programmes utilisateur ont ainsi par essence des pouvoirs limités et certaines opérations leurs sont interdites.

Par exemple, en n'autorisant l'accès aux différentes ressources de la machine qu'aux programmes s'exécutant en mode noyau, le système d'exploitation protège ces ressources et contraint les programmes utilisateur à faire appel à lui (pas nécessairement de façon explicite) pour accéder aux ressources de la machine.

2.2 Les appels système

Sur les systèmes d'exploitation utilisant le mode noyau, tout programme utilisateur doit faire explicitement appel aux services du système d'exploitation pour accéder à certaines ressources. À ces fins, le système d'exploitation propose une interface de programmation, c'est-à-dire qu'il permet d'accéder à un certain nombre de fonctionnalités qu'il exécutera pour l'utilisateur. Les appels système sont l'interface proposée par le système d'exploitation pour accéder aux différentes ressources de la machine.

Par exemple, il est nécessaire de faire appel au système d'exploitation pour créer un fichier sur le disque dur et cela via un appel système comme ceux détaillés dans le

chapitre 11. Si nous demandons que ce fichier soit créé dans un répertoire qui nous est interdit d'accès, par exemple un répertoire appartenant à un autre utilisateur, l'appel système va refuser de créer le fichier.

Des appels système très nombreux

Le nombre d'appels système proposés varie suivant les systèmes d'exploitation et, pour un même type de système d'exploitation (Unix par exemple), ce nombre diffère suivant les versions (4.4BSD, Linux, Irix, etc.).

Ces appels reflètent les services que peut rendre le système d'exploitation et il sont généralement classés en quatre catégories :

- gestion des processus ;
- gestion des fichiers ;
- communication et stockage d'informations ;
- gestion des périphériques.

Notons au passage que les trois premières catégories correspondent à trois concepts sans réalité physique...

Le cas d'Unix

Sous Unix, les périphériques sont gérés comme de simples fichiers – ce qui représente l'abstraction absolue – et il n'y a donc que trois catégories d'appels système. Le manuel en ligne `man` [3] permet de connaître la liste des appels système et le but de chaque appel : tous les appels système se trouvent dans la section 2 du manuel.

Les appels système peuvent être directement utilisés dans un programme C et il faut savoir que la plupart des fonctions de la bibliothèque standard utilisent aussi ces appels système. La deuxième partie de ce document donnera de nombreux exemples d'utilisation des appels système.

Exécution d'un appel système

Un appel système provoque en fait une interruption logicielle et il suffit alors de programmer la machine pour que la routine correspondant à l'interruption fasse partie du système d'exploitation.

En pratique, il est fréquent que toutes les interruptions aient la même routine associée : le système d'exploitation. Suivant le type d'interruption (matérielle ou logicielle) et suivant la nature de l'interruption (erreur, travail terminé, etc.), le système décide alors quelle action il doit entreprendre.

Nous avons vu dans le chapitre précédent que la gestion des interruptions est assez complexe et, notamment, que la coexistence des interruptions logicielles avec les

3. Un bon moyen d'apprendre à utiliser la commande est de faire `man man`. On découvrira alors que les pages de manuel sont organisées en sections et le lecteur assidu apprendra certainement l'utilisation de la commande `man -k`.

interruptions matérielles pose de nombreux problèmes de conflit. En particulier, on peut se demander ce qui se passe lorsqu'une demande d'interruption matérielle a lieu au milieu d'une interruption logicielle. La réponse à cette question dépend non seulement de chaque système d'exploitation, mais aussi de l'implantation d'un même système sur plusieurs architectures différentes. Nous aborderons une partie de ce problème lors de l'étude des signaux dans le chapitre 14.

Nous avons vu aussi que les interruptions peuvent être synchrones ou asynchrones. Cela veut donc dire qu'un appel système peut donner lieu à une requête d'interruption qui sera traitée de façon asynchrone et que, par exemple, il se peut que le système ait d'autres choses plus importantes à faire. Néanmoins, du point de vue du programme qui exécute l'appel système, celui-ci se déroule de façon synchrone, c'est-à-dire que l'appel système interrompt l'exécution du programme et celui-ci ne peut pas reprendre son exécution tant que l'appel système n'est pas terminé.

Toutefois, et afin de ne pas trop pénaliser les programmes effectuant des appels système, le système d'exploitation utilise généralement des intermédiaires rapides et il rend la main au programme avant d'avoir effectivement accompli l'action demandée. Par exemple, si un programme demande l'écriture de données dans un fichier sur disque dur, le système d'exploitation va copier ces données dans une mémoire tampon qui lui appartient et va ensuite rendre la main au programme. Le système d'exploitation écrira ces données sur le disque dur à un autre moment.

La seconde partie du document devrait éclairer le lecteur sur l'utilisation et le rôle des appels système. Pour l'instant, il est capital de retenir les faits suivants :

- un appel système permet de demander au système d'exploitation d'effectuer certaines actions ;
- un appel système interrompt le programme qui l'exécute ;
- un appel système est exécuté en mode noyau même si le programme ayant demandé son exécution est exécuté en mode utilisateur.

L'utilisation des appels système illustre aussi un autre phénomène qui permet de développer des systèmes d'exploitation efficaces et cohérents : le meilleur moyen pour faire intervenir le système d'exploitation est de déclencher une interruption !

2.3 Structure d'un système d'exploitation

Noyau des systèmes d'exploitation

Le système d'exploitation d'une machine n'est en pratique pas constitué d'un seul programme. Il se compose d'au moins deux parties que l'on nomme souvent le noyau (*kernel* en anglais), qui est le cœur du système (*core* en anglais), et les programmes système. Le noyau est exécuté en mode noyau et il se peut qu'il corresponde à plusieurs processus différents (2 ou 3 sous les systèmes Unix, par exemple).

Même si le noyau effectue l'essentiel des tâches du système d'exploitation, ce dernier ne peut se réduire à son noyau : la plupart des services qu'un utilisateur utilise sur une machine sont en fait des sur-couches du noyau. Les programmes système qui

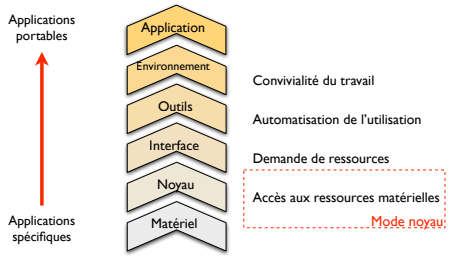


FIGURE 2.2 – *Représentation en couches des systèmes d'exploitation.*

assurent ces services sont néanmoins exécutés en mode utilisateur et, donc, ils doivent faire appel au noyau comme n'importe quel utilisateur pour accéder aux ressources.

Le modèle à couches

La relation entre la machine, le système d'exploitation et les programmes des utilisateurs est toujours représentée sous forme d'empilement d'un certain nombre de couches aux frontières bien définies : le système d'exploitation reliant la machine et l'utilisateur, il s'intercale naturellement entre les couches représentant ces derniers. La figure 2.2 montre un exemple d'une telle représentation.

En pratique, il est toujours difficile de distinguer où finit le système d'exploitation et où commence l'environnement utilisateur. Formulé différemment, il est toujours difficile de dire si un programme s'exécutant en mode utilisateur fait partie du système d'exploitation ou pas. En revanche, la limite entre le noyau et les programmes système est claire : le noyau est toujours exécuté en mode noyau !

L'interface proposée par le noyau est l'ensemble des appels système et, comme tout programme s'exécutant en mode utilisateur, les programmes système utilisent ces appels. Il est donc traditionnel d'intercaler, dans notre modèle à couches, les appels système entre le noyau et les programmes système. De même et pour les mêmes raisons, il est traditionnel d'intercaler le jeu d'instructions entre le matériel et le noyau.

L'objectif du modèle à couches est s'éloigner de plus en plus des spécificités de l'ordinateur et de son système d'exploitation pour s'approcher de plus en plus de l'abstraction fonctionnelle dont a réellement besoin l'utilisateur.

Sur les systèmes bien conçus, chaque couche s'appuie ainsi sur la couche précédente pour proposer de nouveaux services et il devient possible, par exemple, de transporter le même environnement de travail d'un système d'exploitation à un autre : il suffit pour cela que le « dessus » de la couche située en dessous soit le même.

Il est donc tout à fait imaginable de retrouver les mêmes outils, le même environnement et les mêmes applications fonctionnant sur deux systèmes d'exploitation différents. Cela suppose juste que ces systèmes soient bien faits. Rien sur le principe

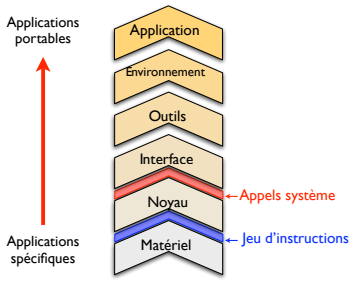


FIGURE 2.3 – *Représentation détaillée en couches des systèmes d'exploitation.*

ne permet donc d'expliquer pourquoi par exemple MS Word ne fonctionnerait pas sous Linux !

De la même manière, il n'y a aucune raison pour qu'un système d'exploitation impose un environnement de travail particulier et, sauf si ce système est mal fait, il devrait par exemple être possible de laisser à chaque utilisateur le choix de son environnement graphique, le choix de son compilateur, etc.

Le système d'exploitation n'intervient pas souvent

L'utilisation du modèle à couches pour représenter la structure des systèmes d'exploitation pourrait laisser croire qu'un système d'exploitation intervient systématique, à chaque accès aux ressources de la machine, et qu'il a donc besoin de beaucoup de temps CPU pour accomplir sa mission.

Par exemple, afin de contrôler que l'accès à toutes les ressources de la machine s'effectue dans de bonnes conditions, que ce soit pour l'allocation de mémoire ou de temps CPU, l'écriture de données, la lecture de données ou la libération d'une ressource, nous pourrions envisager de faire appel au système d'exploitation pour chaque requête d'accès. Ainsi un programme qui souhaiterait écrire en mémoire pour affecter la valeur 3.0 à la variable `x` devrait faire appel au système. Si ce programme effectue une boucle de 10 000 affectations, il faudrait alors faire 10 000 fois appel au système pour contrôler l'accès à la mémoire.

Ce n'est bien entend pas le cas et ce ne serait d'ailleurs par raisonnable de procéder ainsi : même si aujourd'hui ces machines ne coûtent plus très cher, il n'est pas envisageable de les employer essentiellement pour faire fonctionner des systèmes d'exploitation.

En d'autres termes, il ne faut pas que le système provoque une perte de puissance de la machine en voulant faciliter la vie des utilisateurs ou en cherchant à contrôler les accès aux ressources et il est indispensable de garantir la rapidité et l'efficacité du travail des systèmes d'exploitation.

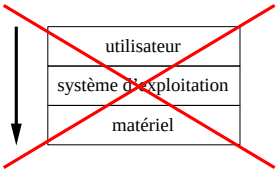


FIGURE 2.4 – *Le système d'exploitation contrôle les accès mais n'a pas besoin d'intervenir tout le temps.*

On pourrait alors songer à travailler très près des couches physiques de la machine, mais nous avons vu que ce n'est pas non plus une solution : certes cela permet de tirer pleinement partie de la puissance de l'ordinateur, mais tous les développements ainsi conçus ne sont valables que pour une machine donnée.

Une des clés pour un système d'exploitation réussi réside donc dans sa capacité à contrôler tous les accès aux ressources en intervenant le moins souvent possible de façon dynamique. Ceci est possible grâce à des composants qui contrôlent directement (i.e. sans intervention du système d'exploitation) que les accès sont licite en se fondant sur des autorisations d'accès gérées par le système d'exploitation.

Le principe est similaire à celui utilisé dans le métro pour la carte Orange :

– le guichetier de la station de Métro (qui représente le système d'exploitation) n'intervient qu'une seule fois par mois afin de valider les droits d'accès de l'usager, après avoir vérifié que ledit usager a bien versé la somme d'argent prévue ;
– chaque accès de l'usager au métro est contrôlé par des couches matérielles (les tourniquets d'entrée et de sortie des stations de métro), sans que l'usager n'ait besoin de solliciter le guichetier et d'attendre que celui-ci soit disponible.

La MMU dont nous avons parlé dans le chapitre précédent est un de ces composants : elle contrôle l'accès aux différentes zones de mémoire à partir de tables de traduction d'adresses et le système a juste besoin d'écrire de temps en temps dans ces tables pour assurer en permanence la protection de la mémoire entre les différentes tâches. Cet exemple sera développé dans le chapitre sur la gestion de la mémoire.

Pour le moment, il faut simplement garder à l'esprit qu'il existe des moyens pour que le système d'exploitation contrôle les accès aux ressources sans pour autant pénaliser le fonctionnement de la machine et que la représentation en couches permet de mettre en évidence les relations entre les différents composants des systèmes d'exploitation sans correspondre nécessairement à la réalité.

2.4 Les différents types de systèmes d'exploitation

Dans cette section, nous présentons les différentes structures des systèmes d'exploitation. Nous en profitons pour essayer de déterminer où se situe exactement un système d'exploitation dans les différentes couches qui composent l'environnement d'un ordinateur.

Les systèmes monolithiques

Le chapitre suivant retrace l'historique des systèmes d'exploitation et montre que ceux-ci ont été développés petit à petit. Bien souvent, les développeurs de tels systèmes reprenaient les travaux menés par les développeurs du système précédent et les complétaient. De génération en génération, le noyau des systèmes d'exploitation se mit donc à grossir et cela mena vite à des systèmes difficilement exploitables.

Ainsi, n'importe quelle procédure du noyau peut en appeler n'importe quelle autre et l'implantation de nouveaux services est très délicate : une erreur à un endroit du noyau peut entraîner un dysfonctionnement à un autre endroit qui, a priori, n'a rien à voir.

Ces inconvénients entraînèrent le développement de systèmes d'exploitation fondés sur d'autres modèles.

Les systèmes à couches

Les systèmes à couches appliquent au noyau le principe des couches tel que nous l'avons expliqué ci-dessus. L'idée consiste à charger chaque couche du noyau d'une tâche bien précise et à proposer une interface pour les couches au-dessus.

Chaque couche masque ainsi les couches situées en dessous d'elle et oblige la couche située au-dessus à utiliser l'interface qu'elle propose. L'avantage est évident : les couches sont totalement indépendantes les unes des autres et seule l'interface entre chaque couche compte. Cela permet de développer, de débuger et tester chaque couche en se fondant sur les couches inférieures qui sont sûres.

Même si ce principe est séduisant, il est très complexe à mettre en œuvre et les systèmes ainsi structurés sont souvent très lourds et peu performants.

Les micro-noyaux

Nous avons vu que les noyaux des systèmes monolithiques ont tendance à être volumineux. Par ailleurs, si un service particulier est très rarement utilisé, il doit néanmoins être présent dans le programme du noyau et il utilisera donc de la mémoire de façon inutile.

Pour éviter ces problèmes, une solution consiste à réduire le noyau à quelques procédures essentielles et à reporter tous les autres services dans des programmes système. Le noyau ainsi réduit s'appelle souvent micro-noyau.

Ce travail est néanmoins considérable car il suppose une refonte totale des systèmes d'exploitation habituels : ceux-ci ayant été développés au fil des ans, il se peut que des services très importants, mais apparus tardivement, soient implantés à la périphérie du noyau et que, donc, l'isolement de ces services demande la réécriture complète du noyau.

Un autre inconvénient des micro-noyaux réside dans la communication entre les différents services. Cette partie fondamentale permet le dialogue entre le noyau et les services (tels que le réseau, l'accès aux disques. Chaque service étant cloisonné dans son espace d'adresses, il est impératif d'établir ce mécanisme de communication entre le micro noyau et les services. Cette couche de dialogue, qui n'existe pas dans un noyau monolithique, ralentit les opérations et conduit à une baisse de performance.

Cela permet néanmoins de clarifier les choses et de concevoir des systèmes très généraux se fondant uniquement sur des services essentiels, comme le partage du temps. La plupart des systèmes à micro-noyau en profitent aussi pour inclure des services pour les machines multi-processeurs ou pour les réseaux de processeurs.

Les systèmes mixtes

Certains systèmes monolithiques tentent d'alléger leurs noyaux en adoptant quelques principes des systèmes à couches ou des micro-noyaux. Par exemple, le système d'exploitation Linux permet de charger de façon dynamique des modules particuliers qui sont agglomérés au noyau lorsque celui-ci en a besoin [4].

Ce procédé est assez pratique car, d'une part, il permet de ne pas réécrire le système d'exploitation et donc de profiter de l'expérience des systèmes Unix et, d'autre part, il minimise la mémoire utilisée par le noyau : les modules ne sont chargés que lorsque c'est nécessaire et ils sont déchargés quand le système n'en a plus besoin.

Le système d'exploitation Windows NT 4.0 est aussi un système mixte : son noyau (appelé Executif par Microsoft) contient un gestionnaire d'objet, un moniteur de références de sécurité, un gestionnaire de processus, une unité de gestion des appels de procédures locales, un gestionnaire de mémoire, un gestionnaire d'entrées / sorties et une autre partie que Microsoft nomme le noyau, probablement pour semer la confusion.

Dans les versions suivantes, les noyaux de Windows XP et de Windows Vista sont des noyaux hybrides. Celui de Vista, en raison du retard de développement, a évolué [5] vers un noyau à module. Une partie importante des pilotes matériels qui s'exécutaient auparavant en mode noyau sont maintenant relégués en mode utilisateur. Subsiste néanmoins peut-être un regret, le fait que l'interface graphique reste un des services du noyau.

4. Les systèmes NetBSD et FreeBSD offrent aussi cette possibilité.
5. Il s'agit de l'épisode surnommé « Longhorn Reset ».

2.5 Les services des systèmes d'exploitation

Pour terminer ce chapitre qui présente les systèmes d'exploitation, nous esquissons rapidement les différents travaux que doit effectuer un système d'exploitation moderne. Ces travaux sont généralement nommés « services » et ils seront détaillés dans les chapitres suivants. La plupart de ces travaux sont pris en charge par le noyau du système d'exploitation.

La gestion des processus La gestion des processus n'a de sens que sur les machines fonctionnant en temps partagé. Elle comprend la création et la destruction dynamique de processus. Le chapitre 5 est consacré à la gestion des processus et les chapitres 10 et 14 montrent comment il est possible de créer et de détruire des processus sous Unix.

La gestion de la mémoire Afin de simplifier la gestion des processus, les systèmes d'exploitation modernes travaillent dans un espace mémoire virtuel, c'est-à-dire avec des adresses virtuelles qui doivent être traduites pour correspondre à des adresses physiques.

Cela permet d'allouer à chaque processus (y compris au noyau) son propre espace mémoire de telle sorte qu'il a l'illusion d'avoir la machine pour lui tout seul.

La façon dont les processus utilisent la mémoire est décrite dans le chapitre 5 sur la gestion des processus.

Une autre partie de la gestion de la mémoire, très différente de celle présentée ci-dessus, concerne l'utilisation de pages de mémoire et la possibilité d'accéder à la mémoire secondaire pour optimiser la mémoire utilisée par tous les processus. Cette gestion est détaillée dans le chapitre 6.

La gestion des entrées / sorties Les entrées / sorties permettent de faire transiter des données par l'ordinateur et d'utiliser ces données pour faire des calculs. Ces données peuvent provenir de périphériques, de processus présents sur la machine ou de processus présents sur d'autres machines (via un réseau).

Nous discuterons brièvement des périphériques dans le chapitre 7 consacré au système de fichiers. Les chapitres 15 et 16 consacrés respectivement aux tuyaux et aux sockets sous Unix montreront un exemple d'utilisation des entrées / sorties.

Le système de fichiers Le système de fichiers est un élément essentiel des systèmes d'exploitation moderne : il permet d'accéder aux différents périphériques et il propose une interface abstraite pour manipuler des données.

Même si le système de fichiers fait souvent référence au disque dur, la notion de fichier est beaucoup plus générale et nous la détaillerons au chapitre 7.

La gestion des communications entre machines Il est aujourd'hui impensable de disposer d'ordinateurs à usage professionnel sans que ceux-ci soient reliés entre eux par un réseau local. Par ailleurs, l'utilisation de réseaux internationaux

comme l'Internet se répand et le système d'exploitation doit donc prendre en charge la gestion des communications par réseaux.

Comme nous l'avons annoncé en introduction, nous ne traiterons pas des réseaux dans ce document (hormis une initiation au chapitre 16).

2.6 Conclusion : que retenir de ce chapitre ?

Ce chapitre a détaillé le rôle des systèmes d'exploitation, les conséquences qui en découlent pour l'utilisation des ordinateurs et la liste des services que doivent rendre les systèmes d'exploitation. Il est important de retenir les points suivants :

- un système d'exploitation est un programme ;
- un système d'exploitation est une machine virtuelle plus facile à programmer qu'une machine réelle ; à ce titre, il facilite l'accès aux ressources de la machine et il permet de développer des applications portables, qui ne sont pas spécifiques d'un ordinateur ou d'un système donné ;
- un système d'exploitation est un gestionnaire de ressources qui attribue les ressources aux différents utilisateurs et qui empêche l'accès illicite à celles-ci ; à ce titre, un système d'exploitation est garant du bon fonctionnement de la machine (fiabilité et stabilité) et de la conservation des données (pas de réamorçages intempestifs, pas de perdre de données) ;
- les appels système sont l'interface que le système d'exploitation met à la disposition des utilisateurs pour qu'ils puissent lui demander des services ; sur les systèmes d'exploitation assurant leur rôle de gestionnaire de ressources, les utilisateurs ne peuvent pas accéder directement aux ressources de la machine et doivent nécessairement faire appel au système d'exploitation ;
- un système d'exploitation ne doit pas monopoliser le temps CPU de l'ordinateur qu'il gère et un bon système d'exploitation est capable d'effectuer son travail (notamment la gestion de l'accès aux ressources) sans avoir à disposer du processeur (utilisation des couches matériels comme la MMU) ;

3

Évolution des ordinateurs et des systèmes d'exploitation

Les ordinateurs et les systèmes d'exploitation sont aujourd'hui des objets complexes et il est parfois difficile de comprendre les motivations des personnes qui les ont développés. Ainsi certains archaïsmes subsistent encore et, les raisons ayant motivé leurs développements disparaissant, il n'est pas rare de se demander pourquoi telle ou telle stratégie a été mise en place. Ce chapitre retrace l'évolution des ordinateurs et des systèmes d'exploitation tout au long des quarante dernières années et il devrait permettre de comprendre quels ont été les choix stratégiques importants.

Outre cet aspect culturel de l'évolution des ordinateurs, il est capital de bien comprendre que les systèmes d'exploitation ont tenté de satisfaire les exigences des utilisateurs d'ordinateurs et que, donc, leur développement n'est pas l'œuvre d'une intelligence supérieure qui aurait tout planifié, mais bien le résultat des efforts parfois contradictoires de nombreuses personnes dans le monde entier. À ce titre, ce serait une grossière erreur de croire que les systèmes d'exploitation n'évoluent plus ou qu'il est difficile de mieux faire...

Il est aussi capital de comprendre que les systèmes d'exploitation et l'architecture des ordinateurs ont évolué ensemble et qu'il n'est pas possible de concevoir l'un sans l'autre : les demandes des développeurs de systèmes d'exploitation ont amené les

constructeurs d'ordinateurs à modifier les architectures de ces derniers et les nouveautés technologiques des architectures des ordinateurs ont permis aux développeurs de systèmes d'exploitation de progresser.

Mots clés de ce chapitre : gestionnaire de périphériques (*device driver*), travail, tâche, traitement par lots (*batch*), moniteur résident, interprète de commande (*shell*), *spool*, tampon (*buffer*), ordonnancement, multiprogrammation, partage du temps, exécution concurrente, multi-tâches, multi-tâches préemptif, multi-utilisateurs.

3.1 Les origines et les mythes (16xx–1940)

Les ouvrages sur l'informatique situent souvent l'origine des ordinateurs au XVII^{e} ou au XVIII^{e} siècle en faisant référence à Blaise Pascal ou à Charles Babbage.

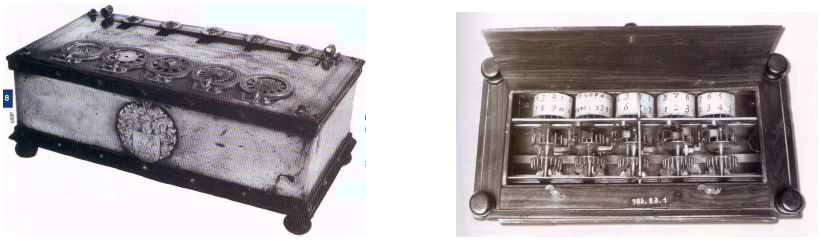


FIGURE 3.1 – *La pascaline et une de ses évolutions, calculatrice inventée par Pascal.*

La pascaline (fig. 3.1) permettait de réaliser des additions et des soustractions. Mais comme l'époque ne connaissait pas encore le système de mesures internationales et qu'il s'agissait avant tout de simplifier les calculs de la vie courantes, différentes versions étaient proposées. Selon le nombre de dents de chaque roue on pouvait ainsi calculer en toises, en sols (avec 20 sols par livre monnaie), en livres (sachant que 12 onces font une livre poids), etc.

La tabulatrice d'Hollerith (voir fig. 3.2) lisait des cartes perforées et en traitait le contenu à l'aide d'un tableau de connexions. Ces connexions étant fixes (dans les premiers temps), les cartes peuvent être considérées comme le jeux de données à manipuler par un programme inscrit de manière permanente au travers de ces connexions.

À partir de 1920 le tableau de connexions sera amovible ce qui permettra de… changer de programme. Un parallèle amusant serait de devoir changer le CPU selon le programme que vous souhaitez exécuter !

3.2 La préhistoire (1940–1955)

Même si les apports antérieurs aux années 1940 sont importants, toutes les machines créées jusqu'alors étaient des calculettes ou des calculateurs dotés de composants

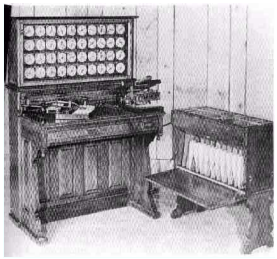


FIGURE 3.2 – The tabulating machine*, utilisée pour le recensement des USA à la fin du* XIX^{e} *siècle. Son inventeur (Hermann Hollerith) fonda pour l'occasion l'entreprise « Tabulating Machine Corporation », devenue en 1911 « Computing Tabulating Recording Corporation », puis en 1924 « International Business Machines Corp » (plus connue sous le nom d'IBM).*

FIGURE 3.3 – *Alan Turing (1912-1954) pose les bases fondamentales de l'informatique théorique et s'interroge sur la « calculabilité » des algorithmes. En particulier, il cherche à déterminer si tout ce qui est calculable humainement peut être calculé par ordinateur et réciproquement. L'application directe de ses travaux lui permettront de casser le code de chiffrement utilisé par les allemands pendant la Seconde Guerre mondiale (la machine Enigma).*

mécaniques ou électromécaniques. Nous considérerons ici que le premier ordinateur est la première machine entièrement électronique et réellement programmable.

Persécuté en 1952 en raison de son homosexualité, Alan Turing décédera dans l'ignorance de la communauté scientifique anglaise qui l'avait pourtant largement applaudit. En 1966 la création du prix Turing récompensant des travaux originaux dans le domaine informatique commencera à rétablir sa place au sein de la communauté scientifique. Cette réhabilitation se poursuivra en 2009 avec les excuses du Premier Ministre britannique Gordon Brown au nom de la justice anglaise pour les traitements infligés à Alan Turing et « ayant entraîné son décès prématuré ».

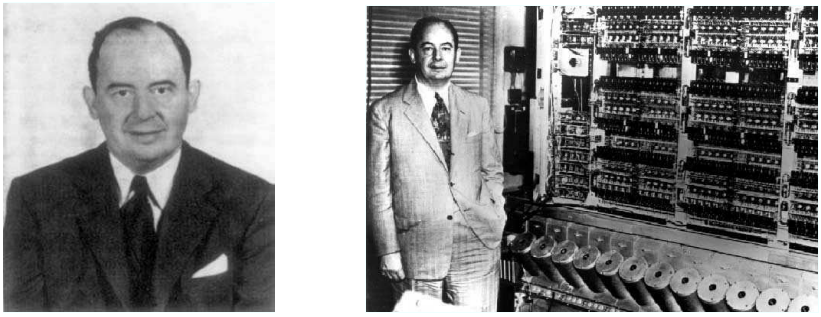


FIGURE 3.4 – *John von Neumann (1903-1957), le père de l'informatique moderne. Les concepts édictés par von Neumann dans les années 40 continuent aujourd'hui de régir la conception des ordinateurs.*

Le premier ordinateur fut construit en Pennsylvanie en 1946 et permettait d'effectuer des calculs de tirs d'artillerie. Il utilisait 18000 tubes à vide et était absolument gigantesque : 30 m de long, 2,80 m de haut et 3 m de large ! Doté d'une mémoire de 20 mots de 10 chiffres, il était capable d'effectuer 5000 additions ou 350 multiplications par seconde.

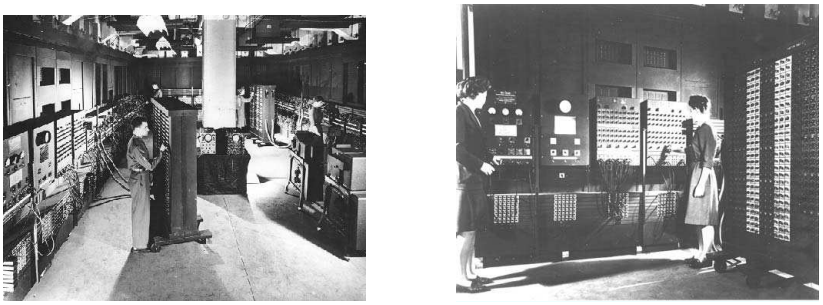


FIGURE 3.5 – *ENIAC (Electronic Numerical Integrator And Computer), le premier ordinateur entièrement électronique (sans partie mécanique), opérationnel en février 1946. Un doute subsiste encore aujourd'hui sur les réelles capacités d'ENIAC et certains pensent qu'il ne s'agissait « que » d'un gros calculateur.*

La durée de vie des tubes à vide était très faible et les ingénieurs avaient à l'époque effectué le calcul de la probabilité de panne d'un tel ordinateur. Le résultat du calcul étant absolument effrayant [1], ils décidèrent de grouper les tubes par cinq sur des râteliers

1. Les différents ouvrages traitant de ce sujet ne sont pas d'accord sur le résultat de ce calcul. Toutefois,

et de changer tout le râtelier dès qu'un tube tombait en panne.

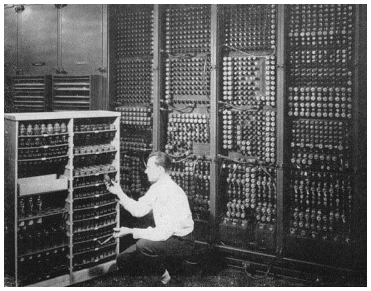
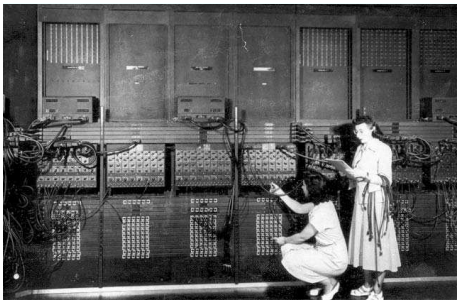


FIGURE 3.6 – *ENIAC consommait 150 kW, pesait 30 tonnes, contenait 18 000 tubes à vide, 70 000 résistances, 10 000 condensateurs et 6 000 commutateurs...*

La programmation s'effectuait directement en manipulant des interrupteurs et des contacteurs. Il n'y avait donc pas de support pour mémoriser les programmes et la programmation devait donc être refaite à chaque changement de programme. L'avantage (le seul !) de ce procédé est que, en cas d'erreur, l'ordinateur restait dans l'état qui avait conduit à l'erreur et il était donc possible de faire du débogage à vif !



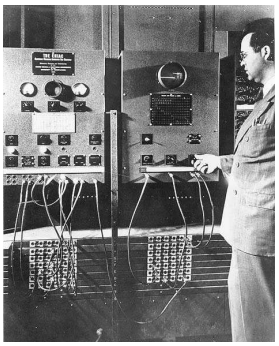


FIGURE 3.7 – *ENIAC était programmable, même si la programmation prenait à l'époque une autre tournure.*

Ces ordinateurs étaient en fait entièrement gérés par un petit groupe de personnes et il n'y avait pas de différence entre les concepteurs, les constructeurs, les programmeurs, les utilisateurs et les chargés de la maintenance. La notion de service était donc totale-

tous s'accordent pour dire que la durée moyenne sans panne de cet ordinateur était de 1 à 2 minutes.

ment absente et, comme par ailleurs les programmes devaient être « reprogrammés » à chaque fois, la notion de système d'exploitation aussi.

3.3 Les ordinateurs à transistor (1955–1965)

D'autres ordinateurs ont vu le jour sur le même principe jusqu'en 1959, mais ils ont rapidement intégré des fonctionnalités indispensables comme les mémoires, les langages de programmation de type assembleur ou l'utilisation de cartes perforées pour stocker les programmes.

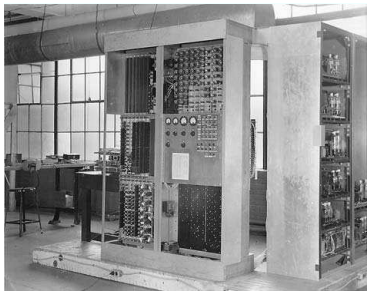


FIGURE 3.8 – *EDVAC (Electronic Discrete VAriable Computer), successeur d'ENIAC, est indéniablement un ordinateur (et non un calculateur).*

FIGURE 3.9 – *UNIVAC (UNIVersal Automatic Computer) construit en 1951. UNIVAC était suffisamment fiable et suffisamment abordable pour être commercialisé.*

Les cartes perforées (voir figure 3.11) permettaient de réutiliser des programmes déjà écrits et, en particulier, de moins se soucier des entrées / sorties : pour chaque périphérique, un programmeur écrivait un petit programme permettant de lire et d'écrire sur ce support et ce petit programme était ensuite inclus dans des programmes plus gros. Ces petits programmes s'appellent des gestionnaires de périphériques (*device*

driver) et représentent la première notion de service, donc la première étape vers un système d'exploitation.

L'apparition des transistors (voir figure 3.10) a permis de construire des ordinateurs plus puissants et surtout plus petits. Par ailleurs, les transitors étaient beaucoup plus fiables que les tubes à vide et il devenait envisageable de vendre des ordinateurs ainsi construits. Cette époque voit donc la séparation entre, d'une part, les concepteurs et les constructeurs et, d'autre part, les programmeurs et les utilisateurs.

FIGURE 3.10 – *Les premiers transistors.*

L'utilisation de cartes perforées s'étant généralisée, la notion de travail ou de tâche à effectuer est apparue. Lorsqu'un programmeur voulait exécuter une tâche, il apportait la carte perforée correspondante (ou les cartes perforées correspondantes) à la personne chargée de l'exécution. Cette personne collectait ainsi les différentes cartes de programme du centre de calcul, les chargeait dans l'ordinateur, déclenchait manuellement leur lecture, puis leur exécution. Lorsque le programme était écrit dans un langage de haut niveau (pour l'époque) comme le FORTRAN ou le COBOL, l'opérateur devait en plus se munir de la carte perforée du compilateur et assurer qu'elle serait chargée avant la carte perforée du programme.

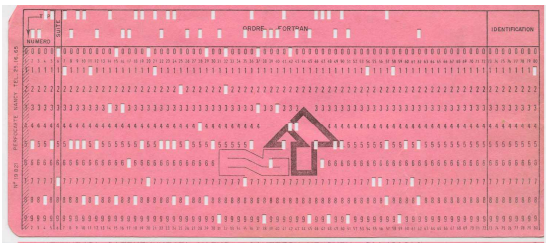


FIGURE 3.11 – *Une carte perforée utilisée à l'*ENSTA *ParisTech.*

À partir de cette époque, les programmeurs et les utilisateurs n'étaient plus les mêmes personnes, même s'il est aujourd'hui difficile de considérer l'opérateur chargé

du fonctionnement de l'ordinateur comme un utilisateur. Par ailleurs, il n'était plus possible de déboguer les programmes en temps réel et il devenait donc nécessaire d'imprimer toutes les informations utiles pour un débogage *a posteriori*.

Ces manipulations faisant intervenir un opérateur humain étaient très lentes et l'ordinateur passait l'essentiel du temps à attendre qu'on le nourrisse. Comme chaque ordinateur coûtait très cher (de l'ordre du million de dollars), il était capital que son utilisation soit optimisée afin de rentabiliser l'investissement. C'est ce qu'on appelle généralement « le retour sur investissement ».

Notons que la programmation de gestionnaires de périphériques réutilisables permet déjà d'améliorer l'utilisation de l'ordinateur : cela évite, d'une part, que chaque programmeur perde du temps à reprogrammer ces fonctionnalités et cela assure, d'autre part, que le gestionnaire utilisé est optimal (et non programmé à la va-vite...).

Le traitement par lots (*batch*)

Afin de limiter les manipulations de cartes perforées, la première étape fut de regrouper les travaux par lots. Plutôt que de charger les cartes dans n'importe quel ordre et d'être obligé de recharger plusieurs fois les cartes perforées des compilateurs, l'opérateur regroupait tous les travaux du centre de calculs et les séparait en différents lots correspondant aux différents langages utilisés : un pour le FORTRAN, un pour le COBOL, etc. La carte perforée du compilateur FORTRAN (par exemple) n'était alors chargée qu'une seule fois dans la journée.

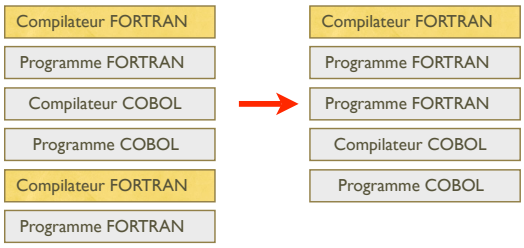


FIGURE 3.12 – *Principe du traitement par lots : regrouper les travaux ayant besoin des mêmes cartes perforées.*

Le tri des cartes perforées était néanmoins toujours assuré par l'opérateur et ce dernier continuait de déclencher manuellement la lecture de ces cartes et l'exécution des travaux.

Le moniteur résident

Afin de réduire encore l'intervention de l'opérateur, la lecture des cartes perforées et l'enchaînement des travaux ont été automatisés grâce à un petit programme appelé « moniteur résident ». Son nom provient du fait qu'il résidait en permanence dans la mémoire (voir figure 3.13), ce qui à l'époque était une première : jusque là, la mémoire était entièrement utilisée par le travail en cours et était vidée à la fin de celui-ci.

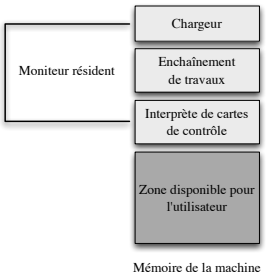


FIGURE 3.13 – *Occupation de la mémoire par le moniteur résident.*

Il fallait néanmoins indiquer au moniteur résident le travail à effectuer et des cartes perforées particulières – des cartes de contrôle – étaient utilisées à cet effet : elles indiquaient le début et la fin d'un travail, le nom du compilateur à appeler, le chargement des données et le début de l'exécution proprement dite. Ces cartes étaient intercalées entre les cartes perforées du programme ou des données (voir figure 3.14) et représentent le premier interprète de commande.

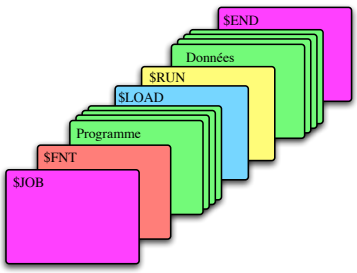


FIGURE 3.14 – *Utilisation de cartes perforées de contrôle pour spécifier le travail à accomplir.*

Le travail de l'opérateur se réduisait alors à trier les cartes perforées pour effectuer le traitement par lots et à ajouter les cartes de contrôle. Le moniteur résident assurait

trois tâches distinctes (le chargement des cartes, l'enchaînement des travaux et l'interprétation des commandes) et représente le premier système d'exploitation de l'histoire de l'informatique.

Le traitement hors-ligne (*off-line*)

Même en utilisant un moniteur résident, l'ordinateur perdait beaucoup de temps en lisant les cartes perforées et en imprimant les résultats. Peu à peu les bandes magnétiques, beaucoup plus rapides, remplacèrent les cartes perforées et les imprimantes, ce qui améliora nettement les temps d'entrées / sorties. Toutefois, il fallait d'abord recopier les cartes perforées des programmes sur une bande magnétique, puis, une fois tous les travaux effectués, relire la bande pour imprimer les résultats.

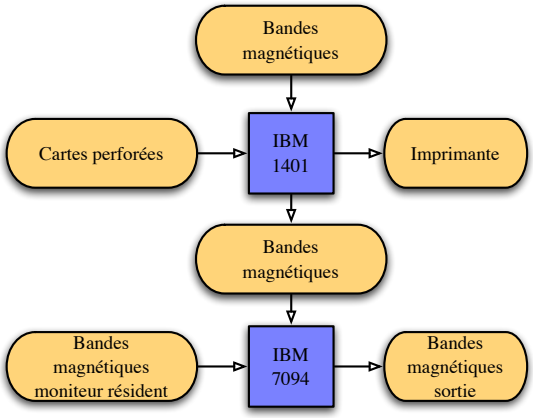


FIGURE 3.15 – *Le principe du traitement hors-ligne.*

Comme il n'était pas envisageable d'utiliser des ordinateurs de calcul pour effectuer ces traitements hors-ligne, de petits ordinateurs, très mauvais pour les calculs, étaient utilisés pour ces transferts des cartes vers les bandes et des bandes vers l'imprimante. Cette pratique imposa petit à petit l'idée de périphériques indépendants et autonomes qui prennent en charge l'essentiel du travail d'entrées / sorties.

Pourquoi ne pas écrire les programmes directement sur les bandes magnétiques ? Les bandes magnétiques sont des systèmes à accès séquentiel, c'est-à-dire que pour lire une donnée située au milieu de la bande, il faut d'abord la rembobiner, puis la lire de bout en bout jusqu'à ce qu'on trouve la donnée en question. L'utilisation de cartes perforées permet donc à plusieurs programmeurs d'écrire en même temps leurs programmes (chacun sur sa carte et chacun sur son ordinateur), ce qui serait totalement impossible avec des bandes magnétiques.

Le traitement différé des entrées / sorties (*spool*)

Le caractère séquentiel des bandes magnétiques était très pénalisant et entraîna le développement d'un périphérique à accès aléatoire, c'est-à-dire pour lequel l'accès à n'importe quelle donnée est direct. Ce périphérique s'appelle le disque dur et est devenu depuis un des périphériques les plus courants.

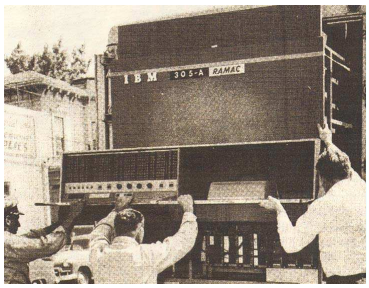


FIGURE 3.16 – *Le premier disque dur.*

Les disques durs se sont rapidement répandus et ont tout naturellement relégué au rang de stockage secondaire les bandes magnétiques (elles sont toujours utilisées pour réaliser des sauvegardes). Le traitement hors-ligne disparut en même temps au profit du *spool* : *Simultaneous Peripheral Operation On Line*. Les cartes perforées sont alors directement copiées sur le disque dur et l'ordinateur y accède quand il en a besoin. De même, lorsque l'ordinateur effectue un travail, il écrit le résultat sur le disque dur et il ne l'envoie à l'imprimante que lorsqu'il a terminé ce travail.

Aujourd'hui, le terme de *spool* perdure pour les travaux envoyés à l'imprimante mais a disparu pour la plupart des entrées / sorties : on parle désormais de *buffering*, c'est-à-dire de l'utilisation de mémoires tampon (*buffer*) pour accumuler les entrées / sorties avant de les envoyer aux périphériques concernés. La différence entre le *spool* et l'utilisation de *buffer* tient dans la petite taille des *buffer*. Par exemple, lorsqu'un programme demande l'impression à l'écran d'une ligne de texte, cette ligne est stockée dans une mémoire tampon et n'est imprimée que lorsque ce tampon est plein ou lorsque le programmeur en fait la demande explicite [2].

3.4 Les circuits intégrés (1965–1980)

Le traitement hors-ligne des entrées / sorties entraîna le développement de deux gammes d'ordinateurs différentes : d'une part, des gros ordinateurs très coûteux spécialisés dans les calculs et, d'autre part, des petits ordinateurs beaucoup moins chers

2. Parfois sans le savoir...

spécialisés dans les entrées / sorties. La première gamme était essentiellement utilisée par l'armée, les universités ou la météorologie nationale. La seconde gamme était utilisée par les banques, les assureurs et les grandes administrations.

Ces deux gammes étaient malheureusement incompatibles et une banque désireuse d'acheter du matériel plus perfectionné ne pouvait pas changer de gamme sans au préalable re-développer tous les programmes qu'elle utilisait. Les constructeurs d'ordinateur décidèrent alors de produire des ordinateurs compatibles, ce qui fut notamment possible grâce à l'utilisation de circuits intégrés.

La coexistence de ces deux gammes représente bien les deux approches des systèmes d'exploitation : les gros ordinateurs devaient être utilisés au mieux pour faire des calculs et tous les efforts tendaient vers l'efficacité des traitements ; réciproquement, les petit ordinateurs devaient être pratiques d'utilisation et tous les efforts tendaient vers la commodité. Le dilemme efficacité versus commodité persiste encore de nos jours et, paradoxalement, les systèmes d'exploitation se sont essentiellement développés sur les petits ordinateurs.

Ordonnancement dynamique

L'ordonnancement des travaux est l'ordre dans lequel ces travaux vont être exécutés par l'ordinateur. Cet ordonnancement était jusqu'alors déterminé par l'opérateur lorsque celui-ci triait les cartes perforées. Puis, les cartes étant recopiées sur des bandes magnétiques, cet ordonnancement était fixé une fois pour toutes et l'ordinateur était contraint d'exécuter les travaux dans l'ordre dans lequel ils arrivaient.

Avec l'utilisation des disques durs, l'ordinateur est capable de choisir lui-même l'ordre dans lequel les tâches vont être exécutées. Si, par exemple, les tâches A, B et C doivent être exécutées et si la tache B a besoin des résultats de C, l'ordinateur pourra décider d'exécuter C avant B. L'ordre dans lequel les programmes sont écrits sur le disque dur et l'ordre dans lequel ils sont exécutés n'ont donc plus rien à voir. Une des tâches du système d'exploitation est donc de déterminer cet ordre.

Multiprogrammation

À partir du moment où le système d'exploitation peut choisir l'ordre dans lequel les tâches s'exécutent et dans la mesure où deux tâches peuvent coexister en mémoire (le système d'exploitation plus la tâche courante), il devenait possible de charger plusieurs tâches en mémoire et de passer de l'une à l'autre. L'idée, comme toujours, était de ne pas laisser l'ordinateur inoccupé. Or les temps d'entrées / sorties étaient assez longs (quelques milli-secondes) par rapport aux temps de réaction des ordinateurs et une façon de rentabiliser la machine consistait à exécuter une autre tâche pendant que la tâche en cours attendait des données stockées (ou à stocker) sur des périphériques.

Par exemple, si une tâche doit afficher à l'écran des lignes de texte, l'ordinateur a le temps entre deux affichages successifs d'un caractère d'effectuer quelques calculs. Ce principe s'appelle la multiprogrammation et il ne faut pas le confondre avec le

temps partagé que nous décrirons juste après : dès que la tâche a terminé ses entrées / sorties, elle reprend la main pour effectuer les calculs sans être interrompue jusqu'aux prochaines entrées / sorties.

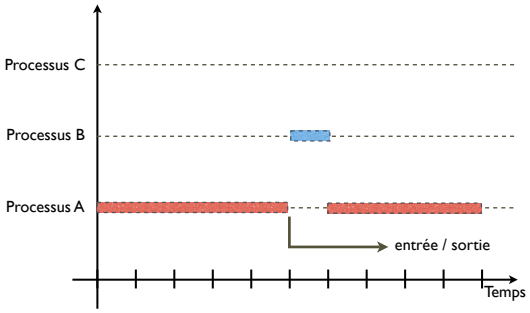


FIGURE 3.17 – *Principe de la multi-programmation : un processus attendant une entrée / sortie n'utilise plus le processeur et celui-ci est affecté à un autre processus.*

La commutation d'une tâche à l'autre suppose que le système d'exploitation est capable de sauvegarder toutes les informations et toutes les données nécessaires à l'exécution des deux tâches. Cela suppose aussi que le système est capable de protéger les tâches entre elles, c'est-à-dire qu'il est capable d'empêcher chaque tâche d'accéder aux zones mémoires utilisées par les autres, y compris celles qu'il utilise lui-même !

Les systèmes d'exploitation à multiprogrammation ont surtout été utilisés pour les petits ordinateurs destinés aux banques, aux assureurs et aux administrations. En effet, les gros ordinateurs étaient essentiellement utilisés pour des calculs scientifiques et les temps d'entrées / sorties étaient très faibles par rapport aux temps de calculs. Pour des entreprises gérant des comptes bancaires ou des dossiers de clients, c'était bien entendu l'inverse !

Le partage du temps

La multiprogrammation permettait de ne pas perdre trop de temps en attendant les entrées / sorties, mais le fonctionnement des ordinateurs était encore et toujours fondé sur le traitement par lots. Avec la disparition des bandes magnétiques et des cartes perforées, il devenait difficile de justifier ce procédé et de nombreux défauts le rendaient très pénible au quotidien. En particulier, un programmeur qui voulait tester son programme devait attendre que celui-ci soit sélectionné par le système d'exploitation, ce qui pouvait prendre des heures si les travaux en cours duraient très longtemps. En cas d'erreur, non seulement le programmeur ne pouvait pas déboguer le programme directement, mais en plus il devait remettre son programme (après

correction) dans la file d'attente. Tester et déboguer un programme prenait donc des jours et des jours, car la moindre erreur supposait une attente de quelques heures.

Il paraît difficile aujourd'hui de se représenter le temps ainsi perdu, mais imaginons que lors de l'écriture d'un gros programme, nous perdions ne serait-ce qu'une demi-heure pour tout point-virgule oublié ou pour toute accolade mal placée...

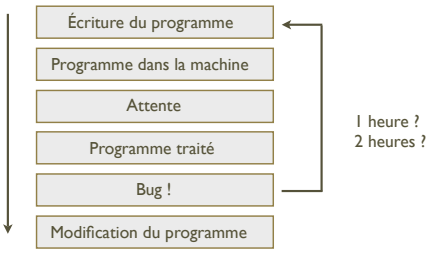


FIGURE 3.18 – *La mise au point fastidieuse des programmes sur des ordinateurs à traitement par lots...*

Ces contraintes ont donc fait place à la notion d'interactivité. L'idée consistait à élargir le principe de la multiprogrammation afin de donner l'illusion aux utilisateurs de disposer de toute la machine pour exécuter leur tâche. Pour cela, chaque tâche dispose d'un petit laps de temps pour s'exécuter, puis est contrainte de passer la main à la tâche suivante, et ainsi de suite jusqu'à ce que la main revienne à la première tâche. Ainsi, si les laps de temps sont assez petits, l'utilisateur a la sensation que, pour une durée donnée, plusieurs tâches s'exécutent de façon concurrente.

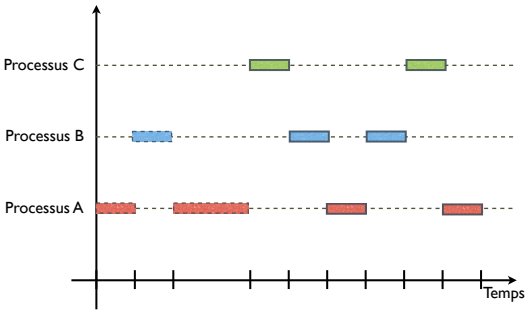


FIGURE 3.19 – *Principe de l'exécution en temps partagé.*

Il ne faut pas confondre l'exécution concurrente de tâches avec l'exécution parallèle : lors d'une exécution concurrente, à tout instant, une seule et unique tâche

s'exécute sur l'unique processeur de l'ordinateur et ce n'est que lorsqu'on observe la situation à une échelle de temps plus grande (de l'ordre de la seconde, par exemple) qu'on a l'illusion que plusieurs tâches s'exécutent en même temps. En revanche, si on utilise un ordinateur doté de plusieurs processeurs, il est possible d'exécuter plusieurs tâches en parallèle, c'est-à-dire qu'à un instant donné chaque processeur n'exécutera qu'une seule tâche, mais l'ensemble des processeurs de l'ordinateur exécuteront bel et bien plusieurs tâches en même temps. Notons que rien n'interdit d'exécuter des tâches de façon concurrente et parallèle sur un ordinateur multi-processeurs.

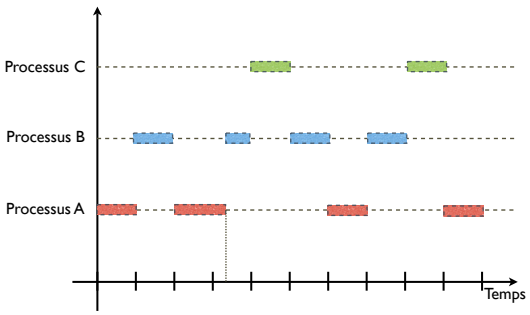


FIGURE 3.20 – *L'exécution en temps partagé inclut la multi-programmation et un processus peut très bien perdre la main avant la fin du quantum de temps qui lui avait été affecté : il suffit pour cela qu'il ait besoin d'une entrée / sortie.*

Par abus de langage, les systèmes d'exploitation permettant d'exécuter les tâches en temps partagé sont appelés multi-tâches, mais ce terme est assez ambigu et il est préférable de ne pas l'employer. Nous verrons d'ailleurs dans la section sur les systèmes d'exploitation actuels que certains vendeurs de systèmes jouent sur cette expression pour tromper le client.

Avec un système à temps partagé, la commutation d'une tâche à l'autre doit être très rapide car elle est effectuée plusieurs fois par seconde. Il devient donc capital d'accélérer le procédé de sauvegarde qui permet de restaurer l'état dans lequel était une tâche avant d'être interrompue pour céder la place à une autre tâche. En pratique, cette opération, appelée changement de contexte, est essentiellement prise en charge par les couches matérielles de la machine.

La protection des tâches entre elles devenait aussi de plus en plus importante et les systèmes à temps partagé ont rapidement imposé l'utilisation de la mémoire virtuelle et des modes noyau ou utilisateur des processeurs. Ceci a d'ailleurs freiné leur propagation car beaucoup d'ordinateurs de l'époque étaient dotés de processeurs qui ne disposaient pas de ce double mode d'exécution et il était bien sûr impensable de racheter des ordinateurs uniquement pour offrir un peu d'interactivité aux program-

meurs (n'oublions pas que le temps ordinateur a longtemps coûté plus cher que le temps programmeur).

FIGURE 3.21 – *Le principe d'interactivité permet d'utiliser des périphériques pour contrôler le fonctionnement de l'ordinateur. Ici la première souris.*

Unix

Le MIT, les laboratoires Bell et la compagnie General Electrics décidèrent de construire une machine sur laquelle pourraient se connecter des centaines d'utilisateurs à la fois. Le système d'exploitation de cette machine s'appelait MULTICS (*MULTiplexed Information and Computing Service*). Il comportait des innovations intéressantes et intégrait l'essentiel des services assurés jusqu'alors par l'ensemble des systèmes d'exploitation. Malheureusement, le projet MULTICS était trop ambitieux et fut abandonné.

Entre temps, les « mini-ordinateurs » avaient vu le jour : le premier de ces ordinateurs était le DEC PDP-1, né en 1961, qui pour 120 000 dollars offrait une capacité de 4096 mots de 8 bits. Cet ordinateur remporta un franc succès et DEC produisit d'autres modèles, tous incompatibles entre eux, ce qui permit de faire les meilleurs choix technologiques pour chaque génération. Parmi ces ordinateurs, le PDP-7 et le PDP-11 sont les plus connus.

Ken Thompson travaillait chez Bell et avait participé au projet MULTICS. Il décida d'en écrire une version simplifiée, en particulier mono-utilisateur, pour son PDP-7. Cette version fut baptisée ironiquement UNICS (*UNiplexed...*) par Brian Kernighan, puis son nom se transforma en Unix. Le résultat étant très satisfaisant, les laboratoires Bell achetèrent un PDP-11 et Thompson y implanta son système. Dennis Ritchie [3], qui conçut et réalisa le langage `C`, travaillait aussi chez Bell et il réécrivit avec Thomson le système Unix en `C`. Comme le `C` est un langage de haut niveau facile à porter de machine en machine, Unix fut rapidement adapté à de nombreuses architectures différentes et devint le système d'exploitation le plus porté de tous les temps.

3. Dennis Ritchie est décédé début octobre 2011. Si sa mort ne fit pas la couverture des journaux comme celle de Steve Jobs, son influence, aux dires des historiens, fut largement comparable.

Unix est une marque déposée des laboratoires Bell, mais ceux-ci distribuèrent (quasi-)librement les sources d'Unix pendant un certain temps. Puis, après avoir pris conscience de la valeur marchande d'un tel système, ils décidèrent de le commercialiser. Le nom d'Unix était déjà connu, mais n'était plus utilisable (sauf par Bell). C'est pourquoi de nombreux systèmes d'exploitation de type Unix ont vu le jour sous des noms très différents, mais généralement de consonance proche : Minix, Ultrix, Irix, Linux...

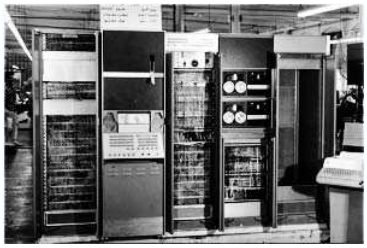


FIGURE 3.22 – *Les « mini-ordinateurs » Dec PDP-1 et PDP-7.*

3.5 L'informatique moderne (1980–1995)

L'informatique moderne se traduit essentiellement par la généralisation de l'utilisation des ordinateurs. Paradoxalement, cette généralisation s'est faite au mépris des systèmes d'exploitation, car comme il devenait possible de donner un ordinateur à chaque utilisateur, le rôle du système d'exploitation a vite été sous-estimé. Ainsi, de nombreuses sociétés ont préféré acheter de grandes quantités de petites machines dotées de systèmes d'exploitation rudimentaires plutôt que de croire en l'utilisation intelligente de l'outil informatique.

Ces choix, datant des années 1980, continuent et continueront de pénaliser les sociétés ainsi dotées et de favoriser leur inertie face aux innovations. Comme ces sociétés ont fondé leur stratégie sur des principes, sur des langages de programmation, sur des outils ou sur des systèmes d'exploitation qui, depuis, sont complètement obsolètes, elles sont confrontées à deux choix pour le futur : ou bien elles continuent malgré tout à utiliser ces outils du passé sachant que ça leur revient très cher (aujourd'hui le temps programmeur est beaucoup beaucoup plus cher que le temps ordinateur), ou bien elles décident de se mettre à jour et cela suppose que tous les développements thésaurisés pendant ces 20 dernières années seront mis à la poubelle. Cela revient à repartir à zéro !

Les ordinateurs personnels

Les ordinateurs personnels datent du début des années 1980. On parle parfois de micro-ordinateurs, mais cette considération est aujourd'hui dépassée : la taille d'un ordinateur n'est pas proportionnelle à sa puissance. Il est d'ailleurs difficile de distinguer de nos jours un ordinateur personnel d'un ordinateur professionnel.

Les ordinateurs personnels sont naturellement groupés en deux catégories : les PC et les Macintosh. L'origine de cette distinction n'est pas déterminée et, une fois de plus, il devient aujourd'hui très difficile de distinguer un PC d'un Mac. D'ailleurs, bon nombre de personnes pensent qu'un PC est un ordinateur qui exécute les programmes Windows ou Windows XP, ce qui n'a rien à voir !

MS-DOS et Unix

Les deux systèmes d'exploitation qui se sont propagés pendant cette période sont MS-DOS et Unix. Une fois de plus, ces deux systèmes représentent les deux extrêmes des systèmes d'exploitation : la commodité contre l'efficacité.

MS-DOS est un système rudimentaire très facile à utiliser. Unix est un système complexe et redoutablement efficace, mais de prime abord parfois rude. Peu à peu, MS-DOS s'est rapproché d'Unix et Unix a fait des efforts pour devenir plus accessible. Nous aborderons l'avenir de ces deux systèmes dans la section sur les systèmes actuels.

Les réseaux d'ordinateurs

La multiplication des ordinateurs a permis de les connecter et de créer des réseaux d'ordinateurs. La gestion des communications entre ordinateurs est alors une des nouvelles tâches des systèmes d'exploitation.

Les machines (massivement) parallèles

Des ordinateurs contenant plusieurs dizaines, voire plusieurs milliers, de processeurs on vu le jour. Sur le principe, ils se rapprochaient en fait des vieux ordinateurs de calcul, car ils étaient très coûteux et il fallait à tout prix les rentabiliser. Ainsi, le traitement par lots a eu un sursaut de vie lorsque ces machines étaient à la mode.

Aujourd'hui cette mode est révolue et les machines de demain devraient être dotées de quelques processeurs seulement (2, 4, 8 ou 16). En revanche, il est exclu d'utiliser ces machines sans un système d'exploitation multi-utilisateurs à temps partagé.

3.6 Les systèmes d'exploitation d'aujourd'hui

Dans cette section, nous présentons rapidement les systèmes d'exploitation que l'on trouve couramment aujourd'hui sur les ordinateurs. Les caractéristiques présentées

FIGURE 3.23 – *La* Thinking Machine *contenait 65536 processeurs répartis dans un* hypercube*, à savoir les processeurs étaient à la « même distance » les uns des autres, par le biais d'un réseau de routage inter-processeurs. Elle pouvait toutefois être utilisée en mode asynchrone pour simuler des cas réels tels que les grands réseaux électriques.*

ici sont susceptibles de ne pas être totalement d'actualité car les systèmes évoluent assez rapidement de version en version.

Avant de décrire les systèmes, nous faisons un rapide point sur le vocabulaire à la mode.

Vocabulaire et définition

Il n'est pas possible de désosser tous les systèmes d'exploitation que l'on rencontre, ne serait-ce que parce que la plupart d'entre eux (tous les systèmes commerciaux) ne sont pas fournis avec leurs sources. Pour parler des capacités des systèmes, il faut donc se mettre d'accord sur un vocabulaire commun et faire confiance aux opinions des développeurs ou des utilisateurs des systèmes.

Comme justement beaucoup de constructeurs ou de vendeurs de logiciels jouent sur l'ambiguïté des mots pour imposer leurs productions, nous allons ici essayer de définir clairement les notions que nous avons énoncées dans les sections ci-dessus.

Le moniteur résident

Le moniteur résident représente le strict minimum du travail d'un système d'exploitation. Les seuls services assurés sont l'interprétation des commandes, l'enchaînement des tâches et l'accès simplifié aux périphériques.

Il paraît impensable aujourd'hui d'utiliser encore des moniteurs résidents pour contrôler des ordinateurs employés professionnellement.

Les systèmes multi-tâches

Comme nous le disions plus haut, le terme multi-tâches est mal choisi et il faut en fait lui préférer l'expression « temps partagé ». Malheureusement, le terme multi-tâches s'est imposé et il est difficile de revenir en arrière.

Un système à temps partagé est un système qui peut exécuter plusieurs tâches pendant une durée donnée. Chaque tâche dispose du processeur pendant un bref laps de temps, puis est interrompue pour laisser le processeur à une autre tâche (et ainsi de suite).

Les systèmes multi-tâches préemptifs

La notion de multi-tâches préemptif ne devrait pas exister et nous pouvons considérer qu'un système qui ne serait pas multi-tâche préemptif ne mérite pas le nom de multi-tâches. Pourquoi alors cette distinction ? En fait, certaines sociétés commerciales qui n'étaient pas capables de produire un système d'exploitation multi-tâches ont développé des systèmes qui donnaient l'illusion du multi-tâches [4]. Ces systèmes, plutôt que d'être appelés « faux multi-tâches », ont conservé le nom de multi-tâches et il a donc fallu trouver un autre nom pour les vrais systèmes multi-tâches.

Un système multi-tâches non préemptif ne peut pas interrompre l'exécution d'une tâche en cours et il doit attendre que celle-ci rende spontanément la main pour attribuer le processeur à une autre tâche. Donc si une tâche décide de garder le processeur pour elle pendant des heures, rien ne l'en empêche ! Mais il est toujours possible de choisir deux ou trois programmes rendant spontanément et régulièrement la main pour faire de jolies démonstrations...

Inversement, un système multi-tâches préemptif décide quand il interrompt une tâche pour attribuer le processeur à une autre tâche. Choisir la prochaine tâche à exécuter n'est pas simple et les différentes politiques d'ordonnancement sont très intéressantes à étudier. C'est ce qu'on appelle l'ordonnancement dynamique des processus (*scheduling* en anglais).

Il est bien entendu beaucoup plus facile de demander gentiment à une tâche de rendre la main plutôt que de l'interrompre d'autorité !

Les systèmes multi-utilisateurs

Il ne faut pas confondre les systèmes multi-tâches et les systèmes multi-utilisateurs, même si souvent les deux vont de pair. Un système multi-utilisateurs est un système sur lequel plusieurs utilisateurs peuvent travailler, mais pas nécessairement de façon interactive. Par exemple, il existe des systèmes multi-utilisateurs à traitement par lots.

Un système multi-utilisateurs doit protéger les données de chaque utilisateur sur les périphériques à support non volatile, comme par exemple, les fichiers sur disque dur.

4. Sachant que, par définition, le multi-tâches donne déjà l'illusion du parallélisme, ces systèmes donnent donc l'illusion de l'illusion du parallélisme...

Cependant, plusieurs tâches du même utilisateur peuvent accéder aux mêmes données sur disque dur. Cette protection est donc très différente de celle de la mémoire décrite ci-dessous.

La mémoire virtuelle

Les systèmes à mémoire virtuelle permettent, d'une part, d'utiliser plus de mémoire que l'ordinateur n'a de mémoire principale et, d'autre part, de ne pas utiliser explicitement des adresses physiques pour accéder aux données.

La protection de la mémoire

Les systèmes à temps partagé doivent protéger les tâches les unes des autres et, en particulier, ils doivent se préserver eux-mêmes des intrusions des autres tâches. Lorsqu'un système protège la mémoire utilisée par les différentes tâches, il lui est toujours possible de garder la main, même lorsqu'une tâche commet une erreur et s'arrête de fonctionner normalement. Ainsi, il est facile de reconnaître les systèmes n'utilisant pas de protection de mémoire : si un programme plante, il est généralement nécessaire de réinitialiser la machine (*reboot* en anglais).

La protection de la mémoire va généralement de pair avec les systèmes multi-tâches et l'utilisation de la mémoire virtuelle.

Les systèmes de type Unix

Il est paradoxalement très difficile de définir ce qu'est un système d'exploitation de type Unix. Pendant longtemps, les systèmes Unix étaient les seuls à être multi-tâches (préemptif, bien-sûr !), multi-utilisateurs, avec mémoire virtuelle et protection de la mémoire.

Avec l'apparition récente de systèmes offrant les mêmes caractéristiques, la question de ce qu'est un système Unix se pose. Est-il défini par l'interface qu'il offre aux utilisateurs et aux programmeurs ? Est-il défini par la façon dont il est programmé ? Est-il défini par la philosophie de gestion des périphériques ?

Cette question reste ici sans réponse et nous nous contenterons de citer quelques systèmes d'exploitation de type Unix :

4.4BSD : système développé par l'université de Berkeley et dont une grande partie est gratuite ;

Minix : système gratuit développé à partir de la version 7 de l'Unix des laboratoires Bell ;

Linux : ce système est gratuit et peut être installé sur de nombreuses machines, comme les PC, les MacIntosh, les DEC alpha ou les stations de travail SUN ;

NetBSD : tout comme Linux, ce système – fondé sur 4.4BSD – est gratuit et il peut être installé sur un grand nombre de machines ;

OpenBSD : issu de NetBSD, il en conserve les caractéristiques mais se focalise sur les architectures de types PC et SPARC. Ses objectifs sont avant tout la sécurité et la cryptographie des données ;

FreeBSD : fondé sur 4.4BSD, ce système est aussi gratuit et le plus répandu de la famille BSD ;

Irix : système utilisé sur les stations de travail SGI ;

Solaris : système utilisé sur les stations de travail SUN ou plutôt Oracle ;

HP-UX : système utilisé sur les stations de travail HP ;

AIX : système développé par IBM ;

Ultrix : système utilisé sur les stations Digital VAX et Decstation a base de processeur MIPS ;

True64 Unix : autrefois appelé OSF/1 puis Digital Unix, ce système est utilisé sur les DEC Alpha.

Les systèmes d'exploitation de Microsoft

La société Microsoft a développé plusieurs systèmes d'exploitation qui ont eu beaucoup de succès : MS-DOS, Windows, Windows 95, Windows 98, Windows NT, Windows XP, Windows Server 2000, Windows Vista, Windows Seven...

MS-DOS

Le principal système d'exploitation fonctionnant sur des PC est MS-DOS (*Disc Operating System*). Initialement, en 1980, le système d'exploitation des PC était le CP/M 80, développé par Digital Research. Ce système était conçu pour des processeurs 8 bits et ne convenait donc pas au « tout nouveau » processeur 8086. Digital Research annonça alors le développement du CP/M 86 adapté au processeur 8086, mais la sortie de ce système tarda, et un programmeur, Jim Paterson, développa en 1981 un nouveau système d'exploitation, de 6 Ko seulement, qu'il nomma 86-DOS. Peu après Microsoft produisit MS-DOS. Précisons que Microsoft n'a pas développé la première version de MS-DOS mais l'a achetée.

Les premières versions de MS-DOS restaient compatibles avec le CP/M 80, ce qui ne facilitait pas leurs développements. En 1983 et 1984 naquirent respectivement les versions 2.0 et 3.0 de MS-DOS qui s'éloignaient au fur et à mesure du CP/M 80 pour tenter de se rapprocher des facilités d'Unix. Voici les grandes étapes de MS-DOS :

MS-DOS 1.0 en 1981,

MS-DOS 2.0 en 1983 qui adopte une structure hiérarchique pour le système de fichiers ;

MS-DOS 3.0 en 1984 qui autorise des lecteurs de 1,2 Megaoctets (jusqu'alors les disquettes avaient une capacité de 360 Ko !) et les disques durs ;

MS-DOS 3.2 en 1986 qui permet l'utilisation de disquette 3 pouces 1/2 (le format qui était plus répandu à l'époque) et qui est aussi la première version de DOS incompatible avec les précédentes ;

MS-DOS 6.22 la dernière version autonome de MS-DOS ;

MS-DOS 7 sortie en 1995, le DOS de Windows 95,

MS-DOS 7.1 enfin une version capable de supporter la FAT32 (!),

MS-DOS 8 il s'agit de la dernière version si l'on exclut MS-DOS 2000 qui rajoute simplement quelques fonctionnalités.

MS-DOS n'est ni plus ni moins qu'un moniteur résident. Il est donc mono-tâche, mono-utilisateur, sans mémoire virtuelle ni protection de la mémoire. Comme tous les moniteurs résidents, il assure uniquement l'interprétation des commandes et l'enchaînement de celles-ci. MS-DOS est construit sur une sous-couche appelée BIOS (*Basic Input Output System*) qui assure l'essentiel des fonctions d'entrées / sorties.

Windows

Initialement, Windows n'était qu'une interface graphique de MS-DOS. Puis, peu à peu, Windows s'est développé indépendamment et la version Windows 3.11 utilisait par exemple la mémoire virtuelle. Néanmoins, Windows 3.11 est un système mono-tâche, mono-utilisateur et sans protection de mémoire.

Les raisons expliquant le succès de Windows sont probablement les mêmes que celles expliquant le succès de MS-DOS : simplicité et possibilité d'accéder directement aux couches matérielles les plus basses de la machine. Il y a cependant une autre raison expliquant le succès de Windows : les traitements de texte, les tableurs et les logiciels de dessins développés par Microsoft avaient de réelles qualités et ne pouvaient fonctionner sur PC qu'avec le système d'exploitation Windows. En liant les deux, Microsoft assurait la propagation de son système d'exploitation. Cette stratégie commerciale est très classique, mais on peut lui reprocher de s'opposer aux progrès informatiques.

Windows et Unix se sont longtemps opposés en proposant des services totalement différents : d'un coté, Windows était simple d'accès mais peu efficace, d'un autre coté, Unix était très efficace mais peu convivial. Cette guerre a eu au moins le mérite de montrer qu'aucune de ces deux attitudes n'était la bonne et il est probable que sans l'intervention de Windows, les systèmes Unix d'aujourd'hui seraient peut-être moins avancés en termes de convivialité. Notons toutefois qu'en matière d'interfaces graphiques, NeXT, société fondée en 1985 par Steve Jobs, offrait un système d'exploitation basé sur Unix avec une interface graphique très soignée. C'est d'ailleurs le rachat, en 1996, par Apple de NeXT qui conduira à la mise en place de Mac OS X tel que nous le connaissons aujourd'hui.

Windows 95

Windows 95 a été annoncé comme une révolution par Microsoft. Il devait être multi-tâches, avec mémoire virtuelle et protection de la mémoire. En pratique, Microsoft a eu beaucoup de mal à tenir ses promesses et le développement du système prenant de plus en plus de retard, il fut finalement vendu alors qu'il n'était pas tout à fait prêt.

Par ailleurs et pour des raisons de compatibilité, les programmes prévus pour Windows 3.11 peuvent aussi être exécutés sous Windows 95. Or ces programmes ont l'habitude d'utiliser la machine comme bon leur semble et il est très difficile les faire fonctionner en multi-tâches (voir section 5.1).

Windows 95 n'est donc pas multi-tâches, quoi que peut en dire Microsoft, et il apparaît à l'usage que la protection de la mémoire ne fonctionne pas toujours.

Windows 98

Windows 98 est la nouvelle mouture du système d'exploitation Windows 95. Il reprend les mêmes caractéristiques que Windows 95 et, donc, les mêmes défauts. L'apparition de Windows 98 a été justifiée par un remaniement de certaines fonctionnalités et par la correction de certains bugs, mais il s'avère que c'était surtout une bonne occasion pour Microsoft de tenter d'imposer ses logiciels de navigation sur l'Internet (Internet Explorer) et de gestion des courriers électroniques (Outlook).

Notons aussi que cela permet, d'une part, d'occuper le marché et, d'autre part, de se faire un peu d'argent en vendant des mises à jour pour Windows 95.

Contrairement aux systèmes d'exploitation évolués, MS-DOS, Windows 3.11, Windows 95 et Windows 98 laissaient l'utilisateur accéder directement aux divers périphériques, avec ou sans l'intermédiaire du BIOS. C'était probablement cette faculté alliée à la simplicité extrême de ces environnements qui expliquait leur succès. Ils sont aujourd'hui totalement dépassés et peuvent à peine servir pour des machines de jeux.

Windows NT

Windows NT est le premier système multi-tâches avec mémoire virtuelle et protection de la mémoire produit par Microsoft. En comparaison des précédentes versions de Windows, Windows NT était donc plus stable, mais aussi moins accessible. Microsoft visait essentiellement avec Windows NT le marché des systèmes d'exploitation pour serveurs et était présenté comme le concurrent direct d'Unix.

Windows NT n'est pas multi-utilisateurs [5] et apparaît comme un système peu mature [6]. Ses utilisateurs lui reprochent en particulier sa lourdeur (il s'agit d'un système d'exploitation mixte comme cela est présenté dans le paragraphe 2.4) et sa lente évolution. Pourtant, malgré ses défauts, Windows NT a atteint son but et Microsoft a ainsi pu pénétrer dans le marché très fermé des serveurs.

5. Il existe cependant des surcouches permettant de gérer plusieurs utilisateurs.
6. Malgré des numéros de version élevés, c'est un système très récent (1993).

Windows NT fonctionne sur des PC, mais aussi sur des stations de travail SGI [7] ou DEC Alpha.

Windows Server 2000

Windows Server 2000 est la première version du système d'exploitation de Microsoft prenant en compte la notion d'utilisateurs et le fait que plusieurs utilisateurs peuvent utiliser le même ordinateur. L'objectif était de créer un système tenant à la fois de Windows 98 et de Windows NT, capable de remplacer les deux. C'est aussi la version qui indique les difficultés de Microsoft à continuer dans la voie du « système d'exploitation bon à tout faire » : il existe en fait plusieurs versions de Windows 2000 et, en particulier, une version dédiée aux serveurs.

En pratique, Windows Server 2000 a des qualités indéniables et sa stabilité est incomparable avec celle des versions précédentes. Il n'y a cependant pas de miracle : pour construire un système stable, Microsoft a dû employer les mêmes méthodes que celles des autres systèmes robustes (notamment les mêmes méthodes qu'Unix) et a introduit de nombreuses limitations dans la légendaire facilité d'utilisation de Windows. En particulier, la compatibilité ascendante n'a pas pu être maintenue (les logiciels fonctionnant sous Windows 95 ou Windows 98 ne fonctionnent pas nécessairement sous Windows Server 2000) et le paramétrage d'un serveur sous Windows Server 2000 est nettement plus complexe que le paramétrage d'une station sous Windows 95 !

Ceci a eu deux effets de bord inattendus :

– Beaucoup d'entreprises ont attendu très longtemps avant de passer sous Windows Server 2000 car cette migration entraînait le redéveloppement de la plupart des logiciels, car elle impliquait un changement important dans les méthodes de gestion et d'administration des postes de travail et car elle entraînait une bascule « violente » d'un système à l'autre, avec des risques d'interruption prolongée de service [8].
– Beaucoup d'entreprises qui ont fait les efforts et les investissements nécessaires pour passer à Windows Server 2000 ont décidé d'amortir sur la durée cette bascule et ont ensuite refusé de passer aux versions suivantes de Windows. Windows Server 2000 est ainsi aujourd'hui le système Windows le plus répandu dans les entreprises (plus que ses prédécesseurs et plus que ses successeurs).

Même s'ils sont assez proches, Windows Server 2000 et Unix reposent l'éternel problème des systèmes d'exploitation : commodité versus efficacité. La querelle qui oppose Windows Server 2000 et Unix est la même que celle qui opposait MS-DOS et Unix, puis Windows et Unix. Il est probable que, comme précédemment, les systèmes

7. Après une brève tentative visant à commercialiser des stations graphiques sous Windows NT, SGI a finalement décidé de revenir à Unix et, en particulier, de développer des stations graphiques fonctionnant sous Linux...

8. Ces risques ne sont pas propres à Windows et se retrouvent à chaque fois que l'on change de système d'information dans une entreprise, dès lors qu'il n'est pas possible de faire une bascule douce de l'ancien au nouveau (poste par poste, par exemple).

d'exploitation de Microsoft s'inspirent largement d'Unix pour améliorer les services qu'ils proposent et qu'Unix s'inspire de la facilité d'accès de ces systèmes pour améliorer sa convivialité. En fait, tant que ces deux systèmes subsistent, l'utilisateur est gagnant car non seulement il peut choisir l'un ou l'autre suivant ses besoins, mais en plus, il se voit proposer des systèmes de plus en plus performants.

Windows XP et les suivants

Windows XP est une évolution de Windows Server 2000 qui porte essentiellement sur des corrections d'ergonomie et sur un allégement des protections mises en place sous Windows Server 2000 : ce retour arrière est vraisemblablement une tentative pour contrer les deux effets de bord décrits ci-dessus.

Windows Server 2003 est une version consacrée aux serveurs, dérivée de Windows Server 2000, corrigeant certains dysfonctionnements.

Windows Vista est une version intermédiaire entre Windows Server 2000 et Windows Seven. Les principales nouveautés sont liées à la gestion des systèmes 64 bits, une interface graphique utilisant de manière massive les données vectorielles (s'appuyant sur *Windows Presentation Foundation*) qui aurait pour effet indirect de faire baisser l'autonomie des ordinateurs portables.

Des efforts ont aussi été consentis dans le domaine de la sécurité. . . , notamment le fait d'avoir des programmes lancés dans un contexte d'exécution plus faible que le niveau administrateur et surtout l'incorporation de la couche TCPA Palladium. Cette couche de DRM [9] inclut deux choses. Tout d'abord TCPA (*Trusted Computing Plateform Alliance*) qui permet d'assurer que seuls des logiciels signés (. . . par Microsoft ?) peuvent s'exécuter. Ensuite l'architecture Palladium qui instaure un chiffrement des données circulant sur les bus de données. Cela permet par exemple d'éviter toute interception d'un flux vidéo provenant d'un lecteur de DVD et arrivant sur la carte graphique. Ce chiffrement devrait être assuré par des composants matériels intégrés à la carte mère et aux cartes graphiques. Le prix de cet ajout permettant de garantir l'impossibilité des copies serait supporté par. . . les utilisateurs !

Les autres systèmes

OS/2

OS/2 est un système multi-tâches, multi-utilisateurs, avec mémoire virtuelle et protection de la mémoire. Il est essentiellement soutenu par IBM et aurait dû avoir du succès. Conçu à la base en 1987 par Microsoft et IBM, il a été conçu comme un descendant commun de MS-DOS et d'Unix.

Il est difficile de dire pourquoi OS/2 n'a pas eu beaucoup de succès. Initialement, la raison invoquée était sa lourdeur, mais il s'est avéré que Windows 95 et Windows NT souffraient du même défaut. Il est probable qu'OS/2 ait en fait souffert d'une guerre

9. *Digital Rights Management*.

qui oppose l'alliance Apple/IBM à l'alliance Intel/Microsoft et qui visait à développer un processeur (le PowerPC) en dehors du contrôle d'Intel.

Mach

Le système d'exploitation Mach est en fait un micro-noyau (voir section 2.4) sur lequel doivent être greffés d'autres services. Il fut conçu à l'Université de Carnagie Mellon en 1985 dans le but de réaliser des travaux de recherche sur les systèmes d'exploitation. Mach a un rapport très net avec Unix car il propose une interface entièrement compatible avec BSD. Cependant, les concepteurs de Mach annoncent que ce n'est pas un système Unix. La question de la définition d'un système Unix se repose donc par ce biais.

Mach est multi-utilisateurs, multi-tâches, avec mémoire virtuelle et protection de la mémoire. Il a été porté sur de nombreuses architectures et il devrait à l'avenir prendre de l'importance car il peut être la sous-couche de n'importe quel système. Comme il assure l'essentiel des services d'un système d'exploitation moderne, la couche supérieure peut être entièrement dédiée à l'interface pour l'utilisateur. L'utilisation d'un micro-noyau comme Mach permet donc d'entrevoir une solution au dilemme efficacité versus commodité : deux couches totalement disjointes s'occupant l'une de l'efficacité et l'autre de la commodité.

Mach est utilisé par les système d'exploitation NeXT et MacOS. Il devait aussi être utilisé pour 4.4BSD, mais il s'est avéré que la structure monolithique de 4.4BSD était beaucoup plus efficace et moins lourde.

Hurd

Hurd est le système d'exploitation du projet GNU de la FSF. Hurd est annoncé comme un système non Unix, mais il a cependant un rapport très net avec Unix : Hurd est fondé sur un micro-noyau Mach. De nombreux espoirs sont fondés sur Hurd et il se pourrait que ce soit le premier système qui soit à la fois commode et efficace. Il est malheureusement trop tôt pour l'instant pour statuer sur son sort.

Mac OS X

Le système Mac OS X (pour *Macintosh Operating System* version 10) équipant les derniers ordinateurs Apple est fondé sur un système d'exploitation Unix de type BSD. Il est donc multi-tâches, multi-utilisateurs, avec mémoire virtuelle et protection de la mémoire (ce qui n'était pas le cas des versions précédentes de Mac OS)

Très réputé pour son savoir-faire en ergonomie des ordinateurs, Apple a tenu ses promesses avec l'environnement de travail proposé sous Mac OS X : celui-ci est

gourmand en ressources (il faut des ordinateurs puissants et bien dotés en mémoire), mais se révèle un excellent [10] compromis entre commodité et efficacité.

Apple a annoncé qu'il abandonnait les processeurs IBM (gamme PowerPC) au profit des processeurs Intel, ce qui lui a permis de proposer des ordinateurs moins chers, plus proches des prix d'entrée de gamme des PC. Si les ventes restent toutefois modestes, la combinaison d'une interface graphique très agréable et d'un noyau très sécurisé comme Unix pour surprendre tout le monde et on pourrait voir Apple gagner la bataille « Windows versus Unix ».

Ce n'est cependant pas la première fois qu'Apple se retrouve en position favorable sur le marché de l'informatique, avec des machines innovantes, bien conçues et de bonne qualité mais, malheureusement, cela ne lui a pas suffit pour s'imposer. Prédire l'avenir de l'informatique est donc un exercice très difficile et l'expérience montre que les événements décisifs se situent souvent en dehors du champs de bataille : il ne serait pas étonnant, par exemple, que la bataille du poste de travail pour l'entreprise se joue sur des considérations ayant trait aux jeux vidéo ou aux capacités multimédia... Il faut en effet savoir que la plupart des processeurs suffisent plus qu'amplement pour des travaux bureautiques, mais c'est principalement l'industrie du jeu électronique qui tire vers plus de puissance et qui dope l'économie numérique.

3.7 Conclusion : que faut-il retenir de ce chapitre ?

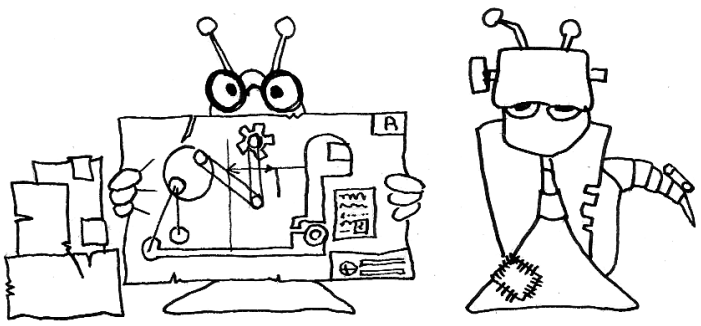
Ce chapitre nous a permis de parcourir l'histoire des ordinateurs et des systèmes d'exploitation afin de mieux expliquer comment l'informatique en est arrivé au stade actuel. Les points suivants sont importants :

- le principal moteur du progrès informatique a longtemps été la nécessité (et la volonté) d'augmenter le retour sur investissement ; cette motivation est aujourd'hui moins présente dans les entreprises, probablement parce que les coûts informatiques sont désormais répartis sur de nombreux postes, mais elle n'en reste pas moins capitale ;
- l'informatique actuelle est fondée sur l'utilisation de composants et de périphériques apparus très tardivement (disque dur, écran, souris, etc.) ; il est donc important de ne pas croire que l'informatique n'évoluera plus et que l'état actuel représente l'optimum que l'on ne pourra pas dépasser ;
- les concepts fondamentaux de l'informatique actuels (multi-tâches, multi-utilisateurs, protection de mémoire et mémoire virtuelle) ont été mis en œuvre de façon opérationnelle depuis 30 ans et certains systèmes comme le système Unix proposent les 4 services à la fois ; il est donc important d'évaluer à sa juste valeur la prouesse technique d'un développeur de systèmes qui n'obtiendrait ce résultat qu'à partir de l'an 2000...

10. Ceci est bien sûr l'avis personnel de l'auteur qui travaille quotidiennement sous Mac OS, Windows Server 2000 et Linux, aussi bien dans son environnement professionnel que pour ses loisirs.

4

Compilation et édition de liens



Depuis très longtemps déjà, plus personne ne programme directement en langage machine : ce type de programmation requiert beaucoup de temps, notamment pour le débogage, et les programmes ainsi écrits dépendent très fortement de l'ordinateur pour lequel ils ont été écrits.

Pour s'abstraire de la machine, des langages de haut niveau ont été développés et ils permettent d'écrire des programmes qui, d'une part, peuvent être réutilisés de machine en machine et, d'autre part, ont une structure plus proche de la façon dont un être humain conçoit les choses.

Ces programmes écrits dans des langages de haut niveau doivent ensuite être traduits en langage machine : c'est le but de la compilation. Il existe de nombreux langages qui peuvent être compilés et il est difficile de prévoir quels seront les langages du futur. Parmi les langages qui ont eu du succès, citons le FORTRAN (FORmula TRANslator, 1954), le COBOL (COmmon Business Oriented Language, 1959) et le LISP (LISt Processing language [1], 1958). Les langages actuellement les plus utilisés sont le C (1972), le C++ et JAVA. Enfin, nous citerons le PASCAL (1970) qui est un langage intemporel : c'est probablement le meilleur langage pour apprendre à programmer, mais il est difficilement utilisable dans des conditions opérationnelles.

1. Parfois appelé List of Inutil and Stupid Parentheses par ses détracteurs

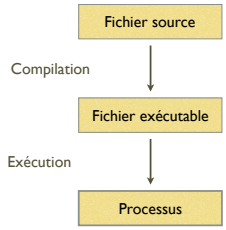


FIGURE 4.1 – *La compilation permet de passer d'un fichier source à un fichier exécutable, l'exécution permet de passer d'un fichier exécutable à une éxécutable en cours d'éxécution (c'est-à-dire un processus)*

Il ne faut pas confondre les langages compilés avec les langages interprétés, comme le BASIC, le SHELL ou PERL : ces langages sont directement interprétés, généralement ligne à ligne, par un autre programme (qui porte parfois le même nom). Les programmes écrits avec de tels langages sont donc moins rapides, mais ils peuvent être débugés plus facilement : l'interprète est souvent couplé avec un éditeur de texte et il signale généralement les zones du programme qu'il ne comprend pas au moment où celles-ci sont écrites. Cette facilité a donné naissance à des langages qui peuvent être à la fois interprétés et compilés, comme le PASCAL ou le LISP.

Si les programmes compilés restent les plus rapide à exécuter, certains langages interprétés intègrent maintenant des compilateurs qui vont transformer un ensemble d'instructions liées au langage en un jeu de codes opératoires. Si l'on prend l'exemple de l'interprète `Tcl`, le court extrait suivant :

```
while (my_condition} {
  do something
  change my_condition
}
```

sera transformé de la façon suivante lors de la première interprétation :

```
goto y
x: do something
   change my_condition
y: evaluate my_condition
branche-if-true x
```

ce qui permet une seconde lecture beaucoup plus efficace, du point de la machine.

C'est aussi sur ce principe qu'est fondé le langage `JAVA`. Il s'agit d'un langage objet (proche du `C++`). Le code est tout d'abord traduite en *bytecode*. C'est ce jeu d'instructions qui est envoyé à la machine virtuelle `JAVA` qui en assurera l'exécution sur le système d'exploitation cible. C'est naturellement la qualité de la machine virtuelle

qui dépend totalement du système d'exploitation sur lequel elle a été conçue et de sa capacité à intégrer les diverses versions du langage que repose la bonne exécution du *bytecode*. Il est naturellement possible de demander à cette machine virtuelle de fournir un fichier binaire qui pourra être exécuté (sous réserve de pouvoir trouver toutes les librairies dont il dépendra obligatoirement). Toutefois on perd totalement dans ce cas la philosophie de JAVA qui est la portabilité [2]. Les machines virtuelles JAVA actuelles les plus sophistiquées fonctionnent selon un mécanisme différent du *bytecode*, jugé trop lent par rapport aux langages compilés tels que C et C++. Le *bytecode* est compilé en code natif par la machine virtuelle au moment de son exécution (JIT pour *Just In Time compiler*). On trouve aussi maintenant des recompilations dynamique du *bytecode* permettant de tirer partie d'une phase d'analyse du programme et d'une compilation sélective de certains morceaux du programme [3].

Suivant les ouvrages traitant du sujet, les compilateurs font ou ne font pas partie du système d'exploitation [4] : rien n'empêche un utilisateur d'écrire son propre compilateur mais il est difficile de faire quoi que ce soit sur une machine sans au moins un compilateur (il peut y en avoir plusieurs). Quoi qu'il en soit, le système d'exploitation intervient doublement dans la phase de compilation. Tout d'abord, le travail du compilateur dépend très fortement des choix faits pour le système d'exploitation et, par exemple, la construction d'un exécutable destiné à un système utilisant la mémoire virtuelle n'est pas la même que la construction d'un exécutable destiné à un système ne travaillant qu'avec des adresses physiques.

Ensuite, le système d'exploitation fournit une interface de programmation aux programmeurs : les appels système. La façon dont ces appels système sont effectivement programmés et exécutés dépend de chaque système d'exploitation et un même programme compilé avec le même compilateur sur une même machine peut avoir des performances très différentes suivant le système d'exploitation utilisé.

Nous allons donc dans cette section expliquer rapidement les principes de compilation en détaillant les notions importantes que nous réutiliserons dans le chapitre sur la gestion de la mémoire.

Mots clés de ce chapitre : compilation, préprocesseur, assemblage, édition de liens, bibliothèque statique, bibliothèque dynamique, bibliothèque partagée, exécutable, segment de code, segment de programme, segment de données, table des symboles, références externes, références internes, adresse relative.

2. Notons à ce propos que cette portabilité n'est pas toujours simple à obtenir et que le slogan de Sun concernant JAVA « Write once, run anywhere » à été détourné pour donner « Write once, debug anywhere » !

3. L'interprétation du *bytecode*, même si elle est toujours beaucoup moins lourde que l'interprétation du langage source, introduit *ipso facto* un surcoût. Pouvoir transformer le *bytecode* en code natif conduit à une réelle amélioration des performances en termes de temps.

4. JAVAOs en est un bon exemple puisqu'il s'agit d'un système d'exploitation ne reconnaissant qu'un seul langage de programmation... JAVA et faisant donc office de compilateur !

4.1 Vocabulaire et définitions

Les termes employés en informatique manquent généralement de précision et, en particulier, ceux utilisés pour tout ce qui concerne la programmation et la compilation. Cette brève section vise simplement à mettre les choses au clair.

Les fichiers

Un fichier est une zone physique de taille limitée contenant des données. Cette zone peut se trouver sur un des périphériques de la machine, comme le disque dur, le lecteur de CD-ROM ou la mémoire, ou sur un des périphériques d'un ordinateur distant, connecté à notre machine via un réseau local.

La notion de fichier sera discutée dans le chapitre sur les systèmes de fichiers, mais, pour l'instant, considérons simplement que c'est une abstraction d'un ensemble de données. Du point de vue de l'ordinateur, un fichier est un ensemble de 0 et de 1 stockés de façon contigüe.

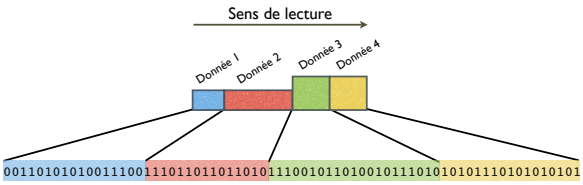


FIGURE 4.2 – *Un fichier n'est qu'un ensemble de 0 et de 1, stockés selon un ordre donné, sur un support indéfini. Cette suite de 0 et de 1 peut ensuite être interprétée comme des « objets informatiques » de taille et de type différents.*

Les fichiers texte et les fichiers binaires

Les fichiers sont arbitrairement rangés en deux catégories : les fichiers texte et les fichiers binaires. Un fichier texte est un fichier contenant des caractères de l'alphabet étendu (comme #, e ou 1) codés sur un octet ou plusieurs octets. Chaque donnée est en fait un entier (variant entre 0 et 2^{32} pour l'UTF-8 par exemple) et une table de correspondances permet de faire le lien entre chaque entier et le caractère qu'il représente.

Pendant de nombreuses années, la table de correspondances utilisée était la table ASCII (*American Standard Code for Information Interchange*). Cette table n'est néanmoins pas adaptée aux langues accentuées (comme le français et la plupart des langues

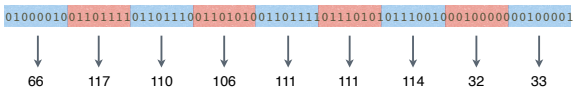


FIGURE 4.3 – *Dans un fichier texte utilisant une représzentation simple telle que le codage ASCII, la suite de 0 et de 1 s'interprête par paquets de 8, chaque paquet représentant en binaire un nombre entre 0 et 255.*

européens) et d'autres tables ont vu le jour après avoir été normalisées. Ainsi, la table ISO 8859-1 contient l'essentiel des caractères utilisés par les langues européennes [5].

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	é	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	|	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	ø	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

FIGURE 4.4 – *Table ASCII.*

La table ASCII, bien que toujours très utilisée, se voit peu à peu supplanter par le système de codage UTF-8. Il serait en effet impossible, en utilisant une table possédant 256 entrées, de gérer des langues telles que le chinois ou l'arabe. Il devint assez incontournable que pour être utilisée de manière quasi universelle, une table de correspondance se devait de pouvoir intégrer la totalité des symboles présents dans les différents langages. Cela suppose donc un codage de chaque caractère sur un nombre d'octets plus grand que 1. Il était pourtant impératif de conserver une compatibilité

5. La table ISO 8859-1 ne contient malheureusement pas les caractères œ et Œ : la légende raconte que ces caractères ont été évincés au dernier moment par les représentants des pays européens en profitant de l'absence du représentant français lors de la dernière réunion qui devait enterrinner la norme...

avec la table ASCII. L'UTF-8 apporte une solution dans le ce qu'il permet, en fonction du bit de poids fort de chaque octet servant au codage d'un caractère, de savoir s'il est nécessaire de lire l'octet suivant pour obtenir la totalité du code du symbole. On conserve ainsi les 128 premiers caractères de la table ASCCI auquel on peut rajouter d'autres symboles, le codage pouvant être étendu à trois octets.

Insistons sur le fait que les fichiers texte ne contiennent que des chiffres codés sur un octet. Cela signifie qu'il suffit de changer la table de correspondances pour qu'un même fichier texte change complètement d'apparence. Par exemple, il suffirait de lire le texte composant ce polycopié avec la table de correspondances utilisée en Chine pour avoir l'impression de lire du chinois...

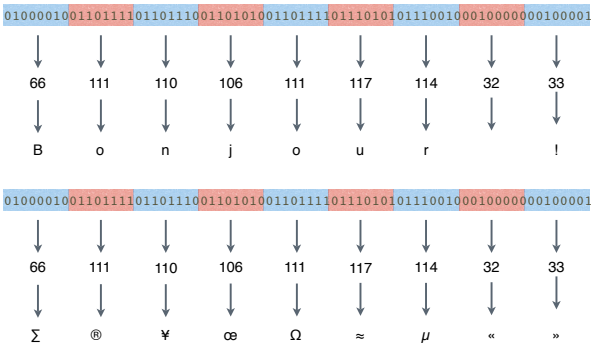


FIGURE 4.5 – *Les nombres compris en 0 et 255 d'un fichier texte peuvent être traduits sous forme de caractères grâce à des tables de correspondance. Un même fichier peut donner des résultats très différents en fonction de la table de correspondance utilisée. En haut, le résultat obtenu en utilisant la table ASCII. En bas, le résultat obtenu en utilisant une autre table.*

La seule chose qui distingue un fichier texte d'un fichier qui n'est pas un fichier texte, c'est que nous connaissons la façon dont les informations sont codées : il suffit de lire les 0 et les 1 par paquets de 8 et de traduire le nombre ainsi obtenu en caractère grâce à la table utilisée (ce qui suppose que le nom de cette table est stocké quelque part dans le système de fichiers).

Les fichiers binaires contiennent des données qui sont stockées dans un format variable et non déterminé a priori : pour lire un fichier binaire, il faut connaître le format des données qu'il contient. Par ailleurs, rien n'assure que toutes les données contenues dans ce fichier sont codées de la même manière. Par exemple, nous pouvons écrire dans un fichier 10 entiers, qui seront chacun codés sur 4 octets, puis 20 caractères, qui seront chacun codés sur 1 octet. Nous obtiendrons ainsi un fichier de 60 octets et il est impossible de deviner comment nous devons le décoder : nous pourrions,

par exemple, considérer que c'est un fichier texte et le lire octet par octet ce qui donnerait probablement un texte sans aucun sens... Ainsi, un fichier ne contient pas des informations qui ont un sens absolu, mais simplement des informations qui n'ont que le sens que nous voulons bien leur donner.

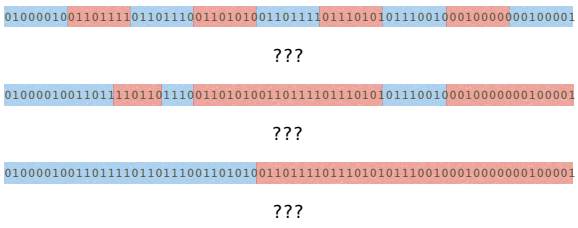


FIGURE 4.6 – *Impossible d'interprèter un fichier binaire sans savoir à l'avance ce qu'il contient.*

Les fichiers texte sont très utilisés car, non seulement les informations sont codées dans un format prédéterminé, mais en plus ils permettent de coder l'essentiel des informations dont nous avons besoin. Ainsi, il est fréquent de coder des entiers non pas directement, mais par une suite de caractères : par exemple, la chaîne de caractères « 1048575 » codée sur 7 octets (un octet par caractère) représente l'entier 1048575 codé sur quatre octets (00000000 00000111 11111111 11111111 en binaire).

Les fichiers exécutables

Les fichiers exécutables sont des fichiers binaires particuliers qui sont compréhensibles directement par la machine. Plus exactement, ils sont directement compréhensibles par le système d'exploitation qui saura comment les traduire en actions concrètes de la machine.

La composition de ces fichiers est complexe et nous en verrons les grands principes un peu plus loin. Pour l'instant, il faut bien comprendre que ce sont des fichiers qui peuvent être exécutés. Mais ce ne sont que des fichiers, c'est-à-dire des objets passifs contenant des informations et des données.

Les processus

La notion de processus sera clairement expliquée dans le chapitre suivant sur la gestion des processus. Disons pour le moment qu'un processus représente l'exécution d'un fichier exécutable, c'est-à-dire un programme qui est en train d'être exécuté.

Un processus est donc un objet actif, contrairement aux fichiers qui sont des objets passifs. Ainsi, au moment où un ordinateur est éteint, tous les processus encore présents disparaissent, alors que tous les fichiers sur support non volatile demeurent.

Les programmes

Le terme de programme regroupe toutes les notions énoncées ci-dessus. Il est donc très mal choisi car pas assez restrictif. Parfois, un programme représente les fichiers ASCII de code. Par exemple, on dit : « j'écris un programme ». Parfois, le terme de programme fait référence au fichier exécutable : « j'exécute un programme ». Enfin, le terme de programme est parfois employé pour désigner le processus obtenu par l'exécution du fichier exécutable : « mon programme s'est arrêté ».

Il n'est donc pas très prudent d'utiliser le mot « programme », mais malheureusement son emploi est répandu et il est probable que nous l'emploierons dans ce texte.

La compilation

Un compilateur est un programme qui lit un fichier contenant du code écrit dans un langage donné, appelé programme source, et qui produit un fichier contenant du code dans un autre langage, appelé programme cible. Lors de cette traduction, le compilateur indique les erreurs présentes dans le programme source. Le principe est représenté sur la figure 4.7.

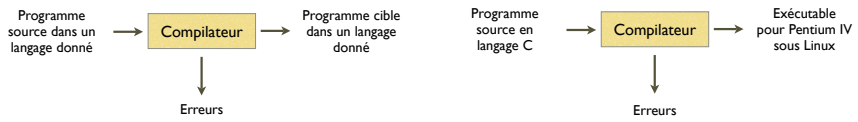


FIGURE 4.7 – *Principe de la compilation et la compilation telle qu'on l'emploie usuellement.*

Dans notre cas, nous nous intéresserons à des programmes source écrits dans un langage de haut niveau, comme le C et à des programmes destination écrits dans un langage de bas niveau, directement compréhensible par un ordinateur (figure 4.7). Le compilateur effectue donc la transformation d'un fichier source en un fichier exécutable.

La façon dont un processus peut être obtenu à partir d'un fichier exécutable ne fait pas partie du procédé de compilation et sera décrite ultérieurement, dans le chapitre sur la gestion des processus.

4.2 Les phases de compilation

Les principes décrits dans cette section sont généraux, mais nous supposerons, pour simplifier les explications, que nous souhaitons compiler des fichiers source écrits en C afin d'obtenir un exécutable. Certaines précisions s'inspirent fortement de la façon

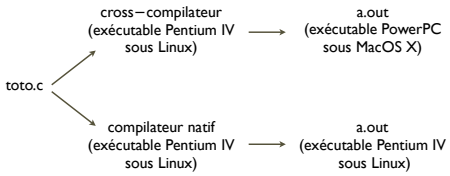


FIGURE 4.8 – *La* cross-compilation *permet d'obtenir des fichiers exécutables à destination d'une architecture donnée à partir d'un fichier source et d'un compilateur fonctionnant sur une autre architecture. C'est par ce biais qu'il est possible de créer de nouvelles architecture sans réinventer la roue : les différents programmes nécessaires au bon fonctionnement de cette nouvelle architecture sont obtenus par cross-compilation à partir d'un architecture existante.*

dont cela fonctionne sous Unix, mais l'ensemble du raisonnement est valable pour tous les compilateurs.

Considérons les fichiers `principal.c` et `mafonc.c` suivant, écrits en langage `C` :

`principal.c`

```
#include <stdio.h>
#include <math.h>

#define NBR_MAX 20
#define VAL 2.0

int main(int argc, char *argv[])
{
  int j;
  float i;

  printf("Ceci est le debut\n");

  i = VAL;
  j = 0;

  while(j < NBR_MAX)
    {
      i = myfonc(2*sqrt(i));
      j++;
    }
  printf("i = %f -- j = %d\n",i,j);

  exit(1);
}
```

`myfonc.c`

```
#define SEUIL 0.5
#define ECHELLE 2.0


float myfonc(float i)
{
  float j;

  if(i < SEUIL) {
    j = i + i/ECHELLE;
  }
  else {
    j = i - i/ECHELLE;
  }

  return j;
}
```

Les fichiers sont tout d'abord traités séparément et leur compilation s'effectue en quatre étapes : l'action du préprocesseur, la compilation, l'assemblage et l'édition de liens.

L'action du préprocesseur

Cette étape, parfois appelée précompilation ou *preprocessing*, traduit le fichier `principal.c` en un fichier `principal.i` ne contenant plus de directives pour le

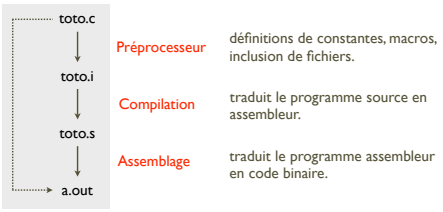


FIGURE 4.9 – *Les phases de compilation.*

préprocesseur. Ces directives sont toutes les lignes commençant par le caractère # et sont essentiellement de deux types : les inclusions de fichiers et les macros. Les commentaires du fichier `principal.c` sont aussi supprimés lors de cette étape. Le préprocesseur remplace aussi les occurences de `__FILE__`, `__LINE__` et autres par ce qui correspond.

Les inclusions de fichiers servent généralement à insérer dans le corps du programme des fichiers d'en-tête (*headers* en anglais) contenant les déclarations des fonctions des bibliothèques utilisées (voir chapitre 26). Ces inclusions sont soit sous la forme `#include <xxx>` pour inclure un fichier se trouvant dans des répertoires prédéfinis (généralement `/usr/include`), soit sous la forme `#include "xxx"` pour inclure des fichiers dont on spécifie le chemin relatif. Traditionnellement, les fichiers contenant des déclarations ont le suffixe « .h » (pour *header*).

Ainsi, le fichier `/usr/include/stdio.h` est inclus dans le fichier `principal.c`, ce qui permet notamment au compilateur de connaître le prototype de la fonction `printf()` et, donc, de vérifier que les arguments passés à cette fonction sont du bon type (pour plus de détail, voir l'aide-mémoire du chapitre 26).

Les macros utilisent la directive `#define` et permettent de remplacer une chaîne de caractères par une autre. Par exemple, dans `principal.c`, le préprocesseur remplacera toutes les occurrences de la chaîne de caractères « VAL » par la chaîne de caractères « 2.0 ».

Les macros sont parfois utilisées pour exécuter des macro-commandes et, dans ce cas, le remplacement effectué par le préprocesseur est plus subtil (voir l'aide-mémoire du chapitre 26).

Il est possible de modifier la valeur d'une macro directement lors de la compilation, c'est-à-dire sans éditer le programme : à cet effet, les compilateurs sont généralement dotés de l'option « -D ». Ainsi, nous pourrions par exemple donner la valeur 3.0 à « VAL » via une commande du type :

```
gcc -c -DVAL=3.0 principal.c
```

La compilation

Une des phases de la compilation s'appelle aussi la compilation, ce qui n'aide pas à clarifier ce concept. En fait, toutes les étapes de compilation décrites ici satisfont à la définition d'une compilation : elles transforment un fichier écrit dans un langage en un fichier écrit dans un autre langage. Seul l'usage permet en fait de leur donner des noms plus significatifs.

La phase de compilation proprement dite transforme le fichier `principal.i` en un fichier `principal.s`, écrit en assembleur. Elle se décompose traditionnellement en quatre étapes : l'analyse lexicale, qui reconnaît les mots clés du langage, l'analyse syntaxique, qui définit la structure du programme, l'analyse sémantique qui permet la résolution des noms, la vérification des types et l'affectation puis enfin la phase d'écriture du code en assembleur.

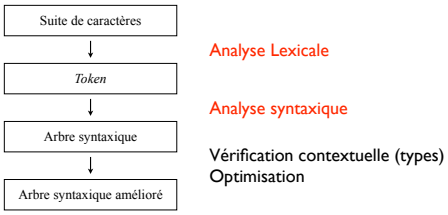


FIGURE 4.10 – *La phase de compilation se décompose elle-même en 4 phases distinctes. Les trois premières étapes, analyse lexicale, analyse syntaxique et analyse sémantique (vérification contextuelle et optimisation) s'enchaînent avant la production du code exécutable.*

Après les étapes lexicale et syntaxique, le programme a une forme indépendante du langage utilisé et indépendante de la machine. Cette représentation des programmes est d'ailleurs commune à beaucoup de compilateurs et elle permet d'effectuer certaines optimisations pour améliorer les performances du programme. Ces optimisations ne tiennent pas compte des caractéristiques de la machine et sont réalisées avant que le programme ne soit traduit en assembleur. D'autres optimisations peuvent avoir lieu pendant la phase d'assemblage : elles dépendent alors fortement de l'architecture de la machine.

```
gcc2_compiled.:
___gnu_compiled_c:
.text
.align 8
LC0:
.ascii "Ceci est le debut\12\0"
.align 8
LC2:
.ascii "i = %f -- j = %d\12\0"
.align 4
LC1:
.word 0x40000000
.align 4
.global _main
.proc 020
_main:
!#PROLOGUE# 0
save %sp,-120,%sp
!#PROLOGUE# 1
st %i0,[%fp+68]
st %i1,[%fp+72]
call ___main,0
nop
sethi %hi(LC0),%o1
or %o1,%lo(LC0),%o0
call _printf,0
nop
sethi %hi(LC1),%o0
ld [%o0+%lo(LC1)],%o1
st %o1,[%fp-16]
st %g0,[%fp-12]
L2:
ld [%fp-12],%o0
cmp %o0,19
ble L4
nop
b L3
nop
L4:
ld [%fp-16],%f2
fstod %f2,%f4
std %f4,[%fp-8]
ldd [%fp-8],%o2
mov %o2,%o0
mov %o3,%o1
call _sqrt,0
nop
std %f0,[%fp-24]
ldd [%fp-24],%f4
ldd [%fp-24],%f6
faddd %f4,%f6,%f4
std %f4,[%fp-8]
ldd [%fp-8],%o2
mov %o2,%o0
mov %o3,%o1
call _myfonc,0
nop
st %o0,[%fp-8]
ld [%fp-8],%f7
fitos %f7,%f2
st %f2,[%fp-16]
ld [%fp-12],%o1
add %o1,1,%o0
mov %o0,%o1
st %o1,[%fp-12]
b L2
nop
L3:
ld [%fp-16],%f2
fstod %f2,%f4
std %f4,[%fp-8]
ldd [%fp-8],%o2
sethi %hi(LC2),%o1
or %o1,%lo(LC2),%o0
mov %o2,%o1
mov %o3,%o2
ld [%fp-12],%o3
call _printf,0
nop
L1:
ret
restore
```

Ci-dessus se trouve le résultat de cette phase de compilation, réalisée avec le compilateur gcc 2.7.2 sur un PC fonctionnant sous Linux 2.0.18. Le sens exact des instructions de ce fichier importe peu, mais il est important de remarquer quelques faits sur lesquels nous reviendrons plus tard : tout d'abord, les chaînes de caractères sont regroupées en tête de ce fichier et, ensuite, l'appel aux fonctions qui ne sont pas définies dans `principal.c` se fait via l'instruction « call ».

L'assemblage

L'assembleur est un langage si proche de la machine qu'il est d'ailleurs spécifique à chaque processeur. Il doit cependant être lui aussi transformé en un fichier écrit dans un langage de plus bas niveau : le langage machine.

Cette transformation s'appelle l'assemblage et elle produit un fichier binaire, `principal.o`, que nous ne pouvons donc pas représenter ici. Ce fichier n'est pas encore le fichier exécutable même s'il en est très proche et s'appelle un fichier objet ou relogeable (*relocatable* en anglais).

L'édition de liens

L'édition de liens est la dernière phase de la compilation. Elle concerne tous les fichiers objet du programme, plus les bibliothèques de fonctions. Les bibliothèques de fonctions sont généralement placées dans le répertoire `/usr/lib` et sont en fait des fichiers objet regroupés sous un format particulier. Elles ont aussi été créées par compilation et contiennent le code d'un certain nombre de fonctions que le système d'exploitation met à la disposition des utilisateurs. Par exemple, la fonction **`printf()`** fait partie de la bibliothèque standard des entrées / sorties.

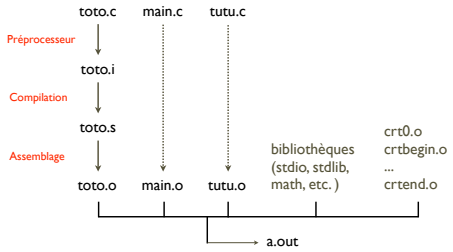


FIGURE 4.11 – *L'édition de liens permet de transformer des fichiers objets en fichier exécutable.*

Pour pouvoir utiliser une fonction de ces bibliothèques, il faut [6] avoir au préalable inclus le ou les fichiers de déclarations correspondant dans le fichier C appelant la fonction en question. Par exemple, pour utiliser la fonction **`printf()`**, il faut inclure le fichier `/usr/include/stdio.h` via une directive du préprocesseur.

Dans notre exemple, les fichiers concernés par l'édition de liens sont `principal.o` et `mafonc.o` et les bibliothèques concernées sont la bibliothèque standard d'entrées / sorties et la bibliothèque mathématique (pour **`sqrt()`**). À ces fichiers s'ajoute un ou plusieurs fichiers très importants qui se nomment souvent `crt0.o`, `crt1.o`, etc. Ces fichiers sont totalement dépendants de la machine et du système d'exploitation : c'est en fait le code contenu dans ces fichiers qui appelle la fameuse fonction **`main()`** des programmes en `C` et ce sont ces fichiers qui contiennent certaines informations nécessaires pour créer un exécutable compréhensible par la machine, comme par exemple les en-têtes des fichiers exécutables [7] (voir plus loin).

L'édition de liens permet donc, à partir des fichiers objet et des bibliothèques de fonctions, de créer (enfin !) l'exécutable désiré. Le fonctionnement de l'éditeur de liens est assez complexe et nous détaillerons quelques-unes de ses fonctions dans la section suivante.

Remarques pratiques sur la compilation

Lorsqu'un compilateur compile des fichiers C, il ne crée pas [8] les différents fichiers intermédiaires (.i et .s). Il faut le lui demander explicitement par diverses options. De même, lorsque le programme ne se compose que d'un seul fichier C, le compilateur ne crée pas de fichier objet, mais directement l'exécutable.

6. En pratique, ces inclusions permettent au compilateur d'effectuer des vérifications. Elles ne sont donc pas obligatoires, mais très fortement recommandées. En cas d'absence, le compilateur affiche généralement de nombreux messages d'avertissement pour signaler les vérifications qu'il ne peut pas faire.
7. À ne pas confondre avec les en-têtes des fichiers sources.
8. Des fichiers temporaires situés généralement dans /tmp sont utilisés, puis effacés.

Voici comment obtenir les différents fichiers intermédiaires avec gcc :

```
gcc -E principal.c -o principal.i
gcc -S principal.c -o principal.s
gcc -c principal.c -o principal.o
gcc principal.o -o principal
```

Par ailleurs, le compilateur n'exécute pas lui-même toutes les phases de la compilation : il fait appel à d'autres outils, comme par exemple le préprocesseur `cpp`, le compilateur `cc1`, l'assembleur `as` ou l'éditeur de liens `ld`. Comme pour les fichiers intermédiaires, une option du compilateur permet de détailler exactement les actions qu'il entreprend.

Voici les différentes phases de compilation, détaillées par le compilateur et effectuées sur un PC sous Linux 2.0.18 avec gcc 2.7.2 :

```
linux> gcc -v -c principal.c
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2/specs
gcc version 2.7.2
 /usr/lib/gcc-lib/i486-linux/2.7.2/cpp -lang-c -v -undef -D__GNUC__=2
-D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux -D__ELF__
-D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386 -D__linux
-Asystem(unix) -Asystem(posix) -Acpu(i386) -Amachine(i386) -D__i486__
principal.c /tmp/cca00581.i
GNU CPP version 2.7.2 (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/i486-linux/include
 /usr/lib/gcc-lib/i486-linux/2.7.2/include
 /usr/include
End of search list.
 /usr/lib/gcc-lib/i486-linux/2.7.2/cc1 /tmp/cca00581.i -quiet -dumpbase
principal.c -version -o /tmp/cca00581.s
GNU C version 2.7.2 (i386 Linux/ELF) compiled by GNU C version 2.7.2.
 as -V -Qy -o principal.o /tmp/cca00581.s
GNU assembler version 2.7 (i586-unknown-linux), using BFD version 2.7.0.2

linux>  gcc -v -c mafonc.c
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2/specs
gcc version 2.7.2
 /usr/lib/gcc-lib/i486-linux/2.7.2/cpp -lang-c -v -undef -D__GNUC__=2
-D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux -D__ELF__ -D__unix__
 -D__i386__ -D__linux__ -D__unix -D__i386 -D__linux -Asystem(unix)
-Asystem(posix) -Acpu(i386) -Amachine(i386) -D__i486__ myfonc.c
/tmp/cca00587.i
GNU CPP version 2.7.2 (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
```

```
 /usr/local/include
 /usr/i486-linux/include
 /usr/lib/gcc-lib/i486-linux/2.7.2/include
 /usr/include
End of search list.
 /usr/lib/gcc-lib/i486-linux/2.7.2/cc1 /tmp/cca00587.i -quiet -dumpbase
myfonc.c -version -o /tmp/cca00587.s
GNU C version 2.7.2 (i386 Linux/ELF) compiled by GNU C version 2.7.2.
 as -V -Qy -o myfonc.o /tmp/cca00587.s
GNU assembler version 2.7 (i586-unknown-linux), using BFD version 2.7.0.2

linux> gcc -v -o principal principal.o mafonc.o -lm
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2/specs
gcc version 2.7.2
 ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.1 -o principal
/usr/lib/crt1.o /usr/lib/crti.o /usr/lib/crtbegin.o
-L/usr/lib/gcc-lib/i486-linux/2.7.2
principal.o myfonc.o -lm -lgcc -lc -lgcc
/usr/lib/crtend.o /usr/lib/crtn.o
```

Notons dans cette édition de liens la présence des fameux fichiers crt :

- `/usr/lib/crt1.o` ;
- `/usr/lib/crti.o` ;
- `/usr/lib/crtbegin.o` ;
- `/usr/lib/crtend.o` ;
- `/usr/lib/crtn.o`.

4.3 L'édition de liens

Le travail de l'éditeur de liens concerne essentiellement la résolution des références internes et la résolution des références externes.

Résolution des références internes

Il arrive que des programmes soient obligés de faire des sauts en avant ou des sauts en arrière. Par exemple, lorsqu'une instruction du type « goto » ou « while » est exécutée, le programme doit ensuite exécuter une instruction qui n'est pas l'instruction située juste après, mais qui est située avant ou après dans le texte du programme.

Les sauts en arrière ne posent pas de problème car le compilateur connaît déjà l'endroit où se situe la prochaine instruction à exécuter : comme il traite le fichier source du début vers la fin (ce qui va de soi), il a déjà traité cette instruction et il connaît son adresse (voir section 4.5).

En revanche, lorsque le compilateur se trouve face à un saut en avant, il ne connaît pas encore la prochaine instruction à exécuter. Lorsqu'il arrive à l'endroit du fichier où

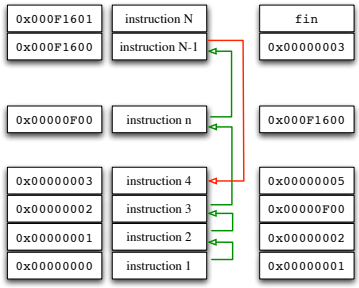


FIGURE 4.12 – *Chaque instruction donne l'adresse à laquelle se rendre une fois son exécution achevée. Un programme peut contenir des sauts avant et des sauts arrière.*

se trouve cette instruction, il ne peut plus indiquer son adresse à l'instruction effectuant le saut en avant, à moins de revenir en arrière ou d'effectuer une seconde passe. Il se trouve que la plupart des compilateurs ne reviennent pas en arrière et n'effectuent qu'une seule passe, laissant la résolution de ce genre de référence à l'éditeur de liens.

Résolution des références externes

Les références externes sont les références faites à des variables externes ou à des fonctions qui ne sont pas définies dans le fichier C traité. Par exemple, dans `principal.c`, **`myfonc()`** et **`printf()`** sont des références externes. Même si apparemment, nous n'utilisons pas de variables externes dans `principal.c`, il est fort probable que des déclarations de variables externes soient faites dans les fichiers d'en-tête inclus.

Pour que l'éditeur de liens puisse résoudre ces références, le compilateur inclut dans chaque fichier objet l'ensemble des références utilisées par le fichier source correspondant. Ces références sont classées en deux catégories : les références connues, c'est-à-dire définies dans le fichier objet et qui peuvent être appelées dans d'autres fichiers objet, et les références inconnues qui doivent être définies ailleurs, c'est-à-dire soit dans un des autres fichiers objet, soit dans une bibliothèque.

Ces informations sont écrites dans une table que l'on nomme table des symboles et il est possible, par exemple grâce à la commande `nm` sous Unix, de lire la table des symboles d'un fichier objet ou d'une bibliothèque. Voici par exemple la table des symboles de `principal.o` (T pour *text* signifie que le symbole est connu, U qu'il est inconnu) :

```
linux> nm principal.o
00000000 t gcc2_compiled.
00000000 T main
```

```
        U myfonc
        U printf
        U sqrt
```

Les fichiers des bibliothèques comportent aussi une table des symboles et on peut donc aussi la lire. Voici un bref extrait de la table des symboles de la bibliothèque mathématique :

```
linux> nm /usr/lib/libm.so
...
0000555c T sin
00005584 T sinh
000055e4 T sinhl
00005644 T sinl
0000566c T sqrt
000056ac T sqrtl
000056ec T tan
00005710 T tanh
0000576c T tanhl
000057c8 T tanl
...
```

Lorsque l'éditeur de liens rencontre une fonction qui est définie dans un des autres fichiers objet, il doit juste faire le lien (d'où son nom) entre l'appel de la fonction et le code de la fonction. En revanche, lorsqu'il ne trouve pas le code de la fonction, il doit chercher ce dernier dans les bibliothèques qui lui ont été spécifiées sur la ligne de commande. Si jamais il ne trouve pas le symbole qu'il cherche dans les tables des symboles de ces bibliothèques, il ne peut pas continuer son travail et il retourne généralement le message « *undefined symbol* ».

En revanche, lorsqu'il trouve le symbole adéquat dans une des bibliothèques, l'éditeur de liens a trois solutions pour créer l'exécutable :

1. Il inclut toute la bibliothèque ;
2. Il n'inclut que la partie indispensable de la bibliothèque ;
3. Il n'inclut rien, mais indique où trouver le code indispensable.

Ce choix ne dépend pas de l'éditeur de liens, mais du format dans lequel les bibliothèques ont été écrites et de la stratégie adoptée par le système d'exploitation. Les exécutables produits avec des bibliothèques du premier type sont énormes et cette méthode a vite été abandonnée. Par exemple, si nous utilisions de telles bibliothèques pour notre programme, il contiendrait le code de toutes les fonctions trigonométriques, exponentielles, etc.

Dans le second cas, les bibliothèques sont des bibliothèques statiques. Ce type de bibliothèques permet de produire des exécutables beaucoup plus petits, mais tout de même assez volumineux. Par ailleurs, cette méthode comporte un petit défaut : lorsque

deux programmes font appel aux mêmes fonctions, par exemple la fonction **`sqrt()`**, le code de cette fonction sera écrit dans chacun des deux exécutables et, lorsque ces deux programmes seront exécutés, sera donc chargé en double dans la mémoire de l'ordinateur .

Le troisième cas évite justement cet inconvénient, mais requiert des bibliothèques particulières : les bibliothèques partagées ou bibliothèques dynamiques. Lorsqu'un programme est exécuté, le système d'exploitation vérifie si le code de la fonction à exécuter n'est pas déjà en mémoire. Si c'est le cas, il se contente de l'utiliser, sinon il charge cette fonction en mémoire directement à partir de la bibliothèque. On parle alors de résolution dynamique de symboles et la plupart des systèmes d'exploitation modernes utilisent ce type de bibliothèques. Sous Unix, il existe un utilitaire, `ldd`, qui indique pour un exécutable donné les bibliothèques dont ce dernier aura besoin au moment de l'exécution.

```
linux> ldd /usr/local/bin/emacs
        libXaw3d.so.6 => /usr/X11R6/lib/libXaw3d.so.6
        libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6
        libXt.so.6 => /usr/X11R6/lib/libXt.so.6
        libSM.so.6 => /usr/X11R6/lib/libSM.so.6
        libICE.so.6 => /usr/X11R6/lib/libICE.so.6
        libXext.so.6 => /usr/X11R6/lib/libXext.so.6
        libX11.so.6 => /usr/X11R6/lib/libX11.so.6
        libncurses.so.3.0 => /usr/lib/libncurses.so.3.0
        libm.so.5 => /lib/libm.so.5.0.6
        libc.so.5 => /lib/libc.so.5.3.12
        libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6.0
        libXt.so.6 => /usr/X11R6/lib/libXt.so.6.0
        libSM.so.6 => /usr/X11R6/lib/libSM.so.6.0
        libICE.so.6 => /usr/X11R6/lib/libICE.so.6.0
        libXext.so.6 => /usr/X11R6/lib/libXext.so.6.0
        libX11.so.6 => /usr/X11R6/lib/libX11.so.6.0
```

La création de bibliothèques partagées est délicate et a poussé bon nombre de développeurs de système d'exploitation à adopter de nouveaux formats comme le format ELF (*Executable and Linking Format*) pour le stockage des bibliothèques. Sans entrer dans les détails, les bibliothèques dans ces nouveaux formats sont plus proches d'un exécutable que ne l'étaient les bibliothèques dans les formats anciens (comme a.out ou COFF).

La création de bibliothèques partagées n'est valable que pour des programmes réentrants. Un programme est dit réentrant s'il ne se modifie pas lui-même en cours d'exécution, c'est-à-dire s'il ne modifie ni son code, ni ses données (variables locales). Cette propriété permet à deux programmes d'exécuter la même copie de code sans se préoccuper des modifications éventuelles du code ou des données de l'un par l'autre.

Sous les systèmes Unix, chaque processus utilise son propre espace d'adressage et une copie des données est effectuée pour chaque processus. La propriété de réentrance se résume donc pour un programme à ne pas modifier son code en cours de route.

4.4 Structure des fichiers produits par compilation

Nous allons dans cette section détailler la structure des fichiers objet, des fichiers de bibliothèques et des exécutables. Nous nous fonderons pour cela sur des systèmes d'exploitation utilisant des formats récents, comme ELF.

Ces formats offrent un avantage majeur par rapport aux formats plus anciens : les trois types de fichiers qui nous intéressent ont la même forme et, structurellement, il n'y a pas de différence entre un fichier objet, un fichier de bibliothèque et un fichier exécutable.

Les fichiers objet et les fichiers de bibliothèques au format ELF sont donc très proches d'un exécutable et il ne reste pas grand chose à faire pour les transformer en exécutable. Cela facilite notamment les résolutions dynamiques de symbole telles qu'elles sont pratiquées avec les bibliothèques partagées.

Structure commune

Les fichiers au format ELF sont constitués d'un en-tête, de plusieurs sections de structure identique et d'une section contenant des informations optionnelles (voir figure 4.13).

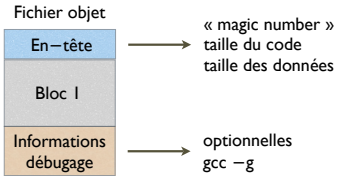


FIGURE 4.13 – *Structure d'un fichier objet.*

La section optionnelle contient les informations utiles pour le débogage des programmes et, sauf demande explicite au compilateur[9], elle n'est généralement pas incluse car elle n'est pas nécessaire pour le fonctionnement du programme.

L'en-tête contient de nombreuses informations permettant de retrouver les données stockées dans le fichier. Elle débute généralement par un nombre (*magic number*) qui indique le type du fichier considéré, puis elle continue en spécifiant le nombre de sections présentes dans le fichier. Enfin, elle précise un certain nombre d'informations

9. La fameuse option `-g`...

qui seront utiles au moment de l'exécution, comme la taille du programme contenu, la taille des données initialisées et les tailles des données non initialisées. Ces informations permettront au système d'exploitation d'allouer la mémoire nécessaire au bon déroulement de l'exécution.

Certains utilitaires, comme `size` sous Unix, permettent de lire les informations contenues dans l'en-tête. Voici par exemple le résultat de l'exécution de la commande `size principal` :

```
linux> size principal
text    data    bss     dec     hex     filename
568     953     4       1525    5f5     principal
```

La structure interne des sections suivantes est un peu plus complexe et varie d'un format à l'autre. Généralement, elle se compose de plusieurs tables et de plusieurs segments (voir figure 4.14) :

- la table des symboles connus ;
- la table des symboles inconnus ;
- la table des chaînes de caractères ;
- le segment de texte (parfois appelé segment de code ou segment de programme) ;
- le segment des données initialisées.

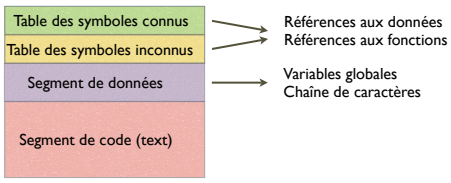


FIGURE 4.14 – *Structure des blocs des fichiers objet.*

Il n'y a pas de segment des données non initialisées car ces dernières sont créées dynamiquement au moment de l'exécution.

Les fichiers objet

Les fichiers objet ne contiennent généralement qu'une seule section et prennent donc la forme suivante :

- un en-tête ;
- la table des symboles connus ;
- la table des symboles inconnus ;
- la table des chaînes de caractères ;
- le segment de texte ;
- le segment des données initialisées ;
- la section optionnelle des informations de débogage.

Les fichiers exécutables

Les fichiers exécutables contiennent plusieurs sections et, en fait, chaque section correspond au corps d'un fichier objet : l'édition de liens se contente donc de concaténer les différentes sections des fichiers objet et de résoudre les références internes et externes.

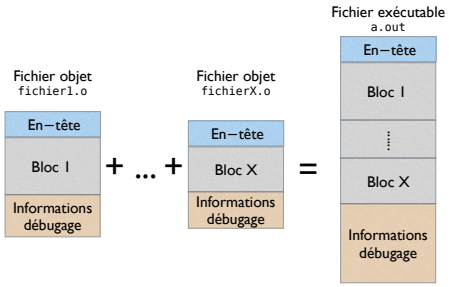


FIGURE 4.15 – *Structure d'un fichier exécutable.*

Cette opération est néanmoins complexe et demande en particulier d'indiquer pour chaque symbole inconnu d'un fichier objet où se trouve le code correspondant. Il faut donc remplacer les occurrences des symboles inconnus par les adresses du programme correspondant.

Nous verrons d'ailleurs dans une des sections suivantes que la détermination des adresses du programme pose quelques problèmes.

Les fichiers de bibliothèques

Les fichiers de bibliothèques ressemblent beaucoup aux fichiers exécutables, mais ils ne possèdent pas de fonction `main()`. Il se composent de plusieurs sections correspondant aux différents fichiers objet utilisés pour créer la bibliothèque.

4.5 Les problèmes d'adresses

Comme nous l'avons déjà annoncé et comme nous le verrons dans le chapitre sur les processus, chaque processus possède son propre espace mémoire et a ainsi l'illusion d'avoir la machine pour lui tout seul.

Les processus ne peuvent donc pas savoir quelle partie de la mémoire physique ils utilisent. Une conséquence de ce principe est qu'il n'est pas possible pour un fichier exécutable de prédire l'endroit où le code et les données du programme qu'il contient seront copiés en mémoire.

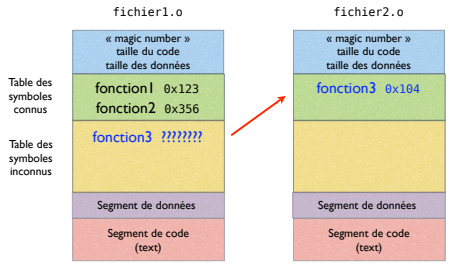


FIGURE 4.16 – *Résolution des symboles inconnus par l'édition de liens.*

Même si cela était possible, il ne serait pas très intéressant que cet exécutable contienne des adresses physiques absolues pour les différents éléments qu'il faudra copier en mémoire au moment de l'exécution. Supposons, en effet, qu'un programme pense utiliser une zone mémoire déterminée. D'une part, rien ne lui assure que cette zone sera libre au moment de l'exécution et, d'autre part, rien ne lui assure que cette zone existe physiquement dans la machine. En utilisant cette méthode, il serait par exemple très dangereux de compiler un programme sur une machine et de le copier sur une autre.

En pratique, donc, les exécutables ne contiennent pas les adresses physiques du code et des données qu'il faudra copier en mémoire au moment de l'exécution. En revanche, il est nécessaire que chaque exécutable puisse repérer les différentes instructions du code (par exemple) : sans cette faculté, il n'est pas possible de faire des sauts en avant ou en arrière ou de demander l'exécution d'une fonction. Formulé différemment, chaque fois que la prochaine instruction à exécuter ne se trouve pas sur la ligne suivante, il faut disposer d'un moyen pour la retrouver.

L'éditeur de liens utilise pour cela des adresses relatives : il suppose qu'effectivement le processus correspondant dispose de tout l'espace mémoire et que donc il peut commencer à copier le code à l'adresse 0. Ainsi, nous avons vu au moment de l'exécution de la commande `nm principal.o` (section 4.3) que la fonction **`main()`** était située à l'adresse 0.

Au moment de l'exécution, il est donc nécessaire d'effectuer une translation d'adresse, c'est-à-dire qu'il faut ajouter à chaque adresse de l'exécutable l'adresse (physique ou virtuelle) correspondant au début de la zone mémoire effectivement utilisée.

Cette opération peut être très longue, car elle est effectuée dynamiquement lors de la demande d'exécution et car l'exécutable peut contenir d'innombrables instructions. Cependant, les systèmes d'exploitation utilisant des pages (ou des segments) de mémoire disposent d'un moyen élégant pour résoudre ce problème : il suffit que les adresses présentes dans l'exécutable correspondent au décalage (*offset*) d'adresse dans

une page et que la zone où le code sera copié corresponde à une page : ainsi, il suffit de spécifier le numéro de page pour que toutes les adresses soient automatiquement traduites par la MMU !

4.6 Les appels système

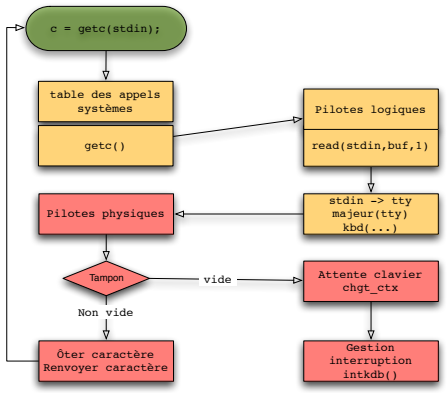


FIGURE 4.17 – *Un appel à* `getc`

Pour terminer ce chapitre sur la compilation, nous allons nous intéresser à ce qui se passe lorsqu'un programme utilise un appel système. Considérons le fichier source `principal2.c` suivant :

```
#include <sys/utsname.h>

int main(int argc, char *argv[])
{
  struct utsname buf;

  if(uname(&buf)==0)
    {
      printf("sysname : %s\n", buf.sysname);
      printf("nodename : %s\n", buf.nodename);
      printf("release : %s\n", buf.release);
      printf("version : %s\n", buf.version);
      printf("machine : %s\n", buf.machine);
    }
}
```

Ce fichier utilise l'appel système **`uname()`** qui permet d'obtenir des informations sur le système d'exploitation et la machine utilisée. Voici, par exemple, le résultat d'une exécution de ce programme sous un PC 486 avec Linux 2.0.18 :

```
linux> ./principal2
sysname : Linux
nodename : menthe22
```

```
release : 2.2.14-15mdksecure
version : #1 SMP Tue Jan 4 21:15:44 CET 2000
machine : i586
```

Après avoir compilé ce programme avec des bibliothèques dynamiques, observons maintenant la table des symboles indéfinis de l'exécutable correspondant :

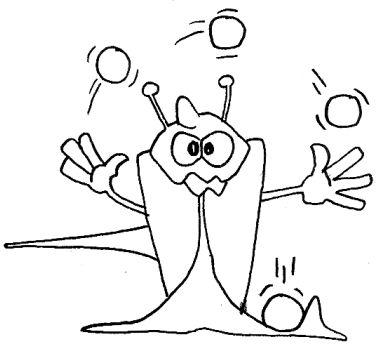
```
linux> nm -u principal2
__libc_init
__setfpucw
atexit
exit
printf
uname
```

L'appel système **`uname()`** est présent dans cette table des symboles indéfinis. Ainsi, l'utilisation d'appels système est strictement identique à l'utilisation de fonctions des bibliothèques standard, comme par exemple **`printf()`** : la résolution de ces références s'effectue dynamiquement à l'exécution. Comme les appels système font partie du code du noyau, il n'est pas nécessaire de les charger en mémoire (puisque le noyau est déjà chargé en mémoire) et la résolution consiste simplement à indiquer l'adresse du code correspondant en mémoire. Ceci étant, il est parfois difficile de se rendre compte du nombre de fonctions mises en œuvre pour réaliser certains appels (voir fig. 4.17).

Néanmoins, il y a au moins deux grandes différences entre un appel système et une fonction de bibliothèque : d'une part, l'appel système sera exécuté en mode noyau et, d'autre part, le code correspondant à l'appel système est déjà chargé en mémoire (au moins pour les systèmes d'exploitation monolithiques) car il correspond à une partie du code du noyau.

5

La gestion des processus



Un processus est l'abstraction d'un programme en cours d'exécution. Une abstraction est toujours difficile à représenter et c'est ce qui explique que bon nombre d'ouvrages restent très discrets sur la gestion des processus. Toutefois, le concept de processus est capital pour la réalisation d'un système d'exploitation efficace et nous tenterons dans ce chapitre d'éclaircir cette notion.

La notion de processus n'a cependant de sens que pour les systèmes d'exploitation fonctionnant en temps partagé ou en multiprogrammation. Si un système d'exploitation ne peut pas commuter d'un processus `A` à un processus `B` avant que le processus `A` ne se termine, il est inutile de concevoir une gestion complexe des processus : il suffit de les exécuter dans l'ordre. Cette remarque insiste sur une caractéristique mise en avant dans ce document : un système d'exploitation digne de ce nom doit fonctionner en temps partagé !

Les systèmes qui sont de plus multi-utilisateurs doivent offrir une gestion de processus plus élaborée, qui comprend notamment la protection des processus d'un utilisateur contre les actions d'un autre utilisateur.

Parmi les différents systèmes multi-tâches et multi-utilisateurs disponibles actuellement, nous avons choisi Unix pour illustrer nos propos. D'une part ce système propose de nombreux utilitaires qui seront facilement mis en œuvre par un lecteur intéressé, d'autre part, il dispose de nombreux appels système qui nous permettront d'écrire

pendant les travaux pratiques des programmes réalisant des tâches très proches de celles du système.

Mots clés de ce chapitre : processus, contexte, changement de contexte (*context switching*), ordonnancement (*scheduling*), segment de texte, segment de données, pile, tas.

5.1 Qu'est-ce qu'un processus ?

Un processus est l'abstraction d'un programme en train d'être exécuté. Afin de bien saisir la notion de processus, la plupart des ouvrages traitant des systèmes d'exploitation font appel à une analogie célèbre et nous la citerons ici essentiellement parce que cette analogie fait désormais partie de la culture générale en informatique.

Une petite analogie

Considérons un père attentionné qui souhaite préparer un gâteau d'anniversaire pour son fils. Comme la cuisine n'est pas son domaine de prédilection, il se munit d'une recette de gâteau et il rassemble sur son plan de travail les différents ingrédients : farine, sucre, etc. Il suit alors les instructions de la recette pas à pas et commence à préparer le gâteau.

Dans cette analogie, la recette représente un programme, c'est-à-dire une suite d'instructions que l'on doit suivre pour réaliser le travail voulu. Les ingrédients sont les données du programme et le père attentionné tient le rôle de l'ordinateur.

À ce moment là, le fils de notre père attentionné arrive en pleurant car il s'est égratigné le coude. Le père décide alors de lui porter secours, mais il annote auparavant sur sa recette la ligne qu'il était en train d'exécuter. Comme la médecine n'est pas non plus son fort, notre père attentionné se procure un livre sur les premiers soins et soigne son fils en suivant les instructions du livre.

Ce que vient de faire le père correspond à ce qui se passe pour les systèmes d'exploitation à temps partagé ou à multiprogrammation : le père a interrompu le processus de cuisine alors que celui-ci n'était pas terminé pour débuter le processus de soins. Comme il a marqué l'endroit de la recette où il s'est arrêté et comme il a conservé tous les ingrédients, il pourra reprendre le processus de cuisine quand il voudra, par exemple dès qu'il aura terminé de soigner son fils.

L'annotation faite par le père sur la recette pour marquer l'endroit où il s'est arrêté représente le contexte d'exécution du processus. Un processus est ainsi caractérisé par un programme (la recette), des données (les ingrédients) et un contexte courant.

En pratique, l'annotation qui permet de déterminer la prochaine instruction à exécuter d'un programme s'appelle le compteur ordinal ou compteur d'instructions. Le contexte d'un processus ne se résume pas simplement à ce compteur et il comprend d'autres paramètres que nous décrivons dans ce chapitre. L'essentiel, pour l'instant, est de bien comprendre la notion de processus.

Le partage du temps

Maintenant que le concept de processus est clair, reprenons l'explication du partage du temps ou de l'exécution concurrente de plusieurs processus. Notons à ce sujet que l'on parle souvent par abus de langage de « l'exécution concurrente de processus », ce qui est structurellement incorrect : un processus est déjà l'exécution d'un programme. La périphrase adéquate serait « l'existence concurrente de processus ».

Supposons que trois programmes doivent être exécutés sur notre ordinateur. À un instant donné `t`, seul un des processus représentant ces programmes disposera du processeur pour exécuter les instructions de son programme (les instructions du segment de texte). Pour faciliter l'explication, nous nommerons `A`, `B` et `C` les trois processus représentant l'exécution des trois programmes.

Supposons alors que nous découpions le temps en petites unités de l'ordre de la milliseconde (souvent appelées quanta) et que nous attribuions une unité de temps à chaque processus. Pendant le laps de temps qui lui est accordé, `A` pourra exécuter un certain nombre d'instructions. Une fois ce laps de temps écoulé, `A` sera interrompu et il sera nécessaire de sauvegarder toutes les informations dont `A` a besoin pour reprendre son exécution (son contexte), en particulier le compteur d'instructions de `A`.

Un autre processus peut alors utiliser le processeur pour exécuter les instructions de son programme. Quel processus choisir ? `A`, `B` ou `C` ? Ce choix est assuré par le système d'exploitation [1] et la façon dont les processus sont choisis s'appelle l'ordonnancement ou le *scheduling*. On parle aussi parfois de *scheduler* pour désigner la partie du système d'exploitation qui prend en charge l'ordonnancement.

Le processus `A` vient d'être interrompu. Pourquoi alors se demander si `A` ne pourrait pas être le prochain processus à choisir ? Parce qu'il est possible que `B` et `C` soient dans des états particuliers qui ne requièrent pas l'utilisation du processeur. Par exemple, nous pouvons imaginer que `B` est un éditeur de texte et que `C` est un lecteur de courrier (*e-mail*) dont les utilisateurs sont partis discuter dans le couloir. `B` et `C` n'ont donc rien à faire, si ce n'est d'attendre le retour de leurs utilisateurs et il n'est donc pas nécessaire de leur attribuer le processeur.

Pourquoi alors avoir interrompu `A` pour lui rendre la main ? Nous pourrions poser la question différemment : est-il possible de savoir si d'autres processus ont besoin du processeur avant d'interrompre le processus courant ? Si on accepte l'idée que seul le système d'exploitation a accès aux ressources permettant de faire fonctionner la machine, alors la réponse à cette question est non ! En effet, le système d'exploitation est un programme et il faut lui donner la main pour qu'il puisse choisir le prochain processus qui se verra attribuer le processeur. Donc, il est nécessaire d'interrompre `A` pour laisser le système d'exploitation prendre cette décision. Cette interruption n'est nécessaire que dans le cadre des microprocesseurs mono-cœur.

En pratique, cette politique ne peut être efficace que si le système d'exploitation utilise beaucoup moins d'un quantum pour sélectionner le prochain processus...

1. L'ordonnancement des processus est généralement directement effectuée par le noyau du système d'exploitation.

Reprenons notre ordonnancement : `A` vient d'être interrompu. Supposons que `B` soit choisi. Tout d'abord, il faut rétablir le contexte de `B`, c'est-à-dire remettre le système dans l'état où il était au moment où `B` a été interrompu. L'opération consistant à sauvegarder le contexte d'un processus, puis à rétablir le contexte du processus suivant s'appelle le changement de contexte (*context switching*) et est essentiellement assurée par les couches matérielles de l'ordinateur.

Après rétablissement de son contexte, le processus `B` va disposer du processeur pour un quantum et pourra exécuter un certain nombre d'instructions de son programme. Ensuite, `B` sera interrompu et un autre processus (éventuellement le même) se verra attribuer le processeur. Ce principe est résumé sur la figure 5.1

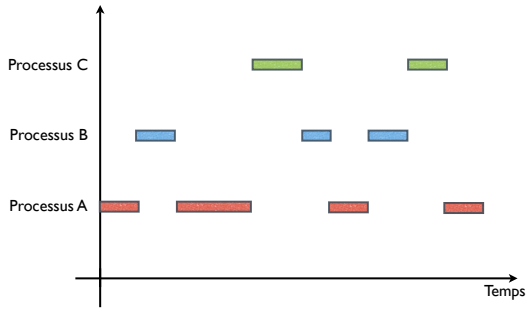


FIGURE 5.1 – *Principe de l'ordonnancement des processus : chaque processus dispose du processeur pendant un certain temps, mais, à chaque instant, un seul processus peut s'exécuter.*

Ainsi, à tout instant, un seul processus utilise le processeur et un seul programme s'exécute. Mais si nous observons le phénomène à l'échelle humaine, c'est-à-dire pour des durées de l'ordre de la seconde, nous avons l'illusion que les trois programmes correspondant aux trois processus `A`, `B` et `C` s'exécutent en même temps. Cette exécution est en fait concurrente, car les unités de temps attribuées à un processus ne servent pas aux autres.

Il reste cependant un problème dans cette belle mécanique : comment le processus correspondant au système d'exploitation est-il choisi ? En effet, nous avons dit que lorsqu'un processus est interrompu, le système d'exploitation sélectionne le prochain processus qui bénéficiera du processeur. Mais comment le processus du système d'exploitation a-t-il eu la main ? Qui a sélectionné ce processus ? La plupart des systèmes utilisent en fait des interruptions matérielles, provoquées par une horloge, qui ont lieu régulièrement (tous les 1/50^{e} de seconde, par exemple). Il suffit alors de choisir le noyau du système d'exploitation comme routine associée à cette interruption...

Précision sur les problèmes de compatibilité

Dans le chapitre consacré aux systèmes d'exploitation actuels, nous avons évoqué le fait que certains systèmes d'exploitation ne pouvaient pas réaliser de partage du temps pour des raisons de compatibilité avec de vieux programmes. Cette remarque laisse supposer qu'un programme doit être prévu pour fonctionner sur un système d'exploitation à temps partagé, ce qui est contradictoire avec la notion de multi-tâches préemptif.

Nous allons dans cette courte section détailler le problème.

Supposons qu'un programme, ignorant les capacités des systèmes d'exploitation modernes, décide d'écrire directement sur un lecteur de disquettes. Les lecteurs de disquettes mettent le support magnétique de la disquette en rotation sur lui-même pour lire et écrire dessus. Cependant, et afin de ne pas user prématurément les disquettes, le lecteur stoppe cette rotation quand on ne fait pas appel à lui. Pour écrire sur une disquette, il faut donc d'abord remettre le support en rotation, attendre que la vitesse se stabilise et ensuite envoyer les données à écrire.

Notre programme demande donc d'abord au lecteur de remettre le support en rotation, puis il effectue une boucle vide pour attendre que la vitesse se stabilise et, enfin, il envoie les données à écrire.

Supposons maintenant que nous décidions d'utiliser ce programme sur un ordinateur doté d'un système d'exploitation moderne, donc multi-utilisateurs, multi-tâches, avec mémoire virtuelle et protection de la mémoire. Tout d'abord, il faudra trouver un moyen pour permettre au programme d'accéder directement à la disquette, ce qui est contraire à la philosophie du système d'exploitation et ce qui suppose donc qu'on laisse une faille dans la protection des ressources.

Ensuite, comme les exécutions sont concurrentes et que le système est à temps partagé, rien n'assure que notre programme aura la main après avoir effectué sa boucle d'attente : il se peut que d'autres processus se voient attribuer le processeur. Ainsi, il se peut que le lecteur de disquettes se remette en pause avant que notre programme n'ait eu le temps d'envoyer ses données. Au moment où il les enverra, le lecteur ne sera pas prêt et les données seront probablement perdues.

Comme les actions d'un programme sont imprévisibles par le système d'exploitation, les systèmes cherchant à maintenir la compatibilité avec de vieux programmes sont obligés de laisser de nombreuses failles dans la protection des ressources, en particulier de la mémoire, et sont contraints de ne pas interrompre ces programmes.

Nous espérons qu'à ce stade de la lecture du document, tous les lecteurs connaissent la démarche à suivre pour effectuer le travail de notre programme exemple. Nous allons cependant la préciser...

Tout d'abord, il ne faut pas essayer d'accéder directement aux ressources de la machine, comme le lecteur de disquettes, mais il faut faire appel au système d'exploitation via l'interface qu'il nous propose. Non seulement, c'est beaucoup plus simple car le système prend en compte lui-même toutes les considérations spécifiques, comme la stabilisation de la vitesse de rotation ou les temps de mise en pause, mais en plus,

cela ne dépend pas du support sur lequel on écrit. Il y a donc de grandes chances que notre programme soit très général et qu'il survive aux changements de matériels ou de système d'exploitation : seule l'interface compte.

Ensuite, il ne faut pas effectuer sa propre gestion du temps, par l'intermédiaire de boucles vides ou d'autres moyens suspects. Il est possible de demander au système d'exploitation de signaler une durée écoulée ou un délai dépassé. Non seulement c'est beaucoup plus simple que de le faire soi-même, mais en plus, c'est le seul moyen d'obtenir des temps précis (temps à la microseconde près !). En outre, le système d'exploitation sait alors si le programme doit attendre sans rien faire et il évitera en conséquence de lui attribuer le processeur pendant ce temps-là.

5.2 La hiérarchie des processus

Les systèmes d'exploitation modernes permettent la création et la suppression dynamique des processus (c'est la moindre des choses !). Plusieurs stratégies sont employées pour cela : par exemple, sous Unix, le nouveau processus est créé par duplication ou clonage (appel système **`fork()`**) d'un autre processus, puis par écrasement du programme et des données du processus clone par le programme et les données du processus que l'on souhaite finalement créer (appel système **`exec()`**).

Certains systèmes préfèrent créer le nouveau processus de toutes pièces sans faire appel au procédé de clonage.

Une arborescence de processus

Toutes les stratégies employées sont fondées sur le même principe : tout processus est créé par un autre processus et il s'établit donc une filiation de processus en processus. Généralement, on appelle processus père le processus qui en crée un autre et processus fils le processus ainsi créé.

Comme chaque processus fils n'a qu'un seul père, cette hiérarchie des processus peut se traduire sous forme d'arbre. En revanche, comme un processus père peut avoir plusieurs fils, le nombre de branches en chaque nœud de l'arbre n'est pas déterminé.

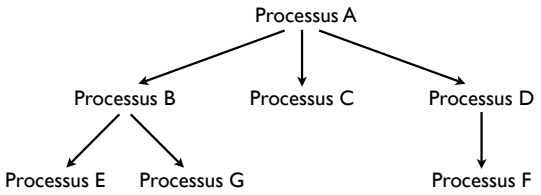


FIGURE 5.2 – *Arborescence des processus.*

Notons que la notion de père et de fils est relative : un processus père ayant plusieurs fils a lui-même un père et, pour ce processus père, c'est un processus fils.

L'exemple d'Unix

Sous Unix, un numéro est attribué à tous les processus ainsi créés et ce numéro reflète l'ordre dans lequel ils ont été créés : le premier processus créé porte le numéro 1, le suivant le numéro 2, etc. Ce numéro est donc unique et il permet d'identifier les différents processus. Il s'appelle généralement le *Process IDentifier* ou `pid`.

Le `pid` est nombre dont la valeur maximale est en général fonction du système d'exploitation. Sous certains Unix (notamment Linux), cette valeur maximale est consignée dans un fichier spécial [2] : `/proc/sys/kernel/pid_max`. Les numéros de `pid` étant affectés de manière séquentielle, une fois ce nombre atteint, l'affectation repart de 0 en évitant tous les numéros de processus encore actif.

La commande `ps` permet de visualiser les différents processus d'un ordinateur. Voici, par exemple, le résultat de cette commande :

```
linux> ps
  PID TTY STAT  TIME COMMAND
  169  p2 S    0:00 -csh
  170  p1 S    0:00 -csh
  189  p1 S   11:11 emacs Compilation/compil.tex
 1513  p1 R    0:00 ps
```

Par défaut, cette commande ne donne qu'une partie des processus qui ont été créés par l'utilisateur qui demande la visualisation. Différentes options permettent de visualiser les autres processus présents. Par exemple, voici le résultat de la commande [3] `ps -ax` :

```
linux> ps -ax
  PID TTY STAT  TIME COMMAND
    1  ?  S    0:01 init
    2  ?  SW   0:00 (kflushd)
    3  ?  SW<  0:00 (kswapd)
    4  ?  SW   0:00 (nfsiod)
    5  ?  SW   0:00 (nfsiod)
    6  ?  SW   0:00 (nfsiod)
    7  ?  SW   0:00 (nfsiod)
   21  ?  S    0:00 /sbin/kerneld
   77  ?  S    0:00 syslogd
   86  ?  S    0:00 klogd
   97  ?  S    0:00 crond
  109  ?  S    0:00 lpd
  125  ?  S    0:00 gpm -t MouseMan
  138   1 S    0:00 /sbin/mingetty tty1
  139   2 S    0:00 /sbin/mingetty tty2
  140   3 S    0:00 /sbin/mingetty tty3
  141   4 S    0:00 /sbin/mingetty tty4
  142   5 S    0:00 /sbin/mingetty tty5
  143   6 S    0:00 /sbin/mingetty tty6
  144  ?  S    0:00 /usr/bin/X11/xdm -nodaemon
  146  ?  S    0:00 update (bdflush)
```

2. Ce fichier n'a pas d'existence sur le disque dur ou sur un périphérique de stockage. Il correspond aux différentes valeurs utilisées par le système d'exploitation et permet, dans certains cas, de réaliser des changements en cours de fonctionnement. L'arborescence `/proc` est ce que l'on appelle un système de fichiers virtuels.

3. Suivant les versions d'Unix, les options des commandes peuvent varier. Sur les Unix System V, la commande correspondante est : `ps -ef`. Dans le doute : `man ps`.

```
  148  ?  R    0:53 /usr/X11R6/bin/X -auth /usr/X11R6/lib/X11/xdm/A:0-a00144
  149  ?  S    0:00 -:0
  163  ?  S    0:00 xterm -sb -sl 1200 -j -cu -name Commande -geometry 80x7+120
  164  ?  S    0:02 xterm -sb -sl 1200 -j -cu -name Terminal -geometry 80x23+12
  157  ?  S    0:00 sh /usr/X11R6/lib/X11/xdm/Xsession
  162  ?  S    0:00 xclock -geometry 100x100+10+10 -display :0
  166  ?  S    0:01 fvwm
  167  ?  S    0:00 /usr/lib/X11/fvwm/GoodStuff 9 4 /home/gueydan/.fvwmrc 0 8
  168  ?  S    0:00 /usr/lib/X11/fvwm/FvwmPager 11 4 /home/gueydan/.fvwmrc 0 8 0
  169  p2 S    0:00 -csh
  170  p1 S    0:00 -csh
  189  p1 S   11:10 emacs Compilation/compil.tex
 1510  p1 R    0:00 ps -ax
```

Lorsqu'un utilisateur ouvre une session [4], un interprète de commandes (un *shell*) est exécuté et cet interprète de commande peut à son tour créer d'autres processus, comme ceux correspondant aux différentes fenêtres apparaissant sur l'écran juste après la procédure d'ouverture de session.

Si l'on se connecte sur une machine de l'ensta au travers d'une connexion sécurisée [5], on obtient un ensemble de processus relativement restreint :

```
ensta:22 ->ps ux
USER       PID  TIME COMMAND
bcollin  14112  0:00 sshd: bcollin@pts/1
bcollin  14113  0:00 -tcsh
```

Si l'on utilise maintenant une connexion directement sur l'écran d'accueil de la station, dans une des salles informatiques, on obtient un nombre de processus plus important. Dans l'exemple donné ci-dessous (obtenu grâce à la commande `ps -ux`), le processus de pid 688 est probablement l'ancêtre commun de tous les processus appartenant à l'utilisateur `gueydan`. Notons que rien ne permet de l'affirmer, car cet utilisateur a très bien pu se loguer deux fois et il y aurait à ce moment-là deux ensembles de processus appartenant à cet utilisateur et issus de deux pères différents.

```
USER       PID    TIME COMMAND
gueydan    688    0:00 [.xsession]
gueydan    739    0:00 xclock -geometry 100x100+10+10 -display :0
gueydan    740    0:00 xbiff -geometry 100x80+9+114 -display :0
gueydan    741    0:00 xterm -sb -sl 1200 -name Commande -geometry 80x9+120+10 -ls -C
gueydan    742    0:58 xterm -sb -sl 1200 -name Terminal -geometry 80x31+120-20 -ls
gueydan    746    0:09 fvwm2
gueydan    747    0:01 /usr/lib/X11/fvwm2/FvwmPager 7 4 .fvwm2rc 0 8 0 0
gueydan    750    0:00 [tcsh]
gueydan    751    0:02 [tcsh]
gueydan  32109   12:21 /usr/lib/netscape/netscape-communicator
gueydan  32119    0:00 (dns helper)
gueydan  14716    0:41 xemacs
gueydan  15559    0:01 xterm
gueydan  15565    0:00 -csh
gueydan  16470    3:48 xemacs poly.tex
gueydan  20875    0:08 xdvi.bin -name xdvi poly
gueydan  20882    0:05 gs -sDEVICE=x11 -dNOPAUSE -q -dDEVICEWIDTH=623 -dDEVICEHEIGHT=879
gueydan  25664    0:00 ps -ux --cols 200
```

Pour pouvoir retrouver le père d'un processus, celui-ci conserve parmi ses différents attributs le numéro d'identification de son père : le `ppid` (*Parent Process IDentifier*). Ainsi la commande `ps -jax` permet de visualiser tous les processus de la machine ainsi que l'identificateur de leur père :

4. On dit parfois qu'un utilisateur se « logue » en référence à la procédure d'ouverture de session des systèmes multi-utilisateurs, procédure qui demande l'identificateur de l'utilisateur ou *login* en anglais.

5. Par exemple en utilisant `ssh`.

```
 PPID   PID  PGID   SID TTY TPGID  STAT   UID   TIME COMMAND
    0     1     0     0  ?     -1  S       0   0:01 init
    1     2     1     1  ?     -1  SW      0   0:00 (kflushd)
    1     3     1     1  ?     -1  SW<     0   0:00 (kswapd)
    1     4     1     1  ?     -1  SW      0   0:00 (nfsiod)
    1     5     1     1  ?     -1  SW      0   0:00 (nfsiod)
    1     6     1     1  ?     -1  SW      0   0:00 (nfsiod)
    1     7     1     1  ?     -1  SW      0   0:00 (nfsiod)
    1    21    21    21  ?     -1  S       0   0:00 /sbin/kerneld
    1    77    77    77  ?     -1  S       0   0:00 syslogd
    1    86    86    86  ?     -1  S       0   0:00 klogd
    1    97    97    97  ?     -1  S       0   0:00 crond
    1   109   109   109  ?     -1  S       0   0:00 lpd
    1   125   125   125  ?     -1  S       0   0:00 gpm -t MouseMan
    1   138   138   138   1   138  S       0   0:00 /sbin/mingetty tty1
    1   139   139   139   2   139  S       0   0:00 /sbin/mingetty tty2
    1   140   140   140   3   140  S       0   0:00 /sbin/mingetty tty3
    1   141   141   141   4   141  S       0   0:00 /sbin/mingetty tty4
    1   142   142   142   5   142  S       0   0:00 /sbin/mingetty tty5
    1   143   143   143   6   143  S       0   0:00 /sbin/mingetty tty6
    1   144   144   144  ?     -1  S       0   0:00 /usr/bin/X11/xdm -nodaemon
    1   146   146   146  ?     -1  S       0   0:00 update (bdflush)
  144   148   148   144  ?     -1  S       0   0:53 /usr/X11R6/bin/X -auth /usr
  144   149   149   144  ?     -1  S       0   0:00 -:0
  149   157   157   144  ?     -1  S     500   0:00 sh /usr/X11R6/lib/X11/xdm/X
  157   162   157   144  ?     -1  S     500   0:00 xclock -geometry 100x100+10
  157   163   157   144  ?     -1  S       0   0:00 xterm -sb -sl 1200 -j -cu -
  157   164   157   144  ?     -1  S       0   0:02 xterm -sb -sl 1200 -j -cu -
  157   166   157   144  ?     -1  S     500   0:01 fvwm
  166   167   157   144  ?     -1  S     500   0:00 /usr/lib/X11/fvwm/GoodStuff
  166   168   157   144  ?     -1  S     500   0:00 /usr/lib/X11/fvwm/FvwmPager
  163   169   169   169  p2   169  S     500   0:00 -csh
  164   170   170   170  p1  1512  S     500   0:00 -csh
  170   189   189   170  p1  1512  S     500  11:10 emacs Compilation/compil.te
  170  1512  1512   170  p1  1512  R     500   0:00 ps -jax
```

Unix est un système multi-utilisateurs et chaque utilisateur est identifié par un numéro (`uid`, *User IDentifier*). Lorsqu'un programme est exécuté par un utilisateur, le processus correspondant appartient à cet utilisateur et possède donc comme attribut son `uid`. Seul le propriétaire d'un processus et le super-utilisateur de la machine peuvent effectuer des opérations sur ce processus, comme par exemple le détruire.

Pour des raisons pratiques, on identifie généralement aussi un utilisateur par un nom et une correspondance est faite entre ce nom et le numéro d'identification [6]. La commande `ps aux` permet par exemple de visualiser tous les processus présents sur la machine ainsi que leurs propriétaires :

```
ensta:22 ->ps aux
UID        PID  PPID      TIME CMD
root         1     0  00:00:05 init [5]
root         2     0  00:00:00 [kthreadd]
...
carrel    1830     1 911-01:31:43 /home/2012/carrel/IN104/traiterimage
...
root      2767     1  00:00:59 automount
nobody    2791     1  00:00:00 oidentd -q -u nobody -g nobody
...
ntp       2851     1  00:00:01 ntpd -u ntp:ntp -p /var/run/ntpd.pid -g
...
smmsp     2902     1  00:00:00 sendmail: Queue runner@01:00:00 for /var/spool/clientmqueue
...
canna     2932     1  00:00:01 /usr/sbin/cannaserver -syslog -u canna
...
omni      3024  3022  00:00:00 /usr/bin/omniNames -errlog /var/omniNames/error.log -logdir /var/omniNames
```

6. La commande `id bcollin` permet de connaître l'`uid` ainsi que les groupes auxquels appartient l'utilisateur dont on connaît le patronyme :`uid=4045(bcollin) gid=401(prof) groupes=401(prof)`.

```
...
haldaemon 3098     1  00:00:04 hald
...
carrel    3658     1  00:01:57 /usr/bin/artsd -F 10 -S 4096 -s 60 -m artsmessage -l 3 -f
arnoult   4125     1  00:00:00 /usr/libexec/bonobo-activation-server --ac-activate --ior-output-fd=17
...
haas      9457     1  00:00:00 /usr/libexec/bonobo-activation-server --ac-activate --ior-output-fd=17
...
bcollin  14345 14343  00:00:00 sshd: bcollin@pts/1
...
dupoiron 27053     1  00:00:00 /usr/libexec/bonobo-activation-server --ac-activate --ior-output-fd=17
denie    28172     1  00:00:00 /usr/bin/gnome-keyring-daemon -d
```

Le premier processus

Quel est le processus situé au sommet de la hiérarchie des processus ? En d'autres termes, quel est le processus ancêtre de tous les autres processus ? La création de processus par filiation ne peut en effet pas s'appliquer au premier processus créé : c'est l'histoire de l'œuf et de la poule ! En fait, le premier processus est créé au moment où l'ordinateur démarre et il fait généralement partie des processus représentant le noyau du système d'exploitation.

Ce premier processus est exécuté en mode noyau et effectue un certain nombre de tâches vitales. Par exemple, il s'empresse de programmer la MMU et d'interdire l'accès aux différentes structures qu'il met en place. Puis il crée un certain nombre de processus, exécutés en mode noyau, qui assurent les services fondamentaux du noyau. Ces processus peuvent alors à leur tour créer des processus système, exécutés en mode utilisateur, qui assureront d'autres services du système d'exploitation ou qui permettront aux utilisateurs d'utiliser la machine.

Le nombre de processus fonctionnant en mode noyau varie d'un système d'exploitation à l'autre. Sur des systèmes monolithiques, il ne devrait en théorie n'y en avoir qu'un, mais il y en a généralement quelques-uns. Par exemple sur le système 4.4BSD, seulement deux processus sont en mode noyau : le processus *pagedaemon* qui permet d'échanger une page située en mémoire principale avec une page en mémoire secondaire (voir chapitre sur la gestion de la mémoire) et le processus *swapper* qui décide quand procéder à un tel échange. Les numéros d'identification attribués à ces processus sont, pour des raisons historiques, 0 pour le *swapper* et 2 pour le *pagedaemon*. Le premier processus créé en mode utilisateur est le processus *init* qui se voit attribuer le numéro d'identification 1. Les numéros d'identification de ces trois premiers processus ne reflètent donc pas l'ordre de leur création, mais c'est généralement le seul contre-exemple [7].

7. Il existe des projets de rustines du noyau Linux pour faire en sorte que les `pid` soient attribués de manière aléatoire. Ces projets ont pour but la sécurité, mais sont souvent rejetés car, *dixit* Alan Cox : « it's just providing security through obscurity ». Les défendeurs de tels projets clament que tant que les logiciels seront développés par des êtres humains faillibles, il y aura des bugs et qu'un moyen comme un autre de prévenir des « exploits » serait de rendre aléatoire l'attribution des `pid`. Le débat reste ouvert !

Le démarrage de la machine

La façon dont le premier processus est créé dépend beaucoup des systèmes d'exploitation et des machines. Généralement, les ordinateurs possèdent une mémoire programmable non volatile appelée PROM (*Programmable Read Only Memory*). Dans cette PROM est stocké un programme qui est directement chargé en mémoire, puis exécuté au moment où l'ordinateur est allumé. L'opération consistant à démarrer l'exécution de ce programme s'appelle le *bootstrapping* [8] (amorçage en français).

Les PROM ont généralement des petites capacités et il est nécessaire de faire appel au constructeur de l'ordinateur pour les reprogrammer. Le système d'exploitation ne peut donc pas être stocké dans la PROM de la machine et il se trouve généralement sur le disque dur.

Lorsque la machine a démarré, le processeur sait comment accéder au disque dur, mais il ne sait pas comment est structuré le système de fichiers et il n'est donc pas capable de lire le fichier exécutable correspondant au système d'exploitation [9]. Sur de nombreuses machines Unix, cet exécutable s'appelle `vmunix` et se trouve très près du sommet de l'arborescence du système de fichiers.

Le programme contenu dans la PROM se contente donc généralement de lancer l'exécution d'un autre programme, stocké au tout début du disque dur sous une forme ne dépendant pas du système de fichiers. Ce programme, souvent appelé le programme de *boot* (amorce en français), peut en revanche contenir l'adresse physique absolue de l'endroit où est stocké le fichier `vmunix` : comme le programme de *boot* est aussi stocké sur disque dur, il est possible de le modifier chaque fois que le fichier `vmunix` est modifié (que ce soit sa place sur le disque dur ou son contenu).

Dans la mesure où le système d'exploitation n'est pas encore exécuté, les programmes de la PROM et de *boot* n'utilisent pas les facilités habituellement proposées par les systèmes d'exploitation, comme la mémoire virtuelle, et ils doivent donc assurer la gestion de la mémoire. En particulier, il est nécessaire de prendre de grandes précautions lorsque le programme de *boot* charge en mémoire le système d'exploitation : si jamais il ne le trouve pas ou si jamais ça se passe mal, la machine reste dans un état où rien ne fonctionne !

Pour éviter ce genre de problèmes, le programme de *boot* et le système d'exploitation peuvent se trouver sur une disquette ou sur un CDROM et ils sont d'ailleurs généralement d'abord cherchés sur ces supports avant d'être cherchés sur le disque dur. Cela permet par exemple d'utiliser un système d'exploitation donné même si le disque dur tombe en panne ou s'il est nécessaire de le reformater.

Malheureusement, cette méthode offre souvent la possibilité de contourner les protections des ordinateurs en plaçant tout simplement un système d'exploitation très

8. Soit littéralement la « languette de botte », sorte de petite pièce de tissu ou de cuir placée en haut à l'arrière des bottes et permettant de les enfiler.

9. Le système d'exploitation est un programme comme un autre et il existe donc un exécutable correspondant au système d'exploitation. C'est cependant un exécutable particulier.

permissif sur une disquette. Ce système sera alors lu au démarrage de la machine et rien n'empêchera alors d'accéder à toutes les données placées sur le disque dur.

Pour éviter ce type de piraterie, il est fréquent que l'ordinateur propose une protection matérielle via un mot de passe : ce dernier est directement stocké dans la PROM et ne peut donc être modifié aussi facilement. Notons cependant que si un pirate peut accéder à une machine pour lui faire lire une disquette, rien ne l'empêche d'ouvrir la machine pour tout simplement enlever le disque dur et le lire ensuite tranquillement chez lui...

Certains systèmes ont des programmes de *boot* qui permettent de choisir un système d'exploitation parmi plusieurs. Par exemple, il est possible d'installer Linux sur une machine ainsi que d'autres systèmes d'exploitation comme DOS ou Windows NT. Le programme de *boot* de Linux charge par défaut Linux, mais donne la possibilité à l'utilisateur de lancer l'exécution des autres systèmes d'exploitation. Il est donc tout à fait possible d'utiliser une machine avec plusieurs systèmes d'exploitation sélectionnés de façon dynamique au moment du démarrage de la machine.

5.3 Structures utilisées pour la gestion des processus

Un processus est obtenu à partir de l'exécution d'un fichier exécutable. Les informations présentes dans ce fichier sont alors utilisées pour effectuer les différentes étapes nécessaires à l'exécution : l'allocation de la mémoire pour stocker le code et les données, l'allocation et l'initialisation de données utilisées par le système d'exploitation pour gérer les processus.

Lorsqu'un programme est exécuté, le processus correspondant a l'illusion de disposer de toute la mémoire pour lui et la correspondance entre les adresses mémoire qu'il utilise et les adresses physiques est assurée par la MMU. Nous discuterons de ce point dans le chapitre suivant.

Ce principe permet en fait d'attribuer des zones mémoire à chaque processus et on dit parfois, par abus de langage, que chaque processus travaille dans son propre espace mémoire. Une conséquence de ce principe est qu'il est assez facile d'autoriser ou d'interdire l'accès au programme ou aux données d'un processus.

Généralement, le programme d'un processus peut être partagé avec d'autres processus, mais les données ne sont en revanche accessibles que par le processus propriétaire. Les processus doivent alors utiliser des communications particulières, gérées par le système d'exploitation, pour échanger des données.

Notons cependant que récemment est apparu un type de processus très particulier, appelé *thread*, qui permet justement le partage du même espace de données entre plusieurs processus de ce type. Les *threads* vont jouer un rôle capital dans le développement des futurs systèmes d'exploitation, mais nous n'en parlerons pas ici, le chapitre 8 leur sera consacré.

Structure de l'espace mémoire d'un processus

L'espace mémoire utilisé par un processus est divisé en plusieurs parties représentées sur la figure 5.6. On trouve en particulier le segment de code (souvent appelé segment de *text*), le segment de données, la pile (*stack* en anglais) et le tas (*heap* en anglais).

Le segment de code

Le segment de code est obtenu en copiant directement en mémoire le segment de code du fichier exécutable. Au cours de l'exécution du programme, la prochaine instruction à exécuter est repérée par un pointeur d'instruction. Si l'exécution d'une instruction du code modifie l'agencement de l'espace mémoire du processus et, par exemple, provoque un déplacement du segment de code, le pointeur d'instruction ne sera plus valable et le programme se déroulera de façon incorrecte.

Afin d'éviter ce problème, le segment de code est toujours placé dans des zones fixes de la mémoire (virtuelle), c'est-à-dire au début de la zone disponible. Comme le système utilise la mémoire virtuelle, cette zone débute généralement à l'adresse 0 et les adresses obtenues seront ensuite traduites en adresses physiques.

Notons à ce sujet que, comme les adresses du segment de texte de l'exécutable débutent aussi à 0, placer le segment de texte à l'adresse 0 de la mémoire du processus évite d'avoir à modifier toutes les adresses du code.

Afin d'éviter les problèmes de réentrance, le segment de code n'est généralement accessible qu'en lecture. Il est fréquent qu'il soit partagé par tous les processus exécutant le même programme.

Le segment de données

Au dessus du segment de code (voir figure 5.6) se trouve le segment de données. Ce segment est traditionnellement composé d'un segment de données initialisées (*data*), qui est directement copié à partir de l'exécutable, et d'un segment de données non initialisées (*bss* pour *block storage segment*) qui est créé dynamiquement.

Les données initialisées correspondent à toutes les variables globales et statiques initialisées des programmes C. Les données non initialisées correspondent aux variables globales et statiques non initialisées.

Le segment de données est agrandi ou réduit au cours de l'exécution pour permettre d'y placer des données. Néanmoins, il est habituel de considérer que ce segment est fixe et correspond à celui obtenu avant l'exécution de la première instruction : le segment *bss* se résume alors aux variables locales de la fonction `main()`.

La pile (*stack*)

Pour stocker les données obtenues en cours d'exécution, le système utilise alors un segment nommé la pile (*stack* en anglais). L'avantage de cette pile est qu'elle

peut justement être gérée comme une pile, c'est-à-dire en empilant et en dépilant les données.

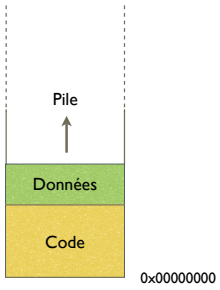


FIGURE 5.3 – *La pile permet de stocker les données des processus.*

Prenons un exemple : la pile (que nous représenterons de gauche à droite dans ce texte) contient les données `ABCD` et l'instruction en cours fait appel à une fonction. Le système peut alors placer sur la pile les paramètres à passer à la fonction (pile `ABCDQR`), puis placer sur la pile les différentes variables locales de la fonction (pile `ABCDQRXYZ`). La fonction s'exécute normalement et le système n'a plus alors qu'à dépiler toutes les données qu'il a placées pour l'appel de la fonction (pile `ABCD`). Le système peut alors accéder directement aux données qu'il avait stockées avant l'appel de la fonction et peut exécuter l'instruction suivante.

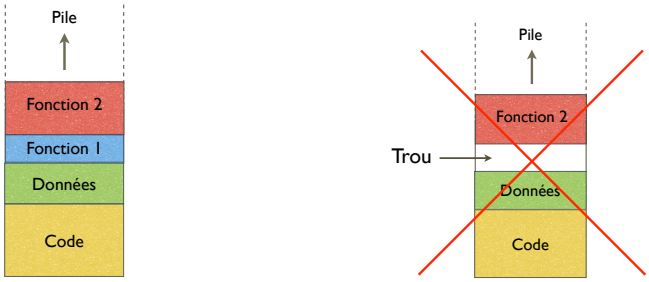


FIGURE 5.4 – *Les variables locales, les paramètres passés à des fonctions, les retour d'appels de fonctions, etc. sont stockés en mémoire par empilement et sont supprimés par dépilement. C'est de cette façon que les variables locales à une fonction sont détruite dès la fin de la fonction. Il n'est donc pas possible de créer de trou au sein de la pile, par définition.*

L'avantage de cette pile est donc évident : les données sont toujours stockées de

façon contiguë et l'espace mémoire du processus forme un ensemble compact qu'il sera facile de traduire en adresses physiques.

Le tas (*heap*)

Malheureusement, ceci n'est plus vrai en cas d'allocation dynamique : l'espace mémoire alloué par une fonction comme **`malloc()`** (par exemple) ne pourra pas être dépilé et les données ne seront plus stockées de façon contiguë.

Reprenons notre exemple de la section précédente et supposons que la fonction appelée alloue 4 octets de mémoire (les octets libres sont représentés par un point et les octets alloués dynamiquement par +) : avant l'appel de la fonction la pile est `ABCD`, juste avant la fin de la fonction la pile est `ABCDQRXYZ+++` et après l'appel de la fonction la pile est `ABCD.....+++`.

Pour éviter de conserver en mémoire l'emplacement des zones libres mais non contiguës, les systèmes d'exploitation utilisent généralement un autre segment pour les allocations dynamiques : le tas (*heap*).

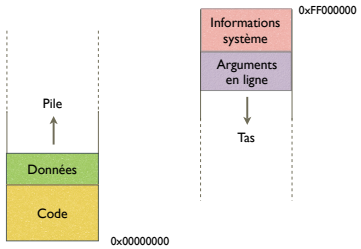


FIGURE 5.5 – *Le tas permet de stocker des données de façon non contiguë et est situé à l'autre extrémité de l'espace mémoire de chaque processus.*

Toutefois, le problème n'est résolu que si l'emplacement du tas ne varie pas en fonction des données stockées dans la pile. C'est pour cette raison que le tas est généralement placé à la fin de l'espace mémoire du processus et qu'il est agrandi du haut vers le bas, contrairement à la pile.

Notons que, comme le système d'exploitation utilise la mémoire virtuelle, l'adresse virtuelle du début du tas importe peu car les zones de mémoire libre entre le tas et la pile ne seront pas traduites en zones physiques.

En pratique, les tailles de la pile et du tas sont bornées et ces bornes font partie des limitations (modifiables) imposées par le système d'exploitation. Voici, par exemple, le résultat de l'exécution de la commande `limit` sous Unix qui indique entre autres la taille maximale de la pile.

```
linux>  limit
cputime         unlimited
filesize        unlimited
```

```
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    1000000 kbytes
memoryuse       unlimited
descriptors     256
memorylocked    unlimited
maxproc         256
```

Les autres informations

D'autres informations sont placées dans l'espace mémoire du processus, comme les paramètres passés sur la ligne de commande lors de l'exécution du programme. Suivant les systèmes d'exploitation, le système peut aussi stocker à cet endroit différents renseignements, comme par exemple le mode dans lequel doit s'exécuter le processus. Ces renseignements se trouvent dans une structure qui se nomme souvent *Process Control Bloc* (PCB) ou *Task Control Bloc* (TCB).

Toutes ces informations sont utilisées par le système et doivent être faciles à retrouver : il ne faut pas qu'à chaque changement de la pile ou du tas, le système ait à remettre à jour ses structure internes. Pour cette raison, ces informations sont généralement stockées à la fin de l'espace mémoire du processus, après le tas. Comme leur taille est fixe, cela ne gêne en rien le fonctionnement du tas.

Conclusion

L'espace mémoire d'un processus est donc divisé en deux morceaux variables situés chacun à une extrémité de l'espace. Ces morceaux sont eux-même divisés en plusieurs segments, certains de taille fixe situés aux extrémités, d'autres de taille variable.

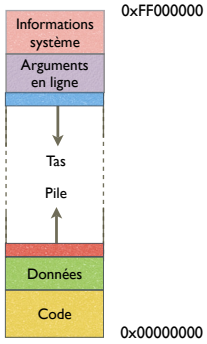


FIGURE 5.6 – *Organisation de l'espace mémoire d'un processus.*

Suivant les systèmes d'exploitation, le tas et la pile peuvent se trouver l'un en haut et l'autre en bas, ou l'inverse.

État des processus

Dans les systèmes à temps partagé, les processus se trouvent dans différents états suivant qu'ils ont la main ou qu'ils attendent d'avoir la main. Le nombre de ces états varie d'un système à l'autre, mais nous pouvons en distinguer au moins six :

Nouveau : le processus est en cours de création ;

En exécution : le processus dispose du processeur et les instructions de son programme sont en train d'être exécutées ;

Prêt : le processus est prêt et attend que le système d'exploitation lui attribue le processeur ;

En attente : le processus a besoin d'un événement particulier pour pouvoir continuer, par exemple l'achèvement d'une entrée / sortie ;

Stoppé : le processus n'a besoin de rien pour continuer et il pourrait demander l'accès au processeur, mais il ne le fait pas ;

Terminé : toutes les instructions du programme du processus ont été exécutées et le processus attend que le système d'exploitation libère les ressources correspondantes.

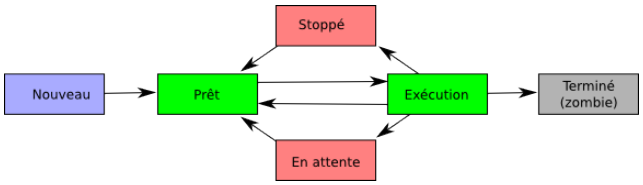


FIGURE 5.7 – *Graphe des états des processus.*

Pour sélectionner le prochain processus, le système d'exploitation parcours la liste des processus à l'état « prêt » et choisit celui qui doit effectuer le travail le plus prioritaire. Le processus choisi passe alors à l'état « en exécution ». Si ce processus a besoin d'une entrée / sortie, il passe alors à l'état « en attente » et le système d'exploitation choisit un autre processus. Une fois que le quantum attribué à un processus est écoulé, le système d'exploitation le remet dans l'état « prêt » et refait une sélection.

Sous Unix, les processus peuvent prendre cinq états :

SIDL : au moment de la création (pour *idle*) ;

SRUN : prêt à être exécuté ;

SSLEEP : en attente d'un événement ;

SSTOP : arrêté par le signal SIGSTOP, parfois sur la demande de l'utilisateur ;

SZOMB : en attente de terminaison.

Notons qu'il n'y a pas d'état pour spécifier si le processus est en cours d'exécution : au moment où ces états sont pris en compte, seul le noyau du système d'exploitation est en train de s'exécuter...

La table des processus

La plupart des systèmes d'exploitation gardent en mémoire une table contenant toutes les informations indispensables au système d'exploitation pour assurer une gestion cohérente des processus. Cette table s'appelle la table des processus et elle est stockée dans l'espace mémoire du noyau du système d'exploitation, ce qui signifie que les processus ne peuvent pas y accéder. Cette table se décompose en blocs élémentaires correspondants aux différents processus.

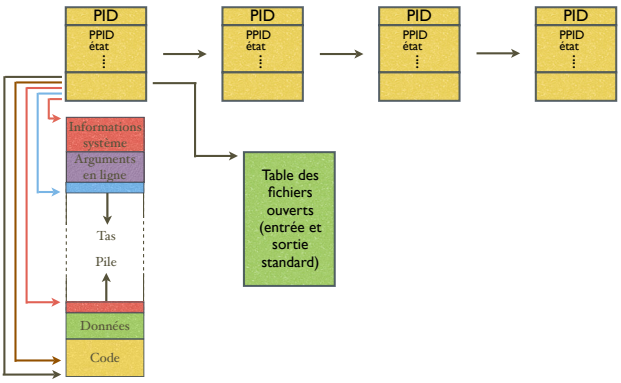


FIGURE 5.8 – *La table des processus.*

Les informations présentes dans cette table ne doivent pas être confondues avec les informations stockées dans l'espace mémoire du processus.

Le nombre et le type des informations contenues dans chaque bloc varient d'un système d'exploitation à l'autre. Nous pouvons néanmoins en distinguer au moins 7 :

L'état du processus : comme nous venons de le voir, cet état est indispensable pour effectuer un ordonnancement efficace ;

Le compteur d'instructions : cela permet au système d'exploitation de connaître la prochaine instruction à exécuter pour chaque processus ;

Les registres du processeur : ces registres sont utilisés par la plupart des instructions et il est donc nécessaire de sauvegarder leur contenu pour pouvoir reprendre l'exécution du programme du processus que l'on souhaite interrompre ;

Des informations pour l'ordonnancement des processus : on y trouve, par exemple, la priorité du processus ou le temps passé dans l'état « en attente » ;

Des informations pour la gestion de la mémoire : comme par exemple, l'adresse des pages utilisées ;

Les statistiques : comme le temps total passé en exécution ou le nombre de changements de contexte.

Outre ces informations, il est fréquent que des systèmes évolués, comme Unix, précisent en plus le numéro d'identification du processus, le numéro d'identification du propriétaire du processus et la liste des fichiers ouverts par le processus.

Néanmoins, il ne faut pas non plus que la table des processus prenne trop de mémoire et, donc, il ne faut pas que chaque bloc contienne trop de renseignements. En particulier, et pour des raisons évidentes, il est toujours difficile de stocker ce type de table en mémoire secondaire et les systèmes d'exploitation ont tendance à les garder toujours en mémoire principale.

Pour limiter la taille de chaque bloc, de nombreux systèmes reportent toutes les informations qui ne sont pas absolument indispensables à la gestion des processus dans l'espace mémoire de ceux-ci, dans le bloc de contrôle. Ces informations pourront en particulier se retrouver (éventuellement) en mémoire secondaire et le système d'exploitation devra recharger les pages correspondantes en mémoire principale s'il souhaite y accéder.

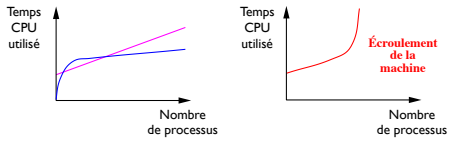


FIGURE 5.9 – *La tenue en charge des systèmes d'exploitation dépend très fortement de leur capacité à gérer les processus, notamment lorsque le nombre de processus augmente.*

Nous reparlerons de l'affectation des priorités et des informations destinées à l'ordonnancement des processus dans la section suivante qui lui est consacrée. Les informations pour la gestion de la mémoire sont détaillées dans le chapitre suivant.

5.4 L'ordonnancement des processus (*scheduling*)

L'ordonnancement des processus est l'ordre dans lequel le système d'exploitation attribue, pour un quantum de temps, le processeur aux processus. Nous avons vu en début de chapitre que cet ordonnancement était grandement facilité par l'utilisation

d'une horloge qui déclenche régulièrement une interruption : à chaque interruption, le système d'exploitation choisit le prochain processus qui bénéficiera du processeur.

Dans ce chapitre, nous revenons sur l'opération de changement de contexte, puis nous détaillons les différentes politiques d'ordonnancement utilisées par les systèmes d'exploitation.

Le changement de contexte

Le changement de contexte est l'opération qui, d'une part, sauvegarde le contexte du processus à qui on retire le processeur et qui, d'autre part, rétablit le contexte du processus qui vient d'être choisi pour bénéficier du processeur.

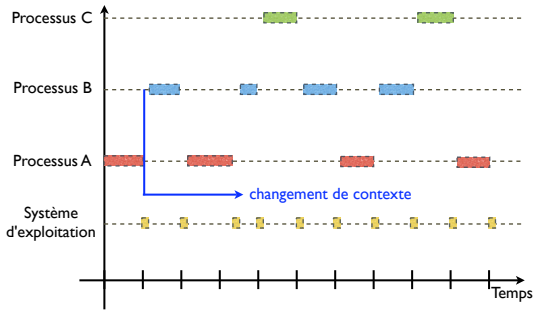


FIGURE 5.10 – *Changement de contexte.*

Le changement de contexte assure à un processus qu'il retrouvera toutes les informations dont il a besoin pour reprendre l'exécution de son programme là où il a été interrompu.

Les changements de contexte arrivent très souvent : après chaque quantum de temps, chaque fois que le noyau du système d'exploitation prend la main et chaque fois qu'un processus est dessaisi du processeur avant la fin du quantum qui lui était attribué (parce qu'il attend une entrée / sortie, par exemple). L'opération de changement de contexte doit donc être très rapide et, en pratique, elle est souvent assurée en grande partie par les couches matérielles de la machine. Sur les processeurs Intel, par exemple, il existe un changement de contexte matériel depuis le 80386. Il est ainsi possible de charger le nouveau contexte depuis le TSS (*Task State Segment*). Cependant cette facilité n'est utilisée ni par Windows ni par Unix car certains registres ne sont pas sauvegardés.

Les machines les plus performantes effectuent un changement de contexte en quelques microsecondes et cette opération influe donc peu sur les quanta attribués aux processus (qui, eux, sont de l'ordre de la dizaine de millisecondes).

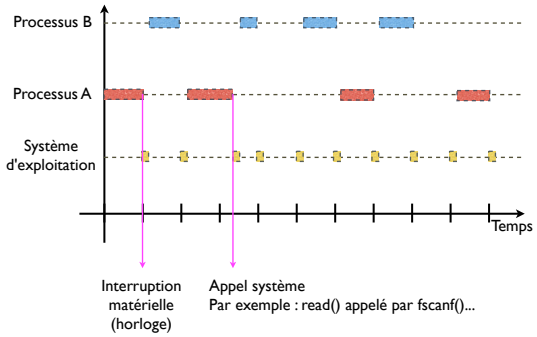


FIGURE 5.11 – *Le système d'exploitation intervient à chaque changement de contexte, c'est-à-dire à la fin de chaque quantum de temps et lors de chaque appel système.*

Les politiques d'ordonnancement

La politique d'ordonnancement qu'un système d'exploitation applique reflète l'utilisation qui est faite de la machine que ce système gère. Par exemple, le système d'exploitation d'une machine destinée à faire de longs calculs sans aucune interaction avec les utilisateurs évitera d'utiliser de petits quanta de temps : il peut considérer que le temps de réaction de la machine n'est pas important et laisser 1 seconde à chaque processus pour exécuter tranquillement son programme.

Inversement, le système d'exploitation d'une machine utilisée pour des tâches interactives, comme l'édition de texte, devra interrompre très souvent les processus pour éviter qu'un utilisateur n'attende trop avant de voir ses actions modifier l'affichage à l'écran.

Si nous nous reportons à l'historique des systèmes d'exploitation (chapitre 3), nous pouvons distinguer 5 critères pour effectuer un bon ordonnancement :

Efficacité : même si cette contrainte est moins présente aujourd'hui, le but ultime est de ne jamais laisser le processeur inoccupé ;

Rendement : l'ordinateur doit exécuter le maximum de programmes en un temps donné ;

Temps d'exécution : chaque programme doit s'exécuter le plus vite possible ;

Interactivité : les programmes interactifs doivent avoir un faible temps de réponse ;

Equité : chaque programme a droit à autant de quanta que les autres.

Nous ne reviendrons pas sur le fait que certains de ces critères sont contradictoires et nous dirons donc qu'un bon algorithme d'ordonnancement essayera de remplir au mieux ces critères.

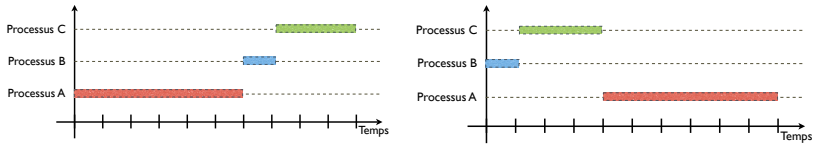


FIGURE 5.12 – *Deux politiques d'ordonnancement pratiquées jadis : premier arrivé, premier servi et le plus court d'abord.*

Ordonnancement par tourniquet (*round robin*)

L'ordonnancement par tourniquet est celui que nous avons utilisé comme exemple tout au long de ce document. C'est probablement l'algorithme le plus simple et le plus efficace : chaque processus se voit attribuer un laps de temps (quantum) pendant lequel il bénéficie du processeur et, donc, pendant lequel il peut exécuter les instructions de son programme.

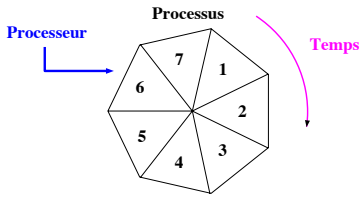


FIGURE 5.13 – *Principe de l'ordonnancement des processus par tourniquet.*

À la fin du quantum de temps, un autre processus dans l'état « prêt » est sélectionné. Si le processus s'arrête avant la fin de son quantum, le processeur est immédiatement attribué à un autre processus, sans attendre la fin du quantum.

La gestion de l'ordonnancement par tourniquet est assez simple : il suffit de maintenir une liste chaînée des différentes processus dans l'état « prêt ». Si un processus vient de bénéficier du processeur, on le place en fin de liste et on sélectionne le processus suivant dans la liste (voir figure 5.14).

Ordonnancement par priorités

L'inconvénient de l'ordonnancement par tourniquet est que tous les processus dans l'état « prêt » sont considérés de la même façon et il n'y a donc pas de notion de priorité d'exécution. Il est pourtant clair qu'il peut y avoir des processus plus importants que d'autres et que l'utilisation de priorités peut être intéressante.

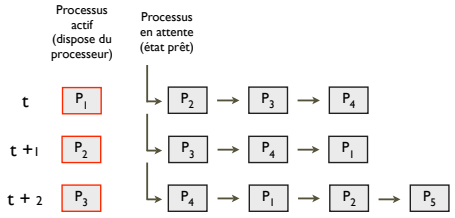


FIGURE 5.14 – *Principe de l'ordonnancement des processus par tourniquet, en pratique.*

Pour effectuer un ordonnancement par priorités, chaque processus se voit attribuer une priorité. Le système d'exploitation maintient alors autant de listes chaînées de processus qu'il y a de priorités différentes et, à chaque changement de processus, il parcourt l'ensemble des listes de la plus prioritaire à la moins prioritaire en appliquant à chaque liste un ordonnancement par tourniquet (voir figure 5.15).

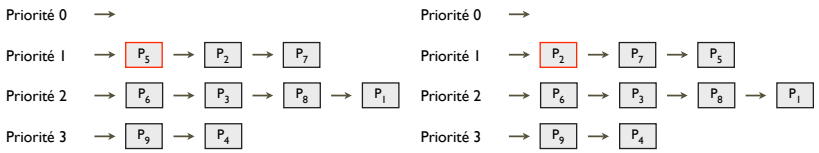


FIGURE 5.15 – *Ordonnancement des processus par priorités.*

La priorité ainsi affectée à chaque processus ne peut pas être statique, car sinon un processus de très haute priorité pourrait tourner seul sur la machine sans jamais rendre la main : il suffit qu'il soit le seul de cette priorité. Par exemple, la figure 5.17 montre l'ordonnancement effectué dans le cas où un processus `A` a une priorité 5 et un processus `B` une priorité 3 : tant que le processus `A` existe, `A` a la priorité la plus élevée et il est donc le seul à bénéficier du processeur.

Il est donc nécessaire de mettre en place une politique de priorité dynamique, c'est-à-dire de déterminer l'algorithme qui permet de calculer à chaque quantum de temps la nouvelle priorité de chaque processus. Outre le maintien de listes chaînées associées à chaque priorité, le système d'exploitation doit alors, à chaque changement de processus, recalculer toutes les priorités et réordonner les différentes listes avant de les parcourir.

Supposons que la priorité de chaque processus est décrémentée pour chaque unité de temps pendant laquelle le processeur lui a été attribué et incrémentée pour chaque unité de temps pendant laquelle le processeur a été attribué à un autre processus. Cette

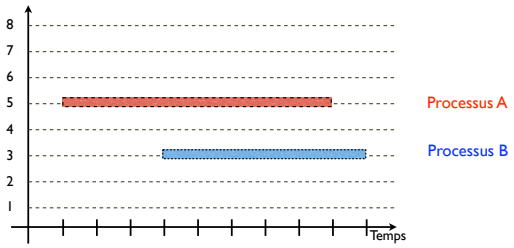


FIGURE 5.16 – *Utilisation de priorités statiques.*

politique de priorité permet bien d'attribuer le processeur aux processus qui sont en attente depuis un certain temps, mais elle pose néanmoins deux problèmes : les priorités absolues et les attentes d'entrées / sorties.

Tout d'abord, il devient difficile d'affecter une priorité absolue à un processus. Supposons en effet que nous ayons un processus `A` de priorité 8 seul sur une machine. La priorité de ce processus va baisser à chaque unité de temps et, ainsi, `A` aura une priorité de 1 au bout de 7 unités de temps. Si nous lançons alors un processus `B` de priorité 6, `B` bénéficiera du processeur pendant 3 unités de temps avant que `A` ne puisse prétendre au processeur. Le dernier processus créé bénéficie ainsi de plus d'unités de temps que les processus plus anciens, ce qui n'a a priori aucun rapport avec les priorités réelles de ces deux processus.

Par ailleurs, après les 3 premières unités de temps, les priorités des deux processus `A` et `B` vont se stabiliser autour d'une position d'équilibre (en l'occurrence la priorité 5) et l'ordonnancement de ces deux processus s'effectuera désormais par tourniquet (voir figure 5.17). La notion de priorité perd son sens et la politique de gestion par priorités décrite ci-dessus n'est donc pas adéquate.

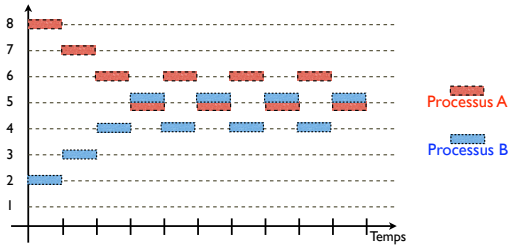


FIGURE 5.17 – *Utilisation de priorités dynamiques conduisant à un ordonnancement par tourniquet.*

Une politique d'ordonnancement cohérente consisterait alors, par exemple, à affecter deux fois plus de quanta à `A` qu'à `B`. Pour réaliser ce type de gestion, les processus ont souvent, en plus de leur priorité dynamique courante, une priorité statique qui définit les quantités utilisées pour incrémenter et décrémenter la priorité dynamique du processus et qui donne une borne maximale et minimale de la priorité dynamique.

Dans notre exemple, on peut ainsi imaginer que la priorité de `A` ne peut varier qu'entre 7 et 8 et celle de `B` entre 1 et 8. Ceci permet de corriger l'effet de « centrage » autour d'une priorité moyenne, mais ne permet pas d'éviter une stabilisation selon un ordonnancement par tourniquet (voir figure 5.18).

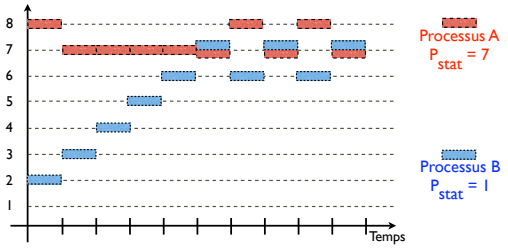


FIGURE 5.18 – *Borner les priorités dynamiques permet d'éviter une stabilisation autour de la priorité moyenne, mais conduit tout de même rapidement à un ordonnancement par tourniquet.*

En revanche, si on incrémente en outre la priorité de `A` de 5 et si on décrémente sa priorité de 1 à chaque fois, tout en incrémentant la priorité de `B` de 1 et en la décrémentant de 4, `A` ne devra attendre que 4 unités de temps avant de reprendre la main (voir figure 5.19).

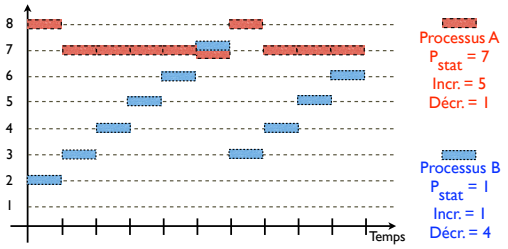


FIGURE 5.19 – *Borner les priorités dynamiques tout en utilisant des incréments et des décréments différents permet d'avantager un processus par rapport à un autre.*

L'attente des entrées / sorties pose aussi un problème. Supposons que nous disposons d'un processus qui effectue beaucoup d'entrées / sorties. Ces entrées / sorties durent généralement assez longtemps et le processus ne pourra jamais profiter de ses quanta de temps : comme il passe de l'état « prêt » à l'état « en attente », il est immédiatement dessaisi du processeur (c'est le principe même de la multi-programmation). Il paraît donc normal que ce processus bénéficie de ses quanta de temps lorsqu'il a terminé son entrée / sortie : il peut ainsi exécuter rapidement quelques instructions et passer à l'entrée / sortie suivante.

Il est alors fréquent d'attribuer des priorités élevées aux processus venant d'effectuer des entrées / sorties, c'est-à-dire aux processus repassant dans l'état « prêt ». La priorité ainsi calculée dépend généralement du temps passé à attendre la fin des entrées / sorties, ce qui représente une bonne estimation du nombre de quanta perdus.

Ordonnancement par durées variables

Les deux politiques d'ordonnancement définies ci-dessus supposent de nombreux changements de contexte et nous avons vu que cela pouvait être nuisible pour les programmes non interactifs effectuant beaucoup de calculs.

Pour éviter d'interrompre sans arrêt ces programmes, une solution consiste à attribuer un nombre différent de quanta à chaque processus : les processus interactifs auront droit à un seul quantum et les processus non interactifs auront droit à plusieurs quanta. Afin de ne pas pénaliser les processus interactifs, le nombre de quanta attribués à chaque processus est inversement proportionnel à sa priorité dynamique et cette priorité diminue à chaque quantum écoulé.

Ainsi, supposons que nous disposons d'un processus qui a besoin de 50 quanta de temps pour terminer l'exécution de son programme. Ce processus se verra attribuer la priorité maximale 100 et un seul quantum. Puis, une fois ce quantum utilisé, il se verra attribuer la priorité 99 et deux quanta. On continue ainsi de suite jusqu'à la fin de son exécution. Ce processus aura donc droit a 63 quanta ($1+2+4+8+16+32$) et aura nécessité 7 changements de contexte (au lieu de 50 pour le tourniquet).

Les changements de contexte prennent désormais assez peu de temps (ce ne fut pas toujours le cas) et cette politique présente un gros désavantage : si un processus très peu prioritaire prend la main pour 50 quanta et que, au bout d'un quantum, un processus prioritaire ou interactif est lancé, il devra attendre 49 quanta avant d'avoir la main.

Néanmoins, cette méthode avait l'avantage de favoriser les programmes interactifs et de ne pas trop interrompre les programmes de calculs longs.

Un exemple concret : 4.4BSD

Afin d'illustrer concrètememt les principes d'ordonnancement énoncés dans les sections précédentes, nous allons maintenant détailler la politique d'ordonnancement appliquée par un système Unix particulier : le système 4.4BSD. Comme la grande

majorité des systèmes Unix, 4.4BSD cherche à favoriser l'interactivité tout en tentant de ne pas pénaliser les programmes qui ont besoin de beaucoup de temps CPU. La prise en charge des changements de contexte par les couches matérielles de la machine et, donc, la grande rapidité de ceux-ci a grandement facilité l'application de cette politique.

Le système 4.4BSD utilise un ordonnancement par priorités, tel qu'il a été décrit précédemment, et il fonctionne donc en affectant une priorité dynamique à chaque processus, en maintenant autant de listes de processus qu'il n'y a de priorités et en effectuant un ordonnancement par tourniquet au sein de chaque liste de processus.

Néanmoins, le système 4.4BSD (comme la grande majorité des systèmes Unix) utilise non pas deux, mais trois priorités différentes afin de tenir compte de l'action (ou de l'état) des processus dans l'ordonnancement et, par exemple, affecter une priorité dynamique élevée à un processus qui vient d'effectuer une entrée / sortie. Ainsi, pour calculer la priorité `p_dyn` affectée à chaque processus, le système 4.4 BSD utilise les paramètres suivants :

- Une priorité statique `p_nice` qui permet de déterminer la priorité réelle (en dehors de toute notion d'ordonnancement) du processus concerné ;
- Une priorité d'événement `p_event` qui permet notamment de favoriser les processus qui n'ont pas pu bénéficier de l'intégralité de leurs quanta de temps (voir les raisons plus loin) ;
- La durée `p_estcpu` pendant laquelle le processus a récemment bénéficié du processeur.

En pratique, les priorités sont indiquées dans l'ordre inverse de celui que nous avons employé jusqu'ici : plus basse est la priorité, plus prioritaire est le processus. Ainsi, la priorité statique `p_nice` varie entre -20 et 20 et la priorité dynamique `p_dyn` entre 0 et 127. La priorité dynamique d'un processus est alors calculée par la formule :

$$\texttt{p\_dyn} = \min(127, \max(\texttt{p\_event}, \texttt{p\_event} + \frac{\texttt{p\_estcpu}}{4} + 2 * \texttt{p\_nice}))$$

Si nous faisons abstraction du paramètre `p_event`, nous pouvons constater que la formule ci-dessus permet effectivement d'atteindre les objectifs fixés :

- Les processus qui ont bénéficié longuement du processeur ont un paramètre `p_estcpu` élevé et ils deviennent ainsi moins prioritaires. Ceci permet de répartir de façon équitable le temps CPU.
- Les processus qui possèdent une priorité statique `p_nice` faible (voir négative) bénéficie du processeur plus souvent, c'est-à-dire jusqu'à ce que le terme `p_estcpu` annule le terme `p_nice`.

Notons que chaque utilisateur peut modifier la valeur par défaut de la priorité statique attribuée à ses processus. Néanmoins, afin d'éviter que tous les utilisateurs ne tentent de rendre leurs processus plus prioritaires que ceux du voisin, il est uniquement possible de rendre ces processus moins prioritaires. Seul le super-utilisateur a le pouvoir de rendre un processus plus prioritaire que les processus de priorité standard.

Le terme `p_event` permet quant à lui, d'une part, de borner la priorité dynamique et, d'autre part, de tenir compte des derniers événements qui ont perturbé l'exécution du processus concerné. Voici un extrait des valeurs usuellement attribuées à `p_event` en fonction des différents événements qui peuvent survenir :

PSWP	0	Lorsqu'un processus a été retiré de la mémoire principale (« *swap* »), faute de place
PVM	4	Lorsqu'un processus est en attente d'une allocation mémoire
PRIBIO	16	Lorsqu'un processus attend la fin d'une entrée / sortie
PZERO	22	Priorité de base des processus exécutés en mode noyau
PWAIT	32	Lorsqu'un processus attend la terminaison d'un de ses fils (*wait*)
PPAUSE	40	Lorsqu'un processus attend l'arrivée d'un signal
PUSER	50	Priorité de base des processus exécutés en mode utilisateur

Ainsi, un processus qui doit effectuer une entrée / sortie ne pourra pas consommer jusqu'au bout le quantum de temps qui lui avait été attribué (principe de la multiprogrammation), mais, en contrepartie, il bénéficiera d'une priorité dynamique élevée dès qu'il aura terminé cette entrée / sortie. En l'occurrence, sa priorité sera PRIBIO au lieu de PUSER pour un processus ayant consommé normalement ses quanta de temps.

La politique d'ordonnancement sous Windows

La création de processus sous Windowsest plus complexe et plus coûteuse que sous Unix. La raison évoquée par les développeurs du noyau actuel (Windows XP, Windows Vista et Windows Seven) est la suivante : « Cette création n'a pas à être peu onéreuse car elle intervient très rarement, généralement lorsqu'un utilisateur clique sur une icône de programme » ! Le choix, plus intéressant, qui est fait est de privilégier la création des *threads* et non des processus. Couplé à un ordonnancement en mode utilisateur des *threads*, ce choix permet d'accélérer les choses.

Dans les versions précédant Windows Seven (notamment Windows XP) l'ordonnanceur avait parfois des comportements peu équitables. Le compteur de cycles n'étant pas pris en compte, il arrivait que certains *threads* soient défavorisés par rapport à d'autres. Ainsi, lorsque deux *threads* `TA` et `TB` de même priorité sont en mode « prêt », le noyau en choisit un, `TA` par exemple. Supposons qu'un *thread* `TC` de priorité plus élevée arrive dans l'état « prêt ». Le noyau décide alors d'interrompre `TA` pour laisser la place à `TC` sans garder trace du nombre de cycles réellement obtenus par `TA`. À la fin de `TC`, `TB` obtiendra le CPU car le noyau aura gardé en mémoire le fait que `TA` a obtenu l'intégralité de son quantum de temps. Et si `TB` n'est pas interrompu, il aura eu, au final, le CPU plus longtemps que `TA` tout en ayant une priorité égale. Cette politique a été modifiée dans les versions ultérieures (Windows Seven). C'est pourquoi, malgré une création de thread efficace, l'ordonnancement peu équitable pouvait parfois donner l'impression que le système était figé en ne laissant que peu de place à certains processus interactifs.

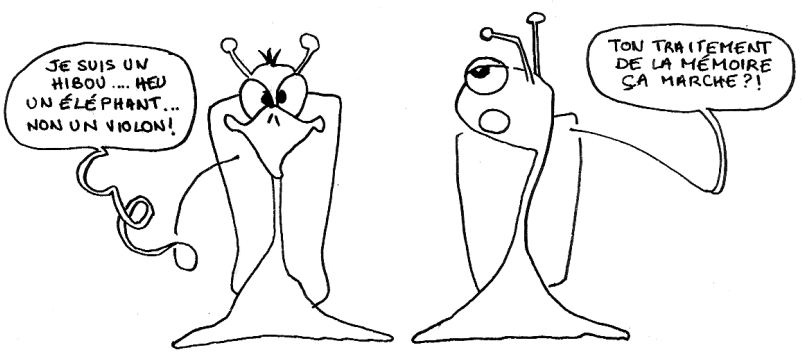
Un nouveau module de l'ordonnanceur a fait son apparition à partir de Windows Vista. Il s'agit du MMCSS (*MultiMedia Class Scheduling Service*). Ce service permet de donner des priorités plus élevées aux applications de type multimédia, gourmandes en CPU, afin d'éviter que le démarrage d'un programme anti-virus ne saccade le décodage et l'affichage (rappelons que la partie graphique fait partie de l'OS) d'un contenu multimédia. Nous reviendrons sur ce module dans le chapitre 8 traitant des *threads*.

5.5 Conclusion

En conclusion, il est important de retenir que, d'une part, la politique d'ordonnancement des processus dépend de l'utilisation qui va être faite de la machine et que, d'autre part, la politique d'ordonnancement mise en place sur les systèmes Unix est un bon compromis pour une utilisation polyvalente d'un ordinateur. Précisons pour les amateurs d'expériences intéressantes qu'il est très difficile de mettre au point en pratique une politique d'ordonnancement efficace et que la formule présentée ci-dessus est le résultat de longues années d'études.

6

La gestion de la mémoire



La mémoire principale d'un ordinateur est une ressource à partager entre les différents processus qui s'exécutent. La qualité de cette répartition joue de façon flagrante sur l'efficacité de la concurrence d'exécution entre les processus et la gestion de la mémoire est donc une fonction essentielle de tout système d'exploitation.

Nous avons vu dans le chapitre 1 que la mémoire principale est rapide, mais relativement coûteuse et de taille limitée. Les mémoires secondaires, comme les disques durs, sont en revanche beaucoup moins chères et, surtout, elles offrent des volumes de stockage beaucoup plus importants. Les temps d'accès à ce type de mémoire sont malheureusement beaucoup trop longs et ils restent inadaptés à la vitesse des processeurs.

L'objet de ce chapitre est de montrer comment les systèmes d'exploitation optimisent l'emploi de la mémoire principale en utilisant, d'une part, des adresses virtuelles pour désigner les données et, d'autre part, des supports plus lents en tant que mémoire secondaire comme le disque dur.

Ce chapitre est en cours de développement et reste donc assez succinct.

Mots clés de ce chapitre : MMU, gestionnaire de mémoire, adresses virtuelles, adresses physiques, fragmentation, pagination.

6.1 Les adresses virtuelles et les adresses physiques

Nous avons déjà rapidement abordé ce sujet dans le chapitre 1 en esquissant l'utilisation et la programmation de la MMU. Nous illustrons ici l'utilisation pratique de la mémoire virtuelle par les systèmes d'exploitation.

Il es toutefois bon de rappeler que ce principe de virtualisation de la mémoire est très ancien et principalement lié à la rareté de cette denrée (la RAM) au début de l'informatique. On en retrouve l'origine dans un article de James Kilburn en 1962 qui décrivait un moyen d'augmenter virtuellement la mémoire disponible pour les processus en associant, au travers d'une translation d'adresses, une mémoire centrale (mémoire à tore) et un disque dur. Remarquons que ce principe de translation se retrouve dans nombre de domaine, nous pouvons ainsi citer les images format GIF dans lesquelles on associe à un pixel une valeur, cette valeur étant une entrée dans une table de 256 éléments contenant chacune les trois valeurs RVB d'une couleur. Il en va de même pour la mémoire virtuelle, chaque adresse virtuelle correspondant, au travers d'une table, à une adresse physique. . . ou à un défaut de page, que nous étudierons dans ce chapitre.

Le rôle de la MMU

Lorsqu'un programme est exécuté par un utilisateur, le système d'exploitation doit allouer la quantité de mémoire nécessaire au processus correspondant à ce dernier. Ce processus (que nous nommerons `proc1`) sera alors situé (par exemple) aux adresses physiques *0* à *999*.

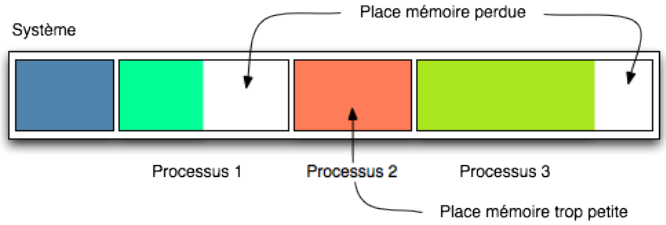


FIGURE 6.1 – *La segmentation a priori de la mémoire ne permet pas de garantir que tout processus aura assez de place, ni que de la place ne sera pas inutilement perdue.*

Si un autre processus, nommé `proc2`, est ensuite créé, il se verra probablement attribuer les adresses physiques *1000* à *1999*. Le processus `proc1` ne pourra alors pas allouer une zone mémoire contiguë à la zone qui lui a été attribuée et, en cas de demande, le système d'exploitation lui attribuera par exemple la zone située entre l'adresse physique *2000* et l'adresse physique *2999* (voir figure 6.2).

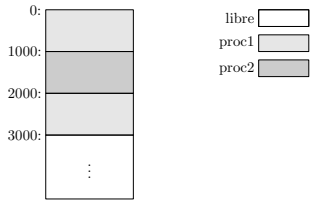


FIGURE 6.2 – *Etat d'allocation de la mémoire.*

Les zones occupées par des processus peuvent ne pas être contiguës et, par exemple, si le processus `proc2` libère les zones mémoires qui lui ont été attribuées, une petite zone de mémoire libre va apparaître. Cette zone ne pourra alors être utilisée que pour des allocations futures ne réclamant que peu de mémoire et toutes les allocations demandant une zone importante devront utiliser les adresses physiques situées au-dessus de l'adresse physique *3000*.

Ce phénomène s'appelle la fragmentation et il est fréquent que, suite à différentes allocations et libérations, la mémoire soit fragmentée en une multitude de petites zones difficilement utilisables.

Lorsque la mémoire est trop fragmentée pour être utilisée correctement, deux méthodes sont utilisées pour proposer des zones mémoire importantes et contiguës : la défragmentation de la mémoire et la simulation d'allocation contiguë.

La défragmentation de la mémoire

Le principe est simple : il suffit de déplacer les zones mémoires allouées vers les adresses basses (voir figure 6.3). Cela revient à tasser la mémoire en éliminant toutes les zones libres.

Cette méthode a néanmoins un inconvénient majeur : il faut prévenir tous les processus du déplacement des zones qui leur sont allouées.

Certains systèmes utilisent néanmoins ce procédé. Par exemple, les Macintosh, avant Mac OS X, effectuent la défragmentation de la mémoire par des *handles*. Un *handle* contient un pointeur sur l'adresse d'un bloc mémoire alloué. Un processus utilisant une zone mémoire doit alors verrouiller cette zone (afin qu'elle ne soit pas déplacée) et utiliser le *handle* pour découvrir l'adresse de la zone mémoire. Après usage, la zone mémoire doit être déverrouillée pour permettre de nouveau au système de déplacer cette zone si nécessaire.

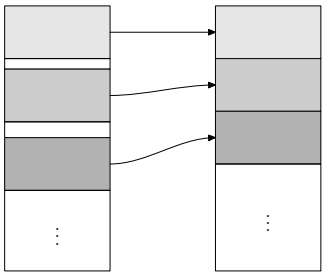


FIGURE 6.3 – *Défragmentation de la mémoire.*

La simulation d'allocations contiguës

Lors d'une demande d'allocation mémoire par un processus, le gestionnaire de mémoire cherche des zones mémoire libres sans se soucier de leurs positions et il laisse ensuite croire au processus que ces zones ne forme qu'une seule et même zone (contiguë !).

Cette illusion est assurée par l'utilisation d'adresses virtuelles. Comme nous l'avons vu dans le chapitre 1, chaque zone mémoire possède deux adresses : une adresse dite physique, qui correspond à la position matérielle de la zone en mémoire et une adresse dite virtuelle qui est un nombre arbitraire auquel il est possible de faire correspondre une adresse physique.

La conversion des adresses virtuelles utilisées par un processus en adresses physiques connues par les couches matérielles a lieu constamment pendant l'exécution d'un processus. Cette conversion doit donc être très rapide (puisqu'elle conditionne la vitesse d'exécution d'un processus) et est assurée par un composant électronique logé dans le processeur : la MMU [1].

Puisque nous allons, au travers de la MMU, utiliser une traduction, il serait fort peu judicieux d'allouer à chaque demande l'exacte quantité souhaitée par le processus. La mémoire est donc paginée et la plus petite quantité que l'on peut demander est en fait une page. Ceci est très lié à l'architecture du microprocesseur et pas vraiment au système d'exploitation. Un Pentium II supportait des pages de 4096 octets. Par contre les processeurs Sparc autorisent des pages de 8192 octets et les derniers UltraSparc supportent de multiples tailles de pages. Ce découpage de la mémoire n'a strictement aucun effet sur les adresses. Supposons qu'une adresse mémoire soit codée sur 16 bits. Cette adresse peut prendre une valeur dans l'intervalle $\{0, 65535\}$. Si la taille de page est de 4096 octets (2^{12}), une adresse est donc simplement un numéro de page sur 4 bits suivi d'un décalage dans la page codé sur 12 bits. La figure 6.4 résume ce principe.

1. La MMU n'a pas toujours été logée directement dans le processeur, mais c'est désormais le cas.

La page étant l'unité atomique de mémoire, un adressage virtuel doit maintenir l'association entre l'adresse virtuelle d'une page et son adresse physique. C'est le rôle de la MMU. Ce tableau associatif est appelé table des pages, il est indexé par les adresses virtuelles.

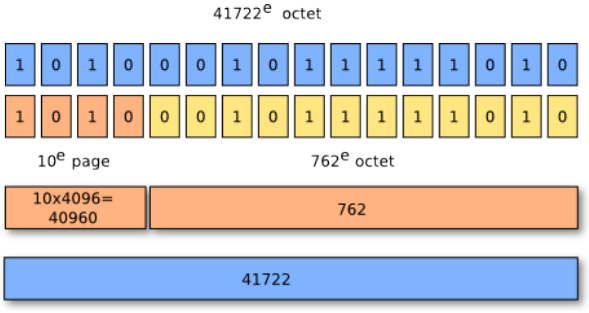


FIGURE 6.4 – *Le découpage de la mémoire en pages n'a aucun effet sur les adresses.*

Sur un système capable d'adresser 4 Go octets de mémoire (32 bits), le nombre de pages de 4 ko dont il faut garder l'adresse est de 1048576 ! Et comme chaque processus possède son système d'adressage virtuel, il faut conserver l'ensemble des associations de pages par processus. Afin de ne garder en mémoire que le strict minimum, nous pouvons reproduire le découpage vu précédemment afin de convenir qu'une adresse est formée par :

- les 10 bit de poids fort permettant d'accéder à 1024 tables de pages différentes au travers d'un catalogue de tables de pages ;
- les 10 bits qui suivent permettent d'accéder aux 1024 pages différentes d'une table de pages ;
- enfin les 12 bits de poids faible permettent de se déplacer dans les 4096 octets d'une page.

Pour un processus et une adresse virtuelle donnée, il faut donc avoir à disposition le catalogue des tables de pages et la table de pages concernée. Les tables de pages inutilisées peuvent très bien ne pas se trouver en mémoire…celles qui sont utilisées doivent être impérativement en mémoire physique !

Le catalogue des tables de pages se trouve dans la mémoire physique, à une adresse dont la valeur est placée dans un registre spécial, le registre `CR3`. La figure 6.5 illustre ces étapes de traduction successives.

Afin d'accélérer la traduction des adresses on utilise un mécanisme de cache des différents catalogues et des différentes tables, le TLB ou « Translation Lookaside Buffer ». Le TLB contient la traduction de certaines adresses virtuelles en adresses physiques. Il est régulièrement tenu à jour, ainsi lorsqu'un catalogue ou une table est modifié, son entrée dans le TLB est invalidée. Dans ce cas on reprend le calcul vu

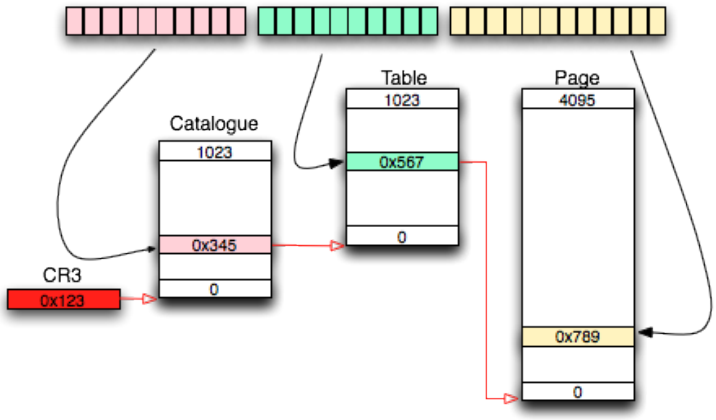


FIGURE 6.5 – *Les étapes de traduction d'une adresse virtuelle en adresse physique. Le registre `CR3` donne l'adresse physique du catalogue des tables de pages. On rajoute à cette adresse l'offset contenu dans les 10 premiers bits de l'adresse virtuelle. On obtient à cet endroit l'adresse physique de la table des pages. À cette adresse physique on rajoute les 10 bits intermédiaires de l'adresse virtuelle et on obtient à cet endroit l'adresse physique de la page à laquelle on rajoute les 12 bits de poids faible de l'adresse virtuelle.*

précédemment pour aller chercher le catalogue ou la table et on valide cette recherche dans une entrée du TLB.

Les entrées dans une table des pages sont organisées de façon à obtenir un certain nombre d'informations sur les pages mémoires physiques. Cette organisation est liée à une architecture matérielle du processeur. Par exemple, sur un processeur de la famille Intel, outre l'adresse recherchée [2], on trouvera les entrées suivantes :

- `_PAGE_PRESENT` : indiquant si la page est présente en mémoire physique ;
- `_PAGE_RW` : indique si la page est en lecture seule ;
- `_PAGE_USER` : indique si cette page est accessible en mode utilisateur ;
- ...

Gardons en mémoire que le mécanisme que nous venons de décrire s'applique en fait pour chaque processus. Un changement de contexte, du point de vue de la mémoire, doit donc sauvegarder le registre CR3 du processus courant, charger dans ce registre

2. Le lecteur curieux peut remarquer qu'une entrée de la table des pages contient 32 bits. 12 bits sont réservés pour décrire la page (présence, lecture / écriture, utilisateur, ...). Il reste donc 20 bits pour décrire une adresse physique de 32 bits ! En fait, les adresses doivent être alignées sur des multiples de 4 ko car chaque page contient 4096 octets. Donc les 12 bits de poids faible de toutes les adresses de début de page sont toujours nuls. La MMU prend donc les 20 bits de poids fort du mot de 4 octets contenu dans l'entrée de la table, rajoute 12 bits nuls et obtient ainsi l'adresse physique de la page.

la valeur sauvegardée du nouveau processus et s'occuper du TLB. Nous comprenons aussi pourquoi parmi ce que réalise un système d'exploitation au démarrage, la programmation de la MMU est fondamentale. Le gestionnaire de mémoire joue un rôle essentiel.

6.2 Les services du gestionnaire de mémoire

Le gestionnaire de mémoire est chargé d'assurer la gestion optimale de la mémoire. Son fonctionnement varie suivant les systèmes d'exploitation, mais les systèmes d'exploitation modernes et efficaces proposent généralement les services suivants : l'allocation et la libération de mémoire, la protection de la mémoire, la mémoire partagée, les fichiers mappés et la pagination.

L'allocation et la libération de mémoire

L'allocation de mémoire permet à un processus d'obtenir une zone mémoire disponible. Nous avons vu dans le chapitre 5 comment l'espace mémoire des processus était organisé et comment un processus pouvait demander l'allocation dynamique, en cours d'exécution, d'une zone mémoire.

Afin de répondre aux demandes des processus, le gestionnaire de mémoire doit :
- maintenir à jour une liste des pages mémoire libres ;
- chercher dans cette liste les pages à allouer au processus ;
- programmer la MMU afin de rendre ces pages disponibles ;
- libérer les pages attribuées à un processus lorsque celui-ci n'en a plus besoin.

Le système d'exploitation Unix propose les appels système `brk()` et `sbrk()` pour gérer la mémoire. Toutefois, ces appels sont très peu pratiques car ils ne fournissent que des blocs mémoire dont la taille est un multiple de la taille d'une page. De plus, les pages allouées à un processus forment un unique bloc dont les adresses logiques sont consécutives ; il n'est pas possible de libérer une page au milieu de ce bloc (voir aussi l'explication sur le tas, chapitre 5).

Les fonctions `malloc()` et `free()` permettent une gestion plus souple de la mémoire. Elles utilisent les appels système précédents pour obtenir des pages, mais un mécanisme indépendant permet d'allouer et de libérer des zones mémoire situées dans ces pages.

La protection

La protection de la mémoire est un service essentiel des systèmes d'exploitation. Il n'est pas acceptable aujourd'hui qu'une erreur dans un programme d'édition de textes, par exemple, provoque l'arrêt d'une machine essentiellement utilisée pour faire des calculs.

De façon générale, il n'est pas tolérable que le fonctionnement d'une machine (quelle que soit son utilisation) soit mis en cause par des erreurs logicielles qui pourraient être jugulées.

La principale fonction de la protection de la mémoire est d'empêcher un processus d'écrire dans une zone mémoire attribuée à un autre processus et, donc, d'empêcher qu'un programme mal conçu perturbe les autres processus. Pour accomplir cette tâche, le système d'exploitation doit programmer la MMU afin qu'un processus ne puisse avoir accès qu'aux pages mémoire qui lui appartiennent.

Le système d'exploitation crée alors un répertoire de pages par processus. Le répertoire de pages d'un processus contient les adresses des pages attribuées à ce processus et, comme les répertoires et les tables de pages sont stockées dans l'espace mémoire du système d'exploitation, seul le noyau peut les modifier.

Si un processus tente d'accéder à une page qui ne lui appartient pas, la MMU prévient instantanément le système d'exploitation (via une interruption) et ce dernier peut envoyer un signal au processus fautif.

Comment est-il alors possible, avec un tel système d'exploitation, de demander à un débugeur de lire les variables d'un processus qu'il scrute ? La réponse est simple et – nous espérons – évidente : il suffit de le demander au système d'exploitation grâce à l'appel système `ptrace()` qui permet à un processus de prendre le contrôle d'un autre (après vérification par le système d'exploitation que toutes les opérations demandées sont licites, bien sûr).

La mémoire partagée

La mémoire partagée permet à deux processus d'allouer une zone de mémoire commune. Cette zone peut être ensuite utilisée pour échanger des données sans appel du système d'exploitation : ce mécanisme est donc très rapide, mais nécessite en géneral d'être régulé par un mécanisme de synchronisation.

Nous avons vu dans le chapitre 4 un exemple d'utilisation de mémoire partagée : le chargement en mémoire d'une seule copie de code exécutée par plusieurs processus. Comme pour les autres services du gestionnaire de mémoire, cette tâche est grandement simplifiée par l'utilisation d'adresses virtuelles.

Un usage pratique de zones mémoire partagées sous Unix est par exemple exploité par le jeu *Doom* porté sur Linux. Cette version utilise un segment de mémoire partagée pour échanger les nouvelles images à afficher avec le serveur *X*, ce qui accélère considérablement l'affichage.

La conduite par défaut du gestionnaire de mémoire est d'empêcher tout processus d'accéder à l'espace mémoire d'un autre. Il est donc indispensable de faire appel à des fonctions particulières pour demander l'allocation d'une zone de mémoire partagée.

Le système d'exploitation Unix propose les fonctions suivantes :

`shmget()`: cette fonction crée ou recherche un segment de mémoire partagée. Une clé est nécessaire afin d'indiquer au système quel segment on souhaite récupérer.

shmat() : cette fonction associe le segment de mémoire partagée à une partie de la mémoire (virtuelle) du processus. Un pointeur est retourné, permettant ainsi au processus de lire ou écrire dans ce segment.

shctl() : cette fonction permet diverses manipulations du segment de mémoire partagée comme la suppression, la lecture des attributs ou la modification de ses attributs.

Le programme suivant crée un segment de mémoire partagée associé à la clé 911 :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>


#define SHM_KEY 911
#define SHM_SIZE 1024

main()
{
  int shmid;
  char *p;

  /* Je cree un segment de memoire partagee */

  shmid = shmget(SHM_KEY,SHM_SIZE,IPC_CREAT|0666);
  if(shmid==-1)
    {
      printf("Memoire partagee non disponible\n");
      exit(1);
    }

  /*
     J'affecte une partie de ma memoire virtuelle a ce segment
     l'adresse logique du segment est retournee dans  p
   */

  p=shmat(shmid,NULL,0);

  if(p== (char *) -1)
    {
      printf("La memoire partagee ne peut etre affectee\n");
      exit(1);
    }
```

```
  strncpy(p,"J'ecris dans la memoire partagee\n",SHM_SIZE);

  /* Je m'arrete... mais ne supprime pas le */
  /* segment de memoire partagee!!!         */
}
```

La présence du segment de mémoire partagée peut être mis en évidence par la commande `ipcs`.

Le programme suivant lit le contenu du segment de mémoire partagée et supprime ce dernier.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>


#define SHM_KEY 911
#define SHM_SIZE 1024

main()
{
  int shmid;
  char *p;

  /* Je cree un segment de memoire partagee */

  shmid = shmget(SHM_KEY,SHM_SIZE,0666);
  if(shmid==-1)
    {
      printf("Memoire partagee non disponible\n");
      exit(1);
    }

  /*
     J'affecte une partie de ma memoire virtuelle a ce segment
     l'adresse logique du segment est retournee dans  p
   */

  p=shmat(shmid,NULL,0);

  if(p== (char *) -1)
    {
```

```
      printf("La memoire partagee ne peut etre affectee\n");
      exit(1);
    }

  printf("Contenu de la memoire partagee : %s\n",p);

  /* Je supprime le segment de memoire partagee */
  shmctl(shmid,IPC_RMID,NULL);
}
```

Les fichiers mappés

Les fichiers mappés sont des fichiers dont le contenu est associé à une partie de l'espace adressable d'un processus. La lecture effective du fichier n'est réalisée qu'au moment de la lecture de la mémoire. Par ailleurs, le système est libre de réallouer la mémoire physique utilisée après cette lecture, puisque le contenu de cette mémoire n'est pas perdu : il est toujours possible de relire le fichier.

Ce service utilise une fois de plus la MMU : comme la MMU est capable d'intercepter tous les accès à des pages mémoires invalides, il suffit de déclarer les pages mémoires associées à un fichier comme invalides. Lorsqu'un processus tentera de lire l'une de ces pages, il produira un défaut de page qui sera détecté par le système d'exploitation. Ce dernier, au lieu d'envoyer un signal au processus comme il le fait habituellement, allouera une page de mémoire physique, chargera dans cette page le contenu du fichier et rendra la page demandée valide afin que le processus puisse y accéder.

Ce service est utilisé sur certains systèmes Unix et sous Windows afin de charger les bibliothèques de fonctions utilisées par la plupart des programmes. L'intérêt est double : le système peut toujours décharger les bibliothèques si cela est nécessaire et il n'a pas besoin de charger plusieurs fois les bibliothèques utilisées par plusieurs processus.

Sous Unix, la fonction utilisée pour mapper des fichiers en mémoire s'appelle `mmap()`. On utilisera `munmap()` pour supprimer la relation créée par `mmap()`.

Vous pouvez observer l'utilisation de ces fonctions en observant le lancement d'un processus dans un shell à l'aide de la commande

```
[moi@moi]~# strace prog_exe
```

Vous constaterez que les librairies partagées sont bel et bien mappées en mémoire.

La pagination

L'utilisation d'adresses virtuelles permet au système d'exploitation de gérer la mémoire principale par zones de grandes tailles (habituellement 4 ko) appelées des

pages. Cette gestion permet d'utiliser aussi bien des zones de la mémoire principale que des zones de la mémoire secondaire. Une conséquence directe de principe est que les processus peuvent demander l'allocation de plus de mémoire qu'il n'y en a de disponible dans la mémoire principale.

Certes, il serait possible dans ce cas-là de créer un grand fichier et de le mapper en mémoire. Cette solution n'est toutefois pas pratique pour plusieurs raisons : tout d'abord, le programmeur doit explicitement demander une telle association ; ensuite, la commande `malloc()` ne peut être utilisée pour gérer la mémoire associée au fichier.

Pour résoudre ce problème, différentes solutions ont été proposées et la solution qui est aujourd'hui la plus répandue est la pagination. Dans les sections suivantes, nous abordons rapidement les solutions historiques et nous détaillons le principe de pagination.

6.3 La gestion de la mémoire sans pagination

Le va-et-vient

Les premiers systèmes permettant la multiprogrammation utilisaient des partitions de la mémoire de taille fixe (*OS/360* avec le *MFT* : *Multiprogramming with a Fixed number of Tasks*). À chaque espace de la partition est associée une liste des tâches pouvant y être affectées de par leur taille. On peut aussi utiliser une liste unique et, lorsqu'un espace de la partition se libère, lui affecter la tâche dont la taille utilise au mieux l'espace libre. Remarquons que si on obtient ainsi un remplissage optimal de la mémoire centrale, l'exécution des tâches de tailles les plus faibles sont les moins prioritaires.

La mise en place du temps partagé ne rend plus possible l'utilisation d'une partition de la mémoire en zones de taille fixe. En effet, le nombre des processus existant à un moment donné peut, a priori, dépasser les capacités de la mémoire centrale de la machine. Il faut pouvoir organiser un **va-et-vient** (*swapping*) des processus entre la mémoire centrale et une mémoire secondaire, comme un disque. L'utilisation de partitions de taille fixe ferait alors perdre beaucoup trop de mémoire centrale par suite de la différence entre tailles des processus et tailles des partitions. Trop peu de processus pourraient être en même temps en mémoire et trop de temps se trouverait perdu en va-et-vient. Il faut donc disposer de partitions dont la taille s'adapte dynamiquement à la création de nouveaux processus.

Utilisation de partitions de taille variable

Avec des partitions de taille variable, l'allocation de place mémoire à des processus est plus complexe que dans le cas de partitions de taille fixe. De plus, il serait possible de déplacer des processus au cours de leur existence de manière à minimiser la fragmentation. En revanche, si un processus grandit au cours de son existence, l'espace dont il dispose peut ne plus lui suffire et il devra être déplacé dans la mémoire. Si

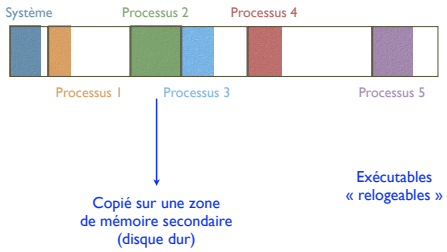


FIGURE 6.6 – *Les zones mémoire des processus « relogeables » peuvent être déplacées ou copiées sur d'autres zones mémoire, comme la mémoire secondaire.*

la mémoire centrale sature, on peut faire intervenir le mécanisme de va-et-vient de manière à utiliser l'espace disque disponible. Si ce dernier arrive aussi à saturation, il ne reste plus qu'à tuer le processus qui est demandeur de mémoire.

De manière à permettre des allocations de partitions de taille variable, la mémoire est divisée en partitions élémentaire d'allocation. Leur taille peut varier de quelques octets à quelques kilooctets suivant les systèmes. L'allocation s'effectue alors sur un ensemble d'unités voisines. Le problème consiste à savoir à chaque instant quelles sont les unités occupées, et ce, de manière à pouvoir trouver rapidement l'emplacement demandé par un nouveau processus. Il existe pour cela deux méthodes.

- L'utilisation d'une table de bits, maintenue par le système, dans laquelle chaque bit correspond à chaque unité physique. La valeur de ce bit indique si l'unité d'allocation est déjà occupée par un processus ou si elle est libre. La taille de la table dépend de la taille de l'unité élémentaire, mais il est facile de voir que l'emplacement occupé par la table ne représente qu'une faible proportion de la mémoire même pour des unités de taille relativement petite. En revanche, l'allocation d'un processus nécessitant k unités demande de trouver k unités libres voisines dans la table et, donc, entraîne une lecture complète de la table. Cette lenteur dans l'allocation des processus importants fait que ce système est peu employé.
- L'utilisation d'une **liste chaînée** dans laquelle sont maintenues la liste des emplacements occupés et la liste des emplacements libres. Un emplacement libre est une suite d'unités élémentaires libres entre deux processus. Un emplacement occupé est un ensemble d'unités élémentaires occupées par un même processus. La liste peut être triée, par exemple, par ordre d'adresses croissantes. Plusieurs algorithmes sont envisageables pour déterminer comment choisir la zone libre, où sera chargé le processus créé ou déplacé :
  - **L'algorithme du premier emplacement libre** (*first fit*) : on place le processus dans le premier emplacement libre rencontré dans le parcours de la liste, qui peut le contenir. L'algorithme est rapide, puisque la recherche est courte.

En revanche, il crée potentiellement de nombreux petits emplacements libres, qui seront difficiles à allouer.

- **L'algorithme de l'emplacement libre suivant** (*next fit*) : idem que l'algorithme du premier emplacement libre, mais la recherche reprend de l'endroit où le parcours s'était arrêté à la précédente recherche. Les résultats sont moins bons que pour l'algorithme du premier emplacement libre.
- **L'algorithme du meilleur ajustement** (*best fit*) : on parcourt entièrement la liste à la recherche de l'emplacement libre dont la taille correspond le mieux avec la taille mémoire nécessaire au processus. Ceci permet d'éviter un gaspillage de mémoire par fractionnement de la mémoire libre, mais l'allocation est plus longue du fait du parcours complet de la liste.
- **L'algorithme du plus grand résidu** (*worst fit*) : on parcourt entièrement de manière à choisir l'emplacement libre tel que l'allocation de la mémoire nécessaire au processus dans cet emplacement donnera naissance à un emplacement libre résiduel de la taille la plus grande possible. La simulation montre que cet algorithme ne donne pas de bons résultats.

Plusieurs optimisations de ces algorithmes sont possibles. Par exemple, l'algorithme de placement rapide (*quick fit*) peut utiliser plusieurs listes séparées suivant la taille des zones libres, ce qui accélèrent les recherches. Cependant, lors de la disparition d'un processus de la mémoire centrale, le maintien d'une seule liste pour les emplacements libres et occupés permet par une simple consultation dans la liste d'effectuer les fusions avec les emplacements libres voisins. Dans le cas de l'utilisation de plusieurs listes, la recherche des fusions est plus longue. Le problème est le même dans le cas d'un classement par taille des emplacements libres (classement qui allie les avantages *best fit* et *first fit* pour l'allocation de la mémoire). Pour obtenir des résultats satisfaisants aussi bien pour l'allocation que pour la libération, il faut mettre au point un compromis.

L'espace de va-et-vient

Le principe du va-et-vient entraîne pour certains processus la nécessité d'être recopiés sur un disque en raison du manque de place en mémoire centrale. Certains systèmes n'allouent pas pour chaque processus créé une place sur le disque, mais réservent une zone de va-et-vient (*swap area*), destinée à accueillir les processus sur le disque (c'est le cas d'Unix). L'allocation de la mémoire dans cette zone du disque répond au même principe de gestion que la mémoire centrale.

Le mécanisme d'*overlay*

Une autre contrainte est la taille des processus. Si les processus sont trop importants pour la mémoire centrale, il ne pourra guère y avoir qu'un seul processus à la fois dans la mémoire centrale. Ce qui enlève tout intérêt au problème des partitions qui vient d'être étudié. Cependant, pour un processus quelconque, le code étant exécuté de manière relativement linéaire, les instructions et les données dont on doit disposer en

mémoire centrale ne représentent qu'une partie de la totalité de l'image mémoire du processus. On pourrait, donc, se contenter d'avoir en mémoire centrale la partie utile de code et mettre sur disque, dans la zone de va-et-vient, le reste de l'image mémoire du processus. L'utilisation de partitions serait, alors, de nouveau possible.

Une première idée consiste à utiliser des **segments de recouvrement** (*overlays*). Cette méthode était utilisée par les programmeurs pour écrire de gros programmes. Comme elle était très fastidieuse à programmer, on chargea le système de gestion de la mémoire de la mettre en œuvre. Elle consiste à découper le programme en parties cohérentes, qui s'exécutent les unes après les autres en occupant successivement le même emplacement de la mémoire centrale, chacune recouvrant son prédécesseur. Les segments non actifs d'un processus sont stockés sur une mémoire secondaire comme un disque. Un processus en attente du chargement d'un de ses segments est, en fait, en attente d'entrée/sorties, le processeur peut donc être affecté à un autre processus pendant le chargement du segment.

Ce procédé n'est, cependant, plus utilisé en tant que tel aujourd'hui. On ne le trouve plus que sur les systèmes n'offrant pas une possibilité de mémoire virtuelle. L'idéal est en effet d'offrir à l'utilisateur l'impression que la mémoire centrale de l'ordinateur est beaucoup plus importante que ce qu'elle n'est en réalité. C'est le concept de **mémoire virtuelle** : la taille mémoire de la machine apparaît comme plus grande par une gestion fine déterminant quelles parties des codes des processus doivent se trouver en mémoire centrale, et quelles parties ne sont pas utilisées et peuvent être reléguées sur disque. Deux méthodes existent pour effectuer cette gestion : la méthode des mémoires segmentées et la méthode des mémoires paginées.

Les mémoires segmentées

Le principe consiste à adopter la vision logique des utilisateurs. À chaque fonction du programme ou ensemble de données, on fait correspondre un espace mémoire. La partition logique d'un processus correspond donc à une partition logique en un ensemble d'espaces mémoires, **les segments**. Seuls les éléments nécessaires au bon déroulement du processus sont chargés en mémoire à un instant donné. À chacun de ces espaces, sont associées des données qui permettent au système de déterminer la longueur du segment.

Une telle méthode pour représenter une mémoire virtuelle est assez difficile à mettre en œuvre. C'est pourquoi les mémoires paginées sont beaucoup plus répandues.

6.4 La pagination

Le principe

Le principe de la pagination est de découper l'image mémoire d'un processus en blocs de taille donnée : les pages. Alors que le système des segments de recouvrement suppose une unité logique du segment, ce n'est pas le cas de la page. Logiquement, le

processus voit donc un espace d'adressage linéaire beaucoup plus vaste que la place physique dont il dispose dans la mémoire centrale. Mais le fait qu'une partie de cet espace se trouve sur un disque est complètement transparent pour le processus : on dit qu'il dispose d'un espace d'adressage virtuel et qu'il utilise des adresses virtuelles. C'est la traduction des adresses virtuelles en adresses physiques qui permet au système d'exploitation de gérer le chargement des pages nécessaires à la bonne exécution du processus.

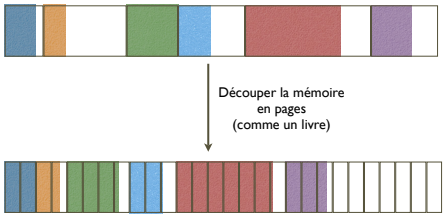


FIGURE 6.7 – *Le principe de la pagination*

La mémoire physique est partagée en cases mémoire (*page frames*). À sa création, un processus se voit allouer un certain nombre de ces cases, elles représentent le nombre de pages dont le processus pourra disposer à la fois en mémoire centrale. Un accès à une adresse par le processus s'effectue soit réellement si l'adresse est située sur une page résidant en mémoire centrale, soit provoque une erreur, si la page adressée ne s'y trouve pas, l'accès mémoire attendra alors pour s'effectuer le chargement depuis le disque de la page en question. Le processus étant en attente d'une entrée/sortie, le processeur pourra être alloué à un autre processus pendant le chargement de la page. Ce déroutement de l'accès mémoire par le système s'appelle un défaut de page (*page fault*). La gestion de la pagination est composée de mécanismes matériels constituant l'unité de gestion mémoire, de manière à être plus efficace. Ainsi, par exemple, l'utilisation de la mémoire virtuelle dans Unix `4.4BSD` est largement dépendante des mécanismes matériels de gestion de la mémoire du `VAX`. En effet, la représentation de l'adresse virtuelle comprend le numéro de page et l'adresse dans la page. Par un masque, l'unité de gestion mémoire a alors à sa disposition le numéro de la page et peut examiner si la page en question est en mémoire en consultant la **table des pages** du processus, dont elle dispose.

Les systèmes mixtes

Des systèmes mixtes pagination-segmentation ont été mis au point. `Multics` a tenté d'élaborer un système où, à chaque segment, était alloué un objet logique (fichier, zone de données, zone de texte...) et où chaque segment était constitué de plusieurs pages. Les 36 bits d'adressage de `Multics` se sont révélés insuffisants pour la réalisation

de ce système. Aujourd'hui, les machines à base de 68000 utilisent sous une forme simplifiée un concept de ce genre avec une mémoire virtuelle mettant à la disposition de 16 processus 1024 pages de 2 ko ou 4 ko. Ainsi, pour des pages de 4 ko, chaque processus dispose de 4 Mo de mémoire virtuelle de 1024 pages. Au lieu d'utiliser un tableau de pages à 1024 entrées, l'unité de gestion mémoire dispose d'une table de 16 sections (une section par processus en mémoire), chaque segment possède 64 entrées de descripteurs de segments, chaque descripteur fournissant un pointeur vers une table de pages à 16 entrées. Un des avantages est de disposer de possibilités simples de partage de pages : il suffit de faire pointer le pointeur du segment vers la même table de pages.

Les algorithmes de remplacement de pages

S'il est facile de calculer quelle page on veut voir chargée en mémoire, il est moins simple de déterminer quelle page déjà chargée va être recouverte lors du chargement de la nouvelle page, en l'absence de case mémoire libre pour le processus. L'efficacité de l'unité de gestion de la mémoire va dépendre du choix de cet algorithme de remplacement. L'algorithme optimal est facile à décrire : il s'agit de laisser recouvrir la page qui ne sera pas référencée avant le temps le plus long. Il est malheureusement impossible à mettre en œuvre. Il existe de nombreux algorithmes proposés :

- **L'algorithme de la page non récemment utilisée** (`NRU` – *Not Recently Used*) : retire une page au hasard de l'ensemble de numéro le plus bas des ensembles suivants :
  - Ensemble 0 : ensemble des pages qui n'ont jamais été lues, ni écrites.
  - Ensemble 1 : ensemble des pages qui n'ont jamais été lues, mais ont été écrites.
  - Ensemble 2 : ensemble des pages qui n'ont jamais été écrites, mais ont été lues.
  - Ensemble 3 : ensemble des pages qui ont été lues et écrites.
- **L'algorithme première entrée, première sortie** (`FIFO`, *First In, First Out*) : retire la page qui a été chargée depuis le plus longtemps. On peut l'utiliser en même temps que l'algorithme `NRU`, en l'appliquant à chaque ensemble, plutôt que de choisir au hasard.
- **L'algorithme de la page la moins récemment utilisée** (`LRU`, *Least Recently Used*) : on retire la page qui est restée inutilisée pendant le plus longtemps.

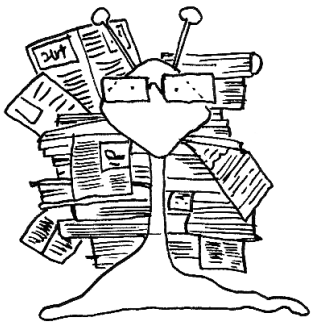
L'algorithme `NRU` n'est pas optimal, mais il est souvent suffisant. Il est, de plus, simple à mettre en œuvre, car il ne nécessite pas de matériel spécialisé, au contraire de l'algorithme `LRU`. L'algorithme `FIFO` est douteux, car il peut entraîner le retrait d'une page très utile, référencée souvent depuis le début d'exécution du processus.

6.5 Conclusion

L'introduction de la mémoire virtuelle et l'utilisation de composants dédiés pour gérer la traduction vers les adresses physiques offre de grandes possibilités, notamment le fait de pouvoir simuler une allocation continue de mémoire et permet ainsi le déplacement des processus sans qu'un quelconque mécanisme de réorganisation interne soit nécessaire. Ce principe de virtualisation permet de s'abstraire des couches matérielles et nous verrons qu'il s'applique à d'autres fonctions d'un système d'exploitation.

7

Le système de fichiers



Le système de fichiers d'un système d'exploitation est la partie la plus couramment sollicitée par l'utilisateur. Par exemple, l'écriture d'un programme en langage `C` passe par la création d'un fichier dans lequel les différentes lignes de code seront conservées et engendre généralement la création de plusieurs autres fichiers (voir chapitre 4). De même, les données d'entrées ou de sorties d'un programme sont très souvent conservées dans des fichiers qui peuvent être lus ou écrits.

L'organisation du système de fichiers est donc primordiale et elle conditionne l'utilisation efficace de la machine. Néanmoins, comme pour les autres services rendus par les systèmes d'exploitation, l'objectif est d'offrir le service le plus efficace tout en proposant une vision simple à l'utilisateur qui ne veut pas se préoccuper des détails d'organisation physique du système.

Ce chapitre montrera ainsi que, au delà des fichiers traditionnels ou de la notion de fichier traditionnelle auxquels l'utilisateur est habitué, le système de fichier représente en fait une abstraction ultime de tout échange et stockage de données, quelle que soit la forme qu'ils peuvent prendre.

Nous allons aborder dans ce chapitre les aspects généraux des systèmes de fichiers en partant des plus visibles aux structures les plus internes.

7.1 Les services d'un système de fichier

Les fichiers

La notion principale manipulée lors de l'utilisation d'un système de fichiers est, on peut s'y attendre, le fichier. Un fichier est une suite ordonnée d'octets [1] qu'un programme peut lire ou modifier. Dans le cas d'un programme écrit en langage `C` ou un texte destiné à être typographié par LaTeX, par exemple, chaque octet peut représenter un caractère ou participer à cette représentation [2] ; différentes normes définissent les valeurs associées à chaque caractère (voir la section 4.1 du chapitre 4). Nous parlons dans ce cas d'un fichier texte par opposition aux fichiers binaires créés selon d'autres conventions (normes GIF, JPEG ou MP3, par exemple) Notons que souvent la plupart des systèmes d'exploitation ne font pas de distinction entres ces fichiers : ce sont les programmes qui les manipulent qui doivent utiliser les bonnes conventions pour interpréter correctement le contenu des fichiers qu'ils utilisent.

Afin de rendre cette suite d'octets disponible après arrêt de l'ordinateur, cette suite est généralement enregistrée sur un support permanent que nous appellerons périphérique de stockage (disque dur dans la plupart des cas, mais nous pouvons aussi citer disque magnéto-optique, mémoire permanente et, dans certains cas, les bandes magnétiques).

Chaque fichier est associé à un nom qui sera utilisé ensuite pour l'identifier. À l'aide de ce nom, un programme peut demander au système d'exploitation la lecture ou l'écriture du fichier. Pour beaucoup d'applications, le chargement complet d'un fichier en mémoire n'est ni utile ni préférable (pour des raisons de coûts, la capacité de la mémoire est très souvent inférieure à celle des périphériques de stockage). Dans ce cas, un programme utilisera le fichier comme une bande magnétique, en lisant successivement des petites parties d'un fichier. On parle dans ce cas d'un accès séquentiel. Pour certaines applications, il peut être utile de lire les informations enregistrées dans un ordre quelconque. Dans ce cas on parlera d'accès aléatoire.

La plupart des systèmes d'exploitation nécessite avant les opérations de lecture et d'écriture l'ouverture d'un fichier. Cette ouverture préalable permet au système d'exploitation de ne rechercher le fichier d'après son nom qu'une seule fois. Une structure est créée afin de mémoriser des informations telles que : où est situé le fichier, quelle est la position courante dans le fichier (quelle partie du fichier sera lue ou écrite à la prochaine opération). Selon les systèmes d'exploitation, le résultat retourné lors de l'ouverture est appelé descripteur de fichier (Unix) ou *handle* (Mac OS, Windows). Il s'agit d'une forme de pointeur sur les informations relatives au fichier ouvert (voir les exemples du chapitre 11).

1. Notons l'exception de Mac OS, où un fichier est associé à deux suites d'octets appelés « Data fork » et « Ressource fork ».
2. Nous avons vu qu'un caractère, selon le système de codage UTF-8 peut être codé sur plusieurs octets.

Les opérations généralement offertes par la plupart [3] des systèmes de fichiers sont les suivantes :

L'ouverture d'un fichier : cette opération consiste à rechercher d'après son nom un fichier sur le périphérique de stockage. Le résultat de cette recherche sert à initialiser une structure qui sera nécessaire aux manipulations suivantes du fichier. Généralement, les permissions d'accès à un fichier sont vérifiées à ce moment, ce qui dispense le système d'une telle vérification lors de chacune des opérations suivantes.

La création d'un fichier : cette opération consiste à créer sur le périphérique de stockage, un nouveau fichier, vide. Ce fichier est immédiatement ouvert (un descripteur de fichier ou *handle* est retourné), puisque la création d'un fichier vide est rarement une fin en soi.

L'écriture dans un fichier : cette opération consiste à modifier le contenu d'un fichier à partir d'une « position courante ». Dans le cas où la position courante arrive au-delà de la taille du fichier, le fichier s'accroit afin de permettre le stockage de ces octets.

La lecture dans un fichier : cette opération consiste à lire une série d'octets dans un fichier, à partir d'une « position courante ». En cas de dépassement de la taille du fichier, une information « fin de fichier » est retournée.

Le déplacement à l'intérieur d'un fichier : cette opération consiste à changer la position courante. Cette opération permet un accès direct à n'importe quelle partie d'un fichier sans lire ou écrire tous les octets précédents.

La fermeture d'un fichier : consiste à supprimer les structures du système d'exploitation associées à un fichier ouvert.

La suppression d'un fichier : cette opération consiste à supprimer sur le périphérique de stockage, les données associées à un fichier. Cette suppression libère une place sur le disque qui pourra être réutilisée pour agrandir des fichiers existants ou créer d'autres fichiers [4].

La lecture ou la modification des caractéristique d'un fichier : cette opération permet de lire les caractéristiques associées à un fichier. Certaines caractéristiques sont disponibles dans la plupart des systèmes de fichiers : taille, dates de création et/ou de modification, propriétaire, droits d'accès. Parfois des caractéristiques spécifiques supplémentaires peuvent être associées : les droits d'accès, un attribut « archive » utilisé sous Windows pour indiquer si un fichier est sauvegardé, le type de fichier sous Mac OS. Notons que certains systèmes de fichiers sont conçus pour permettre la création d'attributs non prévus initialement.

3. Il existe quelques exceptions. Par exemple, un système de fichiers organisé selon la norme ISO 9660, conçue pour les CD-ROM, ne permet pas la modification de fichiers.

4. Quelques systèmes de fichiers sont susceptibles de ne pas rendre réutilisable la place occupée par un fichier à cause de limitations du périphérique de stockage (disque WORM, bande magnétique).

Les répertoires

Un nommage « à plat » des fichiers serait peu pratique à gérer au delà d'une centaines de fichiers, ainsi tous les systèmes de fichiers [5] permettent la création de répertoires qu'un utilisateur utilisera pour rassembler un ensemble de fichiers. Pour rendre le classement encore plus efficace, ces répertoires peuvent contenir à leur tour d'autres répertoires, ce qui donne une structure arborecente [6].

L'ouverture ou la création d'un fichier nécessite, en plus du nom du fichier, la liste des répertoires à parcourir pour trouver le nom du fichier. Cette liste, appelée chemin d'accès se présente sous la forme d'une unique chaîne de caractères dont un caractère particulier est utilisé pour séparer les noms de répertoires et le nom du fichier. Notons que chaque système d'exploitation utilise des conventions différentes (« / » sous Unix, « \ » sous MS-DOS et Windows, « : » sous les anciennes versions de Mac OS).

Pour faciliter l'accès à des fichiers situés dans un même répertoire, une notion de répertoire courant est généralement utilisée et permet l'ouverture d'un fichier à l'aide d'un chemin « relatif » au répertoire courant (ce chemin ne contient que le nom du fichier si ce dernier est directement dans le répertoire courant).

Les opérations usuelles portant sur les répertoires sont les suivantes :

- la création d'un répertoire ;
- la suppression d'un répertoire ;
- la lecture d'un répertoire, afin d'obtenir la liste des fichiers ; cette opération nécessite généralement plusieurs opérations élémentaires, qui peuvent distinguer une ouverture de répertoire, la lecture (répétée) d'un nom de fichier, et la fermeture ;
- le changement de nom ou le déplacement d'un fichier.

Notons que les opérations portant sur les fichiers portent parfois implicitement sur les répertoires du chemin d'accès associé à ce fichier. Par exemple, la création d'un fichier dans un répertoire est une modification du répertoire.

Le système de fichiers virtuel

Chaque système de fichiers possède des spécificités propres, ainsi un système de fichiers d'un Macintosh précise la position des icônes associées à chacun de ces fichiers [7], alors qu'un système de fichiers d'un système Unix précisera le propriétaire de chaque fichier ainsi que les droits d'accès. Ces différences nécessitent des fonctions de manipulation de fichiers adaptées. Ainsi la fonction de lecture d'un fichier sera différente selon les systèmes d'exploitation.

5. Nous avons encore une exception : la version 1 de MS-DOS ne connaissait pas la notion de répertoire, cela ne pouvait être toléré que sur des disquettes de faibles capacités.

6. Cette fois nous pouvons noter le système de fichiers MFS des premiers Macintosh, dont les répertoires ne pouvaient contenir d'autres répertoires.

7. De nombreux fichiers sont ainsi disponibles sous Mac OS X lorsque l'on observe le contenu d'un répertoire : `.DS_Store`,... Ces fichiers sont utilisés par le « Finder » pour le rendu graphique des répertoire et des fichiers.

Pour permettre des échanges de fichiers entre systèmes d'exploitation différents, pour permettre aussi l'utilisation de périphériques de stockage différents, les systèmes d'exploitation supportent plusieurs systèmes de fichiers et donc — par exemple — plusieurs fonctions de lecture. On pourrait imaginer les fonctions `mac_read()`, `unix_read()`, `msdos_read()`, etc.

Ce jeu de fonctions serait peu pratique pour le programmeur : ce dernier devrait adapter son programme à chaque système de fichier. Un tel programme serait par ailleurs incompatible avec tous les systèmes de fichiers qui auraient été développés après l'écriture du programme.

Pour éviter cela, le système d'exploitation présente aux programmes et à l'utilisateur une présentation unifiée des systèmes de fichiers supportées. Par exemple, la lecture est réalisée par une unique fonction qui s'adapte selon le cas. Cette présentation unifiée est souvent appelée « système de fichiers virtuel ».

Dans certains systèmes d'exploitation, le système de fichiers virtuel présente l'ensemble des systèmes de fichiers sous la forme d'une arborescence unique. L'opération appelé « montage » consiste à associer à un répertoire d'un système de fichier, un autre système de fichier. Dans les cas des systèmes Unix, un système de fichiers particulier — le système de fichiers racine — sert de base à cette arborescence. Ci-dessous, le résultat (simplifié) de la commande `mount` présente la liste des systèmes de fichiers considérés par le système d'exploitation avec pour chacun le nom de périphérique utilisé pour le stocker, le répertoire permettant à un programme d'accéder à ce système de fichier, le type de système de fichiers :

```
/dev/wd0s1a on / (ufs)
mfs:362 on /tmp (mfs)
/dev/wd0s1f on /usr (ufs)
/dev/wd0s1e on /var (ufs)
procfs on /proc (procfs, local)
/dev/cd0c on /cdrom (cd9660)
```

Nous remarquons en deuxième ligne, un système de fichiers stocké en mémoire, cela permet d'accélérer la création de fichiers temporaires, mais ces derniers ne survivront pas à un redémarrage du système. Par ailleurs, l'avant-dernière ligne cite un système de fichiers stocké nulle part, mais présentant des informations sur les processus comme s'ils étaient stockés dans des fichiers.

Dans le cas de Windows NT, une arborescence créée en mémoire à l'initialisation du système permet le montage des systèmes de fichiers. Le répertoire racine d'un système de fichiers est alors associé au nom du périphérique le contenant (par exemple `\Device\Floppy0`). Notons que cette arborescence n'est pas rendue visible à l'utilisateur qui continuera à utiliser des conventions du système MS-DOS (`A:` pour la disquette, `C:` pour le premier disque, etc.).

La présentation unifiée des fichiers situés sur des périphériques différents étant bien pratique, elle est souvent généralisée aux périphériques mêmes. Ainsi, un fichier et

une bande magnétique seront utilisables avec les mêmes fonctions. Pour permettre des accès aux périphériques, des noms leur sont donnés :

- Sous MS-DOS, certains périphériques sont associés à des chaînes de caractères [8] (ce qui peut rendre certains fichiers inaccessibles lors de l'ajout d'un pilote de périphérique ou rendre impossible la création d'un fichier homonyme) ;
- Les systèmes de fichiers standards des systèmes Unix permettent la création des références aux périphériques appelés « fichiers spéciaux ». Par convention, ces fichiers spéciaux sont placés dans le répertoire `/dev` ;
- Sous Windows NT, les pilotes de périphériques enregistrent les périphériques qu'ils gèrent dans l'arborescence utilisée pour le montage des systèmes de fichiers. Nous pouvons citer par exemple `\Device\Floppy0`, où un même nom est utilisé pour un périphérique et le répertoire utilisé pour monter le système de fichiers qu'il contient.

Autres services

Selon les systèmes d'exploitation, des services supplémentaires sont rendus par un systèmes de fichiers. Les services les plus fréquents et/ou utiles sont présentés ci-dessous.

Le verrouillage

L'accès simultané à un même fichier par deux processus peut corrompre ce fichier. Par exemple, si deux processus sont chargé de chercher dans une base de données une chambre d'hôtel libre et la réserver, ces deux processus risquent de trouver la même chambre et, sans concertation, l'associer à deux clients différents... Pour éviter ces problèmes, le système d'exploitation peut verrouiller un fichier pour assurer à un processus qu'aucun autre ne l'utilise en même temps. Deux types de verrous sont généralement disponibles : les verrous exclusifs, qui empêchent tout autre processus de verrouiller le même fichier et les verrous partagés qui tolèrent la pose d'autres verrous partagés.

Généralement, les verrous exclusifs sont utilisés lors des écritures, et les verrous partagés, lors des lectures : une lecture simultanée par plusieurs processus ne risque pas de corrompre un fichier.

Selon les systèmes d'exploitation, les verrous sont impératifs (*mandatory*) ou indicatifs (*advisory*). Les premiers empêchent non seulement la pose d'autres verrous, mais aussi les accès au fichier alors que les seconds ne protègent que les accès entres programmes qui prennent soin de poser des verrous lorsque nécessaire. Sous Unix, par exemple, les verrous sont indicatifs, partant du principe qu'une application qui ne verrouillerait pas correctement un fichier est de toute manière susceptible de le corrompre.

8. Cela ne concerne pas tous les périphériques... certains ne sont pas gérés de manière uniformisée, c'est au programmeur de choisir le jeu de fonctions adapté au périphérique utilisé.

Par ailleurs, la pose d'un verrou peut porter sur l'ensemble du fichier (c'est plus simple, mais imaginez que vous devez attendre la réservation d'un chambre voisine pour consulter les tarifs de l'hôtel) ou sur une partie limitée.

La notification d'évolution

La notification d'évolution permet à un processus d'être averti de l'évolution d'un fichier ou d'un répertoire. Cela peut être utilisé par une interface graphique pour rafraîchir une fenêtre à mesure qu'un répertoire est modifié.

L'interface de programmation *inotify* fournit un mécanisme permettant de surveiller les événements qui interviennent dans le système de fichiers. On peut à la fois l'utiliser pour surveiller un fichier particulier ou un répertoire entier. Cette interface propose, entre autres, les appels systèmes suivants :

- `inotify_init()` : cette fonction retourne un descripteur de fichier qu'il conviendra de lire pour observer la venue d'événements ; la liste d'événements est vide ;
- `inotify_add_watch()` : cette fonction ajoute des événements à surveiller ;

Les événements qui se produisent sur le système de fichiers (et relatifs aux fichiers et/ou répertoires surveillés) seront lus de manière très « classique » sur le descripteur de fichier renvoyé par l'appel de `inotify_init()`. Cette API peut être particulièrement utile pour réaliser une supervision de données utilisateurs et de données relatives à un site web.

Les quotas

Les quotas permettent à un administrateur de limiter pour chaque utilisateur le volume occupé par ses fichiers sur un disque.

La création de liens

La création de liens permet l'association de plusieurs noms à un même fichier. Les systèmes Unix considèrent les liens *hard* où un fichier possède plusieurs noms qui sont considérés de la même manière par le système et les liens symboliques qui associent à un nom de fichier le nom d'un autre fichier. Le système Mac OS ne gère que les liens symboliques, appelés « alias ». Sous Windows, les liens symboliques sont simulés par l'interface graphique pour pallier l'absence de liens symboliques gérés par le système. Appelés raccourcis, ces liens ne sont que des fichiers normaux interprétés de façon particulière par l'interface graphique.

Le *mapping* en mémoire

Le *mapping* d'un fichier en mémoire permet à l'aide du gestionnaire de la mémoire d'associer une partie de l'espace d'adressage d'un processus à un fichier. Les accès au fichier peuvent alors être programmés comme de simples accès à un tableau en mémoire, le système d'exploitation chargeant les pages depuis le fichier lorsque nécessaire. La

vérification de la présence d'une page est réalisée de façon matérielle (par la MMU) à chaque accès, cela peut permettre un gain de performance pour des applications utilisant beaucoup d'accès aléatoires à un même fichier et dont beaucoup sont susceptibles de lire les mêmes pages. C'est notamment le cas lors de l'exécution d'un programme qui est généralement mappé avec les bibliothèques de fonctions qu'il utilise.

Les accès asynchrones

Les accès asynchrones permettent à un programme la planification d'accès à un fichier sans attendre le résultat. Un appel système permet d'attendre ensuite le résultat demandé au préalable. Certains systèmes permettent aussi la demande d'exécution d'une fonction particulière à la fin de cette opération. Enfin notons l'interface POSIX qui permet la définition de priorités associées aux accès asynchrones.

7.2 L'organisation des systèmes de fichiers

Généralités

Un disque dur présente au système d'exploitation un ensemble de blocs de même taille (généralement 512 octets), accessibles d'après leurs numéros. À l'aide de ce service limité (les disques durs ne connaissent pas la notion de fichiers), le système d'exploitation réservera certains blocs pour stocker les informations nécessaires à la gestion d'un système de fichiers. Ces informations, appelées « méta-données » ne sont pas accessibles directement par l'utilisateur ou ses programmes.

Les méta-données fournissent généralement les informations suivantes :

- Les caractéristiques du système de fichiers lui-même (nombre de blocs total, nombre de blocs disponibles, nombre de fichiers, etc.). Le terme superbloc est parfois utilisé pour identifier le bloc stockant toutes ces informations. De telles informations, indispensables à l'utilisation d'un système de fichiers, sont souvent dupliquées dans plusieurs blocs par mesure de sécurité.
- Les caractéristiques de chaque fichier (taille, listes des blocs associés au fichiers, etc.).
- L'état libre ou alloué de chaque bloc.

Afin de faciliter la gestion des blocs, ces derniers ne sont pas toujours gérés individuellement mais par agrégats (*clusters*) qui deviennent l'unité minimum d'allocation utilisée par le système de fichiers. Certains systèmes d'exploitations (ou plutôt leur documentation) utilisent le terme « blocs » pour désigner ces agrégats. Le sens du terme bloc dépend alors du contexte : parle t-on du système de fichiers ou du périphérique ? L'usage des termes « blocs logiques » et « blocs physiques » permet de lever l'ambiguïté.

Le système de fichiers Unix (UFS)

Le système de fichiers UFS (Unix File System) [9] place les caractéristiques des fichiers dans un zone de taille fixe appelée table des *i-nodes*. Cette table contient pour chaque fichier une structure appelée *i-node* fournissant les caractéristiques d'un fichier, à l'exception du nom. Les noms des fichiers sont situés dans les répertoires qui ne sont qu'une liste de noms associés à leur numéro d'*i-node*. Une caractéristique rare des systèmes Unix est de permettre l'association d'un même *i-node* (et donc d'un même fichier) à plusieurs noms (situés éventuellement dans plusieurs répertoires). Un compteur de références permet la suppression du fichier (et la libération de l'*i-node*) après la suppression du dernier lien avec un nom de fichier.

La commande `ls -i` permet d'obtenir les numéros d'inode. Le résultat de la commande `ls -li` montre un répertoire contenant 3 noms : `a`, `b` et `c`. Ces deux derniers sont associés à un même *i-node* (1159904) et donc à un même fichier. Le compteur de référence associé à ce fichier a la valeur 2, ce qui évite la suppression du fichier lors de la suppression d'un des noms :

```
~/WORK/Cours/essai$ls -il
total 0
1159903 -rw-r--r--  1 loyer  loyer  0 12 nov 15:18 a
1159904 -rw-r--r--  2 loyer  loyer  0 12 nov 15:18 b
1159904 -rw-r--r--  2 loyer  loyer  0 12 nov 15:18 c
~/WORK/Cours/essai$rm c
~/WORK/Cours/essai$ls -il
total 0
1159903 -rw-r--r--  1 loyer  loyer  0 12 nov 15:18 a
1159904 -rw-r--r--  1 loyer  loyer  0 12 nov 15:18 b
```

Plusieurs types d'*i-nodes* sont considérés par le système Unix :

Les fichiers normaux sont les plus courants : ce sont ceux que vous créez habituellement.

Les répertoires servent à organiser les fichiers.

Les liens symboliques sont des références à d'autres fichiers.

Les tuyaux nommés permettent à deux programmes de s'échanger des données : l'un écrivant dans ce tuyaux et l'autre lisant ce même tuyaux. Vous pouvez créer de tels tuyaux avec la commande `mkfifo`. Le chapitre 15 aborde cette notion.

Les sockets permettent à un programme client d'échanger des données avec un serveur. L'usage des sockets est plus complexe que celui des tuyaux nommés, mais permet une communication bidirectionnelle et individualisée avec chaque programme

9. L'intérêt de ce système de fichiers est plutôt d'ordre historique, des optimisations importantes ayant étés introduites en 1984 pour former le Fast File System. Ces optimisations seront présentées ensuite.

client. Aucune commande n'est disponible pour créer de tels sockets : il sont créés par chaque serveurs utilisant ces sockets. Le chapitre 16 donne plus de détails sur l'utilisation des sockets.

Les fichiers spéciaux sont associés à des périphériques et permettent des accès à ces derniers comparables aux accès aux fichiers. Seul l'administrateur est habilité à créer de tels fichiers spéciaux.

Un *i-node* associé à un fichier normal contient la liste des 12 premiers blocs occupés par le fichier qu'il représente et, si nécessaire, le numéro d'un bloc d'indirection simple contenant les numéros des blocs suivants. Pour des fichiers plus gros, l'*i-node* contient le numéro d'un bloc d'indirection double contenant une liste de numéros de blocs d'indirection simple. Enfin, un dernier numéro de blocs désigne un bloc d'indirection triple dont le contenu est une liste de blocs d'indirection double (voir figure 7.1).

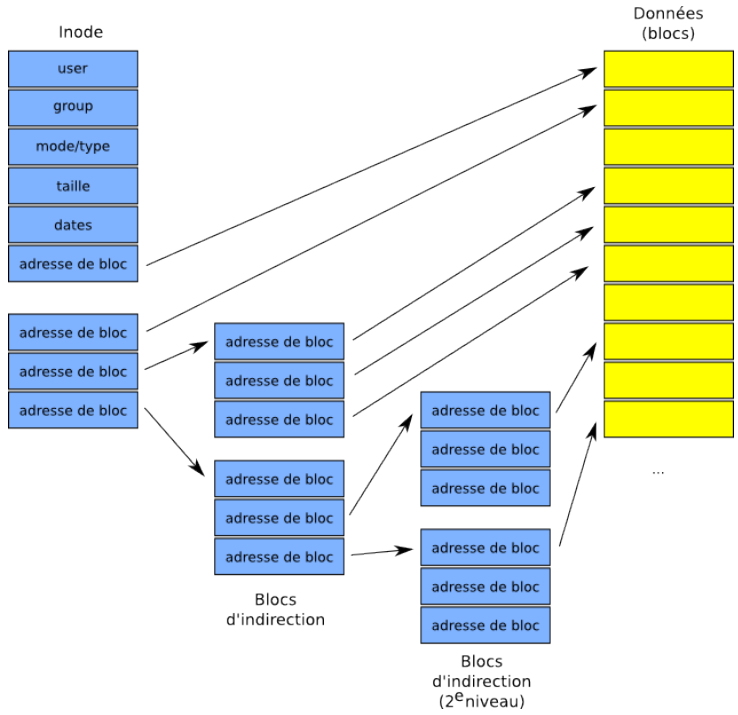


FIGURE 7.1 – *Le système de fichiers sous Unix.*

Soit un système de fichiers constitués de blocs de 4096 octets, chaque numéro de blocs occupant 4 octets, un bloc d'indirection simple permet de désigner jusqu'à 1024 blocs d'un fichier et donc 4 Mo. De même, un bloc d'indirection double désigne

jusqu'à 1024 blocs d'indirection simple et donc (indirectement) jusqu'à 1024^2 blocs appartenant au fichier. Cela représente 4 Go. L'usage de blocs d'indirections triples porte la limite des tailles de fichiers à 4 To. Cette limite suffisant à la plupart des usages, l'usage de blocs d'indirection quadruple n'est pas prévu (au delà de 4 To, il restera possible de doubler la taille des blocs...).

Ce principe utilisant des indirections d'ordres différents permet une recherche rapide des numéros de blocs associés à de petits fichiers, sans limiter la taille des fichiers supportés.

Le système de fichiers MS-DOS

Le système de fichiers MS-DOS place la description des fichiers (l'équivalent d'un *i-node*) directement dans un répertoire. Par ailleurs, nous ne trouvons que le numéro du premier bloc du fichier. Pour chercher le bloc suivant, il faut consulter une table, la `FAT` (*File Allocation Table*), associant à chaque numéro de bloc d'un fichier, le numéro du bloc suivant. Cette méthode, simple et adaptée à la lecture séquentielle d'un fichier nécessite, pour l'accès à un bloc quelconque, un nombre d'opérations proportionnel à la taille du fichier (au lieu de 3 lectures pour un système de fichiers UFS). Cela rend ce système de fichiers inadapté aux bases de données dont les accès aléatoires sont fréquents.

Le système de fichiers MS-DOS a vu plusieurs évolutions, portant la taille des numéros de blocs de 12 bits à 16, puis 32, et permettant l'enregistrement de noms de fichiers de plus de 11 caractères (en pratique, pour la compatibilité ascendante, un fichier est associé à un nom limité à 11 caractères et des entrées spéciales fournissent des brides du nom long).

Le système de fichiers `HFS` de Mac OS

Comme son nom l'indique, *Hierarchical File System*, le système de fichiers de Mac OS ajoute au précédent la possibilité de ranger des répertoires à l'intérieur d'autres répertoires. Cependant, ce système de fichiers se distingue par l'usage récurent de structures *B-tree* héritées des index de base de données. Ces structures permettent la création de tables associant une structure à une clef d'accès. Comme pour les blocs d'indirection sous Unix, les temps d'accès à cette structure sont proportionnels au logarithme du nombre de structures enregistrées, mais un arbre *B-tree* peut être utilisé avec n'importe quel type de clef. En comparaison, les blocs d'indirection ne peuvent être utilisés que pour des suites d'entiers consécutifs (des trous peuvent être toutefois tolérés).

Par ailleurs, les méta-données de ce système de fichiers sont placées eux-mêmes dans des fichiers. Cela peut simplifier l'accroissement de l'espace alloué à ces méta-données. En comparaison, la table des *i-node* sous Unix est allouée à la création du système de fichiers. Cela limite d'emblée le nombre maximum de fichiers.

Comme sous Unix, un fichier est accessible par le système d'exploitation à l'aide de son numéro. En revanche l'interface de programmation rend possible de tels accès aux programmes. Cela est utilisé par les applications qui souhaitent accéder à leurs fichiers même après un déplacement ou un changement de nom.

Pour trouver l'emplacement d'un bloc d'un fichier sur le disque, un arbre appelée *extents B-tree* est utilisé avec le numéro de fichier et le numéro du bloc comme clef (ajoutons aussi un entier qui indique quelle moitié de fichier (*fork*) est considérée). Contrairement au système de fichiers Unix, les blocs ne sont pas pointés individuellement, mais par suites de blocs consécutifs appelées *extents*. Chaque *extent* peut être caracterisés par le numéro du premier bloc et le nombre de blocs. Lorsque qu'un fichier peut être placé dans des blocs consécutifs, l'usage d'*extents* permet un gain de place et limite le nombre de lectures du fichier *extents B-tree*.

En ce qui concerne les répertoires, ces derniers ne sont pas stockés sous forme de fichiers associées chacun à un répertoire. Au contraire, un unique fichier, *catalog B-tree*, associe à une paire (numéro de répertoire, nom du fichier ou du répertoire) une structure décrivant le fichier ou le répertoire. La description d'un fichier fournit diverses informations telles que le type, l'application qui l'a créé, les informations nécessaires au tracé de sa représentation graphique (icône, position), taille, etc. De plus, cette structure inclut la position des premiers *extents* afin d'éviter la lecture de du fichier *extents B-tree* pour des petits fichiers.

Par ailleurs, un fichier spécial appelé *Master Directory Block* situé à une place fixe fournit les informations importantes du système de fichier. En particulier, la position des premiers blocs de l'*extents B-tree*... les autres blocs sont situés à l'aide de ce même fichier ! Des précautions lors de l'accroissement de ce fichiers sont nécessaires pour éviter que la recherche d'un *extent* ne dépende directement ou non du résultat de cette même recherche.

Le nouveau système : ZFS

Ce système de fichiers est apparu en 2005 lors de son intégration au système d'exploitation Solaris de Sun [10] Il s'agit d'un système de fichiers 128 bits, ce qui signifie que chaque pointeur de bloc est codé sur 128 bits ce qui permet d'atteindre des tailles de fichiers très importantes (16 exbioctets sachant que l'exbioctet vaut 2^{60} octets soit pas moins d'un million de téraoctets !).

ZFS intègre les notions de gestionnaire de volumes (voir la section suivante) et construit un ensemble de stockage [11] de manière hiérarchique en intégrant des volumes physiques dans un premier temps, puis les volumes virtuels. Chaque feuille de l'arbre décrivant ces volumes virtuels contient des informations relatives à tous les parents dont elle descend comme le montre la figure 7.2

10. Sun a depuis été racheté par Oracle.
11. *ZFS pool* ou encore `zpool`.

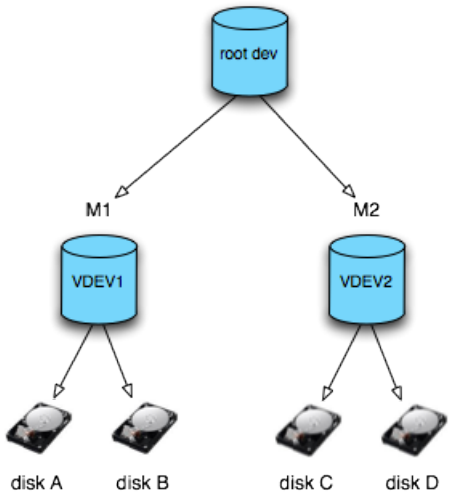


FIGURE 7.2 – *Les périphériques physiques, appelés feuilles, sont regroupés au sein de volumes virtuels, lesquels peuvent être aussi regroupés. Le* `VDEV1` *est un volume virtuel dans lequel toute écriture est réalisée en « miroir » (RAID1) sur les disques physiques A et B. Il en est de même pour le volume virtuel* `VDEV2`*. Le volume racine, noté symboliquement* `root dev`*, est une agrégation des deux volumes virtuels (RAID0). Le volume « réel » ou physique* `C` *contient dans sa description des informations sur le disque* `D` *mais aussi sur le volume* `VDEV2`*. En lisant ces informations, le système d'exploitation peut ainsi connaître l'organisation des différents éléments, même si l'un d'entre eux fait défaut.*

Ayant à disposition des volumes de stockage, la création d'un système de fichiers à l'intérieur est rendu, avec ZFS, particulièrement simple et ressemble, à s'y méprendre, à la création d'un répertoire. Chaque système de fichiers créé est automatiquement monté (à moins qu'une option l'interdise et qu'il faille recourir dans ce cas à la traditionnelle commande `mount`). Chaque utilisateur d'un système d'information pourrait ainsi avoir sa propre « partition ».

ZFS permet la réalisation de *snapshot*, c'est-à-dire d'une photographie du système de fichiers placée en lecture seule. Cette prise de vue ne doit pas être confondue avec une simple copie à l'identique puisque à l'instant de sa réalisation, une *snapshot* ne consomme pratiquement aucune place, tout est affaire d'indirection dans les blocs de données. Lorsqu'un bloc de données change (l'utilisateur modifie un fichier par exemple), le bloc original sera conservé pour la *snapshot* tandis qu'un nouveau bloc sera mis à disposition de l'écriture accueillant les changements. Ce mécanisme de « *Copy On Write* », déjà remarqué pour la gestion de la mémoire virtuelle, est particulièrement

efficace et peu gourmand en occupation du disque.

ZFS introduit aussi la dé-duplication des blocs de données. Deux fichiers différents peuvent très bien posséder des zones de données parfaitement identiques. La déduplication permet de faire pointer des blocs issus de deux inodes différentes vers la même zone du disque (tant que ces blocs sont communs naturellement).

Enfin, un mécanisme de compression à l'écriture (utilisant une librairie très répandue, la `libz`) permet un gain de place assez conséquent. Sur un système hébergeant un logiciel de gestion des messages électroniques, chaque utilisateur peut avoir une « partition », c'est-à-dire un système de fichiers pour lui. Les messages comportent très souvent du texte et leur compression peut entraîner un gain de l'ordre de 80 %. Associé au mécanisme de *snapshot* précédemment cité, la restauration des messages, effacés par mégarde, devient très rapide.

On assiste hélas en ce moment à une véritable guerre entre NetApp et Oracle, le premier soutenant que les idées et les principes sur lesquels reposent ZFS ayant fait l'objet de brevet, le second soutenant qu'il s'agit d'une création originale. L'intégration de ZFS dans BSD est déjà réalisée, mais sa mise à disposition dans le noyau Linux n'est pas encore d'actualité. Notons enfin que, annoncée dans Mac OS 10.6, ZFS n'est pas encore disponible nativement pour les utilisateurs de la Pomme.

7.3 La gestion des volumes

Pour des raisons pratiques, il peut-être utile de placer sur un même disque, plusieurs systèmes de fichiers indépendant. C'est en particulier le cas pour faciliter les sauvegardes ou lors de l'usage de plusieurs systèmes d'exploitation. Pour cela, des structures doivent être utilisées pour mémoriser les espaces alloués aux différents systèmes de fichiers.

Les tables de partition

La structure la plus simple pour décrire un espace alloué sur le disque est la table des partitions [12]. Une telle table associe à une partition identifiée par un numéro un ensemble de blocs [13] consécutifs situés à l'aide du numéro du premier bloc et du nombre de blocs.

Sur les architectures de type Intel on trouve sur le premier secteur du disque une table de partitions principale qui en plus des informations sur les différents secteurs alloués aux partitions contient également le programme d'amorçage (appelée pour cette architecture `MBR` ou « *Master Boot Record* »). Cette table de partitions ne pouvant contenir que 4 partitions, lorsque le besoin s'en fait sentir, la dernière partition est une

12. Sous MS-DOS, d'une conception initiale limitée, combinée à la compatibilité ascendante maintenue lors de chacune des évolutions, résultent des structures assez compliquées.

13. Ici, il s'agit des blocs d'un périphérique, généralement 512 octets pour un disque dur.

partition étendue (« *Extended Boot Record* ») et ne sera reconnue que par le système d'exploitation. C'est le BIOS qui a la charge de lire et d'interpréter cette table principale.

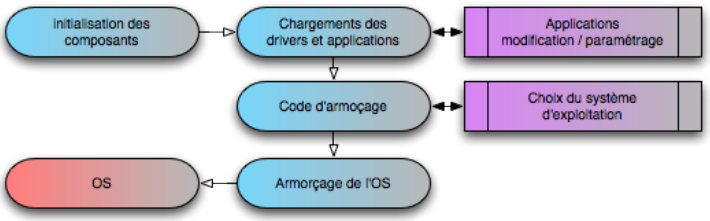


FIGURE 7.3 – *La séquence d'allumage d'un ordinateur utilisant l'UEFI. Dans un premier temps l'UEFI se charge de l'initialisation des différents périphériques présents sur la carte mère. Puis il charge les différentes applications qui permettent de configurer les composants, ce à la demande de l'utilisateur. Il démarre enfin son chargeur de démarrage qui va aller rechercher le chargeur de démarrage du système d'exploitation.*

Les limitations imposées par le BIOS et surtout sa dépendance aux extensions propriétaires ont ouvert le chemin à l'UEFI (« *Unified Extensible Firmware Interface* ») mais aussi à *Open Firmware*. Cette interface entre le système d'exploitation et le matériel permet l'utilisation d'un nouveau modèle de table de partitions, le format GPT mis pour « *Globally Unique IDentifier Partition Table* ». Ce format permet d'accéder à un volume de disque plus important que celui géré par MBR. Le format GPT permet de plus d'associer un numéro unique à chaque partition. Peu importe alors l'ordre dans lequel les périphériques sont vus par le système d'exploitation, une partition sera toujours affectée à son point de montage correct grâce à cet identifiant unique. Vous avez peut-être déjà remarqué cela lors des séquences de boot d'un ordinateur utilisant Linux (et éventuellement GPT) ainsi que dans le fichier décrivant les point de montage des partitions :

```
UUID=2532b597-dfa5-4766-8391-405b7126f8e8 /      ext3 0 1
UUID=8a0ada47-375f-4ab7-b01e-b86b1e00a280 /home  ext3 0 1
UUID=218fe85e-94bb-4e97-8386-0eaa1fbdb50b /cours ext3 0 1
UUID=5af07829-57de-4dad-9853-b24a7ac0b6d0 /opt   ext3 0 1
UUID=4dd9b92a-1842-432f-92df-c3760c3cb998 /part  ext3 0 1
UUID=61d0ff68-8438-4418-96a1-2fba91314abb none   swap 0 0
```

L'utilisation de GUID pour marquer les partitions n'est pas lié à une table de partitions GPT, mais par contre GPT utilise GUID pour identifier les partitions et les disques.

Quelques remarques sur l'UEFI. Il s'agit d'une interface entre le matériel (les « *firmwares* ») et le système d'exploitation. Une architecture conçue pour exploiter l'UEFI permet un dialogue beaucoup plus agréable entre l'utilisateur et les composants matériels. En plus d'être utilisé au démarrage, l'UEFI est aussi une interface de communication entre le système d'exploitation et le matériel pour certains aspects.

Enfin l'`UEFI`, comme on peut le remarquer dans la figure 7.3, permet de se départir des chargeurs de démarrage des systèmes d'exploitations (tel que `GRUB` ou `Lilo`). L'`UEFI` a la capacité d'identifier une partition `EFI` à l'intérieur de laquelle il pourra trouver un chargeur de démarrage de second niveau (par exemple `elilo.efi`) qui permettra de charger un système d'exploitation avec des options passées en ligne de commandes.

Une peur secoue actuellement la communauté libre. Depuis une version récente (2.3.1), l'`UEFI` permet de protéger les systèmes d'exploitation amorçables en vérifiant leur signature. La signature d'un programme est obtenue en chiffrant à l'aide d'une clé privée (vendue 100$ par Microsoft) la somme `MD5` (ou son équivalent). L'`UEFI` compare alors cette signature en la déchiffrant à l'aide la clé publique à la signature obtenue en calculant la somme `MD5` du programme. Si la fcontionnalité de *Secure Boot* n'est pas désactivable, un système d'exploitation non signé ne peut pas être chargé. La mise en place de cette infrastructure semble sécuriser un ordinateur, mais elle peut aussi servir à contrôler ce que chaque particulier a le droit d'utiliser sur son ordinateur...

Si, dans le cas des architectures classiques, cette gestion de l'espace disque suffit pour des besoins limités où des systèmes de fichiers sont alloués une fois pour toutes, des mécanismes plus complexes appelés gestionnaires de volumes sont utilisés dans les plus gros systèmes (c'est le cas avec `ZFS` et les `zpools`). Ces gestionnaires sont comparables à des systèmes de fichiers simplifiés. Cependant l'allocation de volumes étant plus rare et portant sur des ensembles importants de blocs, les méta-données sont relativement peu nombreuses et sont chargées en mémoire au démarrage.

Les volumes

Un gestionnaire de volumes est généralement capable de créer un volume à l'aide de plusieurs ensembles de blocs consécutifs, éventuellement situés sur des disques différents. Par ailleurs certains gestionnaire sont associés à une organisation redondante des données permettant de garantir l'accès à un volume après une panne d'un disque dur. Ces organisations sont appelée « en miroir » ou RAID 1 lorsque les données sont dupliquées sur des disques différents. D'autres mécanismes appelées RAID 2 à RAID 5 ajoutent aux données situées dans plusieurs disques leur bits de parité sur un disque supplémentaire. En cas de panne d'un disque, chaque bit de ce disque pourra être calculé à partir des autres disques.

Dans certains cas, le gestionnaire de volumes est couplé aux systèmes de fichiers, ce qui permet l'accroissement d'un système de fichier. Ce couplage est nécessaire puisqu'un tel accroissement doit modifier les méta-données du système du fichiers afin de déclarer les blocs ajoutés. Une autre fonction disponible avec certains gestionnaire de volumes permet une copie logique instantanée d'un volume, ce qui peut être utile en particulier pour des sauvegardes de fichiers ouverts.

Bien que les partitions ou volumes sont souvent utilisés pour y placer des systèmes de fichiers, d'autres usages sont aussi fréquents pour d'autres structures de tailles importantes et allouées rarement (généralement une fois, à l'installation). Cette allocation,

moins flexible qu'avec un système de fichiers permet de meilleures performances [14]. Les usages les plus courants sont :

– l'espace utilisé par la gestion de la mémoire pour la pagination ;
– les bases de données.

7.4 Améliorations des systèmes de fichiers

Le cache

Afin d'accélérer les accès aux disques, un mécanisme de cache est généralement mis en place. Ce cache permet, en maintenant en mémoire les blocs d'un fichier lus, d'éviter lors de leurs relectures d'autres accès au disque dur.

Par ailleurs, les écritures peuvent être différées afin de regrouper les écritures sur des blocs voisins et limiter le nombre d'accès au disque dur en cas d'écritures successives sur un même bloc.

La gestion du cache peut avoir des conséquences importantes sur les performance d'un système. Une optimisation, « *read ahead* » consiste à anticiper la lecture de blocs consécutifs lors de la lecture d'un fichier. Ces blocs seront fournis immédiatement aux programmes lorsque nécessaire. Cette optimisation profite des caractéristiques des disques durs dont les débits sont importants lors de la lecture de blocs consécutifs, mais dont le temps d'accès (temps de positionnement d'une tête de lecture sur le premier bloc) reste relativement grand [15].

Si le cache accélère les accès en lecture et écriture, il peut avoir des conséquences sur l'intégrité des données. En effet, l'appel système `write()`, au lieu de modifier un fichier ne fait que planifier cette modification. Pour des applications où la sécurité des données est importante, un appel système (`fsync()` sous Unix) permet la demande d'écriture effective des blocs modifiés au préalable. Cela est en particulier utilisé par le serveur de messagerie `sendmail` pour enregistrer réellement un courrier électronique avant d'indiquer sa prise en compte au serveur qui l'envoie.

Un aspect important de la gestion du cache est l'adaptation de la quantité de mémoire occupée par le cache. Cette dernière doit être ajustée en fonction de la mémoire disponible : dans l'idéal, le cache utilise toute la mémoire disponible... et se réduit lorsque des applications demandent de la mémoire. Sur les systèmes d'exploitation moderne, la gestion du cache est intimement liée à la gestion de la mémoire.

14. Dans certains systèmes temps réels, le système de fichiers permet la création de fichiers contigus, ce qui permet la flexibilité et des performances garanties. Cependant, l'appel système utilisé pour cette création peut échouer lorsque l'opération n'est pas possible.

15. Cela est malheureusement de plus en plus vrai : les débits augmentent proportionnellement à la densité des données et à la vitesse de rotation alors que les temps d'accès (environ 10 ms) s'améliorent très lentement.

La fragmentation

Des accroissements successifs d'un fichier sont susceptibles d'allouer des blocs dispersés sur la surface d'un disque dur. Cette dispersion appelée fragmentation (le fichier est découpé en fragments, séries de blocs consécutifs), rend nécessaire, pour la lecture du fichier, des mouvements des têtes de lecture, ce qui ralentit les accès. Par opposition, une allocation de blocs consécutifs, permet la lecture d'un fichier de taille importante à l'aide d'un nombre limité de déplacements des têtes de lecture. Notons que les progrès les plus importants relatifs aux disques durs portent sur la densité des données et sur le débit des lectures et écritures de blocs consécutifs. En revanche les temps d'accès moyens (principalement dûs au déplacement de la tête de lecture et à la vitesse de rotation) progressent très lentement, ce qui rend la baisse relative de performance due à la fragmentation de plus en plus importante.

L'une des techniques les plus simples pour limiter la fragmentation consiste à augmenter la taille des blocs. Cependant, cette technique rend inutilisable l'espace situé entre la fin d'un fichier et la limite du dernier bloc occupé par ce fichier. Cette perte d'espace est d'autant plus importante que la taille moyenne de fichiers est petite.

Le système de fichiers FFS (*Fast File System*) très utilisé sous Unix s'inspire de cette technique sans en avoir les inconvénients. Les blocs (typiquement 8 Ko ou 16 Ko) sont divisés en fragments (par exemple 1 Ko). Des fichiers différents peuvent se voir allouer les fragments d'un même bloc, ce qui permet une économie de place comparable à un système de fichiers utilisant des petits blocs. Par ailleurs lors de l'accroissement d'un fichier, les fragments sont alloués dans un même bloc jusqu'au remplissage du bloc. Ces fragments peuvent même être déplacés pour permettre de continuer l'allocation à l'intérieur d'un même bloc. Ce principe d'allocation à deux niveaux permet de cumuler les avantages des petits blocs (perte moindre d'espace disque) et de gros blocs (fragmentation limitée des fichiers). Évidemment, l'allocation d'un nouveau bloc favorise les blocs les plus proches du bloc précédent.

Le système de fichiers *ext3* privilégié sous Linux utilise un mécanisme plus simple : des petits blocs sont utilisés, mais l'accroissement d'un fichier est toujours réalisé en allouant plus de blocs que nécessaires (par défaut 8)... lorsque de tels blocs sont consécutifs. Cela favorise l'allocation de blocs consécutifs. Afin d'éviter un gaspillage d'espace, les blocs inutilisés sont libérés à la fermeture du fichier.

Une autre technique utilisée (notamment dans le système de fichiers FFS) pour limiter l'impact de la fragmentation consiste à diviser un système de fichiers en groupes de cylindres (appelés groupes de blocs sous Linux). À chaque répertoire et ses fichiers est associée une partie du disque dans laquelle auront lieu préférentiellement toutes les allocations. Le choix d'un nouvel espace disque privilégie un usage équilibré de chaque groupe de cylindres. Ainsi, lorsqu'un fichier doit être agrandi, cette répartition assure l'existence d'un bloc libre non loin du dernier bloc de ce fichier. De plus, une application qui manipule des fichiers situés dans un même répertoire (cela est fréquent) provoquera des déplacements de la tête de lecture de petites amplitudes, puisqu'à l'intérieur d'un même groupe de cylindre.

En revanche, les systèmes Microsoft sont souvent dotés de logiciels de défragmentation. Le principe consiste à rassembler après coup les fragments de fichiers afin d'augmenter les performances. De plus, les fichiers sont souvent rassemblés au début du disque, ce qui permet d'obtenir un espace libre non fragmenté important. En revanche, avec un système de fichiers ainsi « tassé », l'accroissement d'un fichier risque d'allouer un bloc assez loin du début de fichier... défaut qu'il faudra corriger à nouveau. Si ce mécanisme est simple, il nécessite un système assez disponible, puisque la défragmentation implique beaucoup d'accès au disque. Cela est généralement le cas pour des serveurs peu sollicités en dehors des heures de travail. Par ailleurs, pour des systèmes ne contenant que des fichiers en lecture seule, ce mécanisme est le plus efficace.

La protection contre la corruption

Un arrêt brutal du système (plantage, panne, défaut d'alimentation) risque d'entraîner la corruption du système de fichiers, le disque dur n'étant pas capable d'assurer l'écriture de plusieurs blocs simultanément. Par exemple, l'accroissement d'un fichier comporte plusieurs opérations :

- l'allocation d'un bloc ;
- l'allocation éventuelle de blocs d'indirection ;
- l'ajout de ce bloc au fichier (modification de numéros de blocs) ;
- l'écriture dans le fichier ;
- le changement de l'*i-node* afin d'enregistrer le changement de taille ;
- le changement du superbloc afin de renseigner le nombre de blocs disponibles.

Dans le cas d'un arrêt brutal du système entre deux écritures, les structures listées ci-dessus ne seront pas cohérentes, ce qui s'appelle une corruption du système de fichiers. Cela peut provoquer des pertes d'espace disque (des blocs sont alloués, mais n'ont pas été affectés à un fichier), ou pire encore un comportement incohérent du système de fichiers (c'est le cas si un bloc est affecté à un fichier mais marqué comme disponible : il risque d'être affecté à un autre fichier).

Pour limiter les dégâts, le système de fichiers FFS utilise des séquences d'écritures telles qu'un arrêt brutal laisse le système de fichiers dans un état sain : les seules conséquences d'une corruption sont la perte d'espace disque. Afin d'éviter un changement de cet ordonnancement par le cache, une écriture effective est attendue avant toute écriture suivante... l'impact sur les performances est tel que le choix est laissé à l'administrateur qui peut monter le système de fichiers en mode synchrone si l'on souhaite privilégier la sécurité (option par défaut sur les systèmes BSD) ou asynchrone si l'on souhaite privilégier les performances (option par défaut sur les systèmes Linux). Notons qu'une évolution récente des systèmes BSD, *soft updates*, modifie le cache et l'empêche de changer un ordonnancement calculé pour sécuriser le système de fichier... tout en permettant un fonctionnement asynchrone.

Malgré un ordonnancement adéquat des écritures, la récupération d'un système de fichiers dans un état normal nécessite une réparation, opération susceptible d'être

d'autant plus longue que le système de fichiers est grand : cette réparation consiste entre autres à faire l'inventaire des blocs affectés à un fichier et le comparer à la liste des blocs marqués comme non disponibles. En cas de divergence, une correction évidente s'impose. D'autres défauts sont aussi détectés tels que l'affectation d'un même bloc à deux fichiers.

Afin d'éviter de trop longues réparations, certains systèmes de fichiers (NTFS de Microsoft, JFS d'IBM, XFS de Silicon Graphics, VxFS de Veritas, ReiserFS et Ext3 sous Linux [16]) utilisent un mécanisme hérité des bases de données : la journalisation. Ce principe consiste à décrire dans un journal toute opération entreprise avant l'exécution proprement dite. Cette opération est ensuite réalisée. En cas d'arrêt brutal, le journal est lu, et les opérations décrites sont rejouées. Bien entendu, la journalisation nécessite des accès supplémentaires qui impactent les performances des systèmes de fichiers dont les méta-données sont très sollicitées.

Il convient toutefois de faire attention, les systèmes de fichiers journalisés protègent contre des corruptions portant sur les méta-données du système de fichier, et non le contenu des fichiers mêmes. Une application qui doit se prémunir contre une corruption de ses fichiers en cas d'arrêt brutal devra prendre ses dispositions pour sécuriser ses données, ce qui nécessite souvent un mécanisme de journalisation indépendant. C'est en particulier le cas des bases de données.

Signalons le cas du système de fichiers LFS introduit dans 4.4BSD. Ce système de fichiers est conçu comme un journal où les écritures ont lieu dans des blocs successifs. Le pointeur d'écriture revient au début une fois arrivé à la fin du disque. Au démarrage, il suffit de se positionner sur le dernier ensemble d'écritures formant un ensemble cohérent pour obtenir un état normal. La modification d'un bloc de données nécessite donc au préalable son déplacement à la position d'écriture courante, ce déplacement nécessite la modification d'un bloc d'indirection ou d'une *i-node*, opérations provoquant à leur tour d'autres déplacements. L'effet de ces déplacements en chaîne est compensé par la localisation des écritures. Afin que la position courante pointe toujours sur des blocs libres, un démon tourne en tâche de fond et déplace les blocs situés devant la position courante à la position courante.

Notons que ce système de fichiers — encore expérimental — risque de fragmenter les fichiers lors d'accès aléatoires en écriture (ce qui peut ralentir des accès en lecture sur des bases de données), mais au contraire défragmenter des fichiers lors d'accès séquentiels en écriture. L'optimisation des lectures n'était pas le but recherché, ces dernières doivent être limitées par le cache.

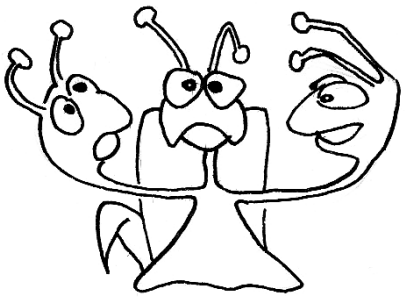
Malgré toutes ces précautions, il existe une erreur contre laquelle la plupart des systèmes de fichiers, en dehors de `ZFS`, ne savent pas se prémunir : la faute silencieuse d'écriture. En temps normal, l'utilisation de l'appel système `write()` et de `fsync()` établit un dialogue entre le système d'exploitation et le contrôleur du disque dur. Ce contrôleur, à l'issue de l'écriture effective sur le disque, répond à le système d'exploitation pour signifier si l'écriture a pu être faite ou non. Il existe toutefois un cas

16. Notons que les systèmes de fichiers JFS et XFS sont aussi portés sous Linux.

de figure impossible à détecter par des moyens matériels, le cas d'une écriture erronée au bon endroit. Le matériel répond en effet qu'il a réussi à écrire, mais le bloc qui vient d'être écrit ne correspond pas à ce qui aurait dû être écrit. ZFS se prémunit contre ces fautes silencieuses en associant à chaque bloc un contrôle d'erreur (une sorte de signature). Cela rajoute des écritures lors de chaque accès au périphérique, mais cela garantit l'intégrité des données.

8

Les architectures multi-processeurs et les *threads*



Nous allons volontairement mettre en parallèle dans ce chapitre les notions d'architectures multi-processeurs et celles de *threads* (que l'on traduit souvent par « fils d'activité »). Si l'utilisation de plusieurs processeurs a existé avant l'utilisation des *threads*, ce concept et cette architecture matérielle vont maintenant de pair. Mais nous allons voir que la notion de *thread* n'implique pas celle de multi-cœurs ou multi-processeurs, et que la possession d'une architecture multi-cœurs ne signifie pas obligatoirement l'utilisation de programmes conçus autour des *threads*.

8.1 De la loi de Moore au multi-cœurs

Gagner en puissance de calcul

Le phénomène *Andy giveth, and Bill taketh away* est une façon de dire que peu importe la rapidité et la puissance des processeurs, de nouvelles applications arriveront toujours à demander et consommer davantage de puissance de calcul. S'il y a 20 ans, la possibilité de disposer d'une puissance formidable de deux millions d'opérations par seconde était une chance pour un doctorant, aujourd'hui, le moindre ordinateur de bureau, et le système d'exploitation qui va avec seraient incapables de fonctionner

correctement avec aussi peu d'opérations disponibles. Dans cette course à la puissance, il est difficile de savoir si ce n'est pas l'arrivée de nouveaux processeurs plus rapides qui crée le besoin applicatif plutôt que l'inverse.

Au cours des trente dernières années, les concepteurs de CPU ont réalisé des gains en performance en agissant sur trois domaines :

– la vitesse d'horloge ;
– l'optimisation de l'exécution ;
– le cache.

L'augmentation de la vitesse d'horloge permet simplement de faire les choses plus rapidement. Cependant on se heurte assez vite à une barrière physique, celle de la propagation des ondes électromagnétiques à l'intérieur des circuits électroniques. Augmenter l'horloge c'est devoir impérativement diminuer la taille du circuit, donc probablement limiter le nombre de transistors présents et donc diminuer la puissance ! Certes, les technologies de gravures actuelles n'ont pas forcément atteint leur maximum et les processeurs d'hier, gravés à 95 nanomètres, font figure de dinosaures face aux gravures à 35 nanomètres.

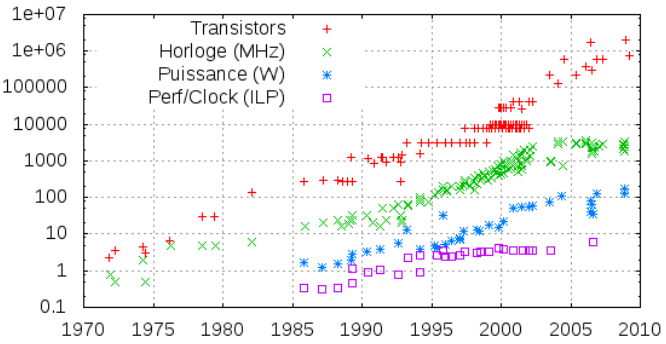


FIGURE 8.1 – *Si le nombre de transistors continue d'augmenter, on remarque que la fréquence d'horloge et la puissance consommée stagnent. La performance ramenée à la vitesse de l'horloge stagne elle aussi.*

Augmenter la taille du cache présent à côté du processeur permet de rester éloigné de la mémoire qui continue à être beaucoup moins rapide que les processeurs. Cela augmente naturellement les performances à condition toutefois de savoir gérer à la fois la cohérence de cache (nous en reparlons plus loin) et le fait de placer les bonnes données dans le cache.

L'optimisation de l'exécution est une chose beaucoup plus intéressante et qui passionne nombre de chercheurs. Il s'agit en effet de réaliser l'exécution de plus de choses pendant le même nombre de cycles. Cela implique de travailler sur le *pipelining*,

le réordonnancement des instructions et la possibilité d'exécuter plusieurs instructions de manière parallèle au cours du même cycle d'horloge.

Si l'utilisation d'un seul micro-processeur ne suffit pas, pourquoi ne pas en utiliser plusieurs ? Pourquoi ne pas placer plusieurs processeurs au sein du même circuit ? Il existe une différence fondamentale entre l'utilisation de plusieurs processeurs sur la même carte mère et l'utilisation de plusieurs cœurs au sein d'un même circuit.

Nous allons aborder ces deux architectures en examinant les modifications qu'elles imposent au système d'exploitation et notamment aux concepts que nous avons déjà évoqués, à savoir la gestion des processus et la gestion de la mémoire. Mais commençons tout d'abord par bien apprendre à différencier les deux types d'architectures.

Cluster versus `SMP` *versus* multi-cœurs

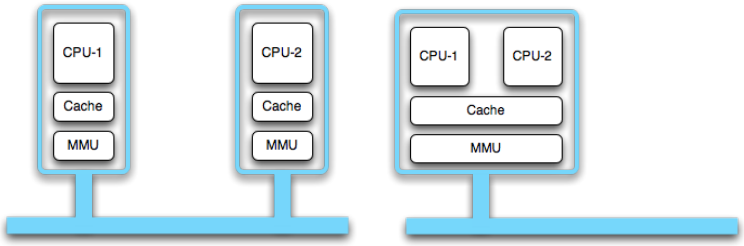


FIGURE 8.2 – *Organisation du cache et de la MMU dans une architecture* Symmetric Multi-Processing *(ou* `SMP`*) (à gauche) et dans une architecture double cœur (à droite).*

Un processeur est avant tout une unité d'exécution d'instructions, et malgré les progrès évoqués plus haut dans l'organisation du traitement des suites d'instructions (réordonnancement ou *pipelining*), une unité d'exécution (un CPU en fait) est avant tout sériel.

La première réponse qu'il était possible d'apporter à la demande croissante de puissance de calcul fut l'emploi de plusieurs machines interconnectées par des liens réseaux à haut débit ainsi qu'un système d'exploitation adéquat permettant de répartir les processus sur les différentes machines : le parallélisme faiblement couplé (*loosely-coupled multiprocessing*). L'exemple typique de ce parallélisme faiblement couplé est le *cluster Beowulf*. Il s'agit d'une architecture multi-ordinateurs comportant un nœud central ou nœud serveur et plusieurs nœuds clients qui se trouvent connectés les uns aux autres par un lien réseau (Ethernet, *Fiber Channel*...). Le serveur et les clients utilisent un système d'exploitation Linux ainsi qu'une bibliothèque de contrôle et de communication telle que `PVM` (*Parallel Virtual Machine*) ou `MPI` (*Message Passing Interface*). Le serveur central se comporte vraiment comme une machine unique ayant à disposition des unités de calcul (les ordinateurs ou nœuds clients). Nous pouvons

aussi citer *Amoeba*, un système à base de micro-noyaux qui permet de transformer un ensemble de stations (ou de serveurs) en un système distribué.

La deuxième réponse apportée fut l'utilisation de plusieurs processeurs ou *Symmetric Multi-Processing*. Mais comme le montre la figure 8.2, il ne s'agissait pas pour les constructeurs de re-développer un processeur, mais bel et bien de faire coexister sur une même carte mère plusieurs processeurs identiques. Cela signifiait que chaque processeur possédait son cache et son gestionnaire de mémoire virtuelle. Nous avions presque deux ordinateurs à disposition. Tant mieux ! En effet un processus pouvant accéder à l'un des deux CPU devait attendre que le système d'exploitation charge le `CR3` (voir, ou revoir 6.1) pour accéder aux catalogues et aux pages mémoires le concernant et donc réalise toutes les opérations de changement de contexte. Deux processus différents pouvaient donc s'exécuter en même temps ce qui devait, théoriquement, augmenter la puissance par deux... Mais accéder à la mémoire physique de manière parallèle au travers de deux MMU peut avoir des conséquences dramatiques si cela n'est pas correctement géré. Nous savons en effet que les données sont avant tout transférées de la mémoire centrale vers les différents caches mémoires. Dans une architecture multi-processeurs, chaque processeur possède des caches qui lui sont propres. Il est impératif de gérer une cohérence entre les caches des différents processeurs afin d'avoir une vision cohérente de la mémoire. Dans une architecture SMP, cette cohérence de cache passe obligatoirement par la circuiterie de la carte mère dont les bus de communication ne permettent pas d'obtenir la même rapidité que les bus internes du processeur [1]. Le gain en performance s'en trouve grevé.

Dans une architecture multi-cœurs, nous allons retrouver sur le même substrat deux (ou plus) unités de calcul. La figure 8.3 présente une vue du circuit imprimé de l'Intel Conroe.

Ces unités de calcul vont pouvoir dialoguer entre elles sans sortir de la puce. On gagne donc en performance pour maintenir la cohérence de cache [2], on gagne aussi en homogénéité puisque la gravure des deux processeurs est effectuée en même temps donc dans les mêmes conditions, et enfin on gagne en circuiterie sur la carte mère chargée d'accueillir une seule puce.

Nous détaillerons par la suite une solution utilisant le multi-cœurs mais rendant les accès à la mémoire asymétriques.

Le rôle du système d'exploitation

Jusqu'à présent notre système d'exploitation maintenait une vision de la mémoire par processus s'exécutant sur le CPU (au travers des catalogue de pages, du TLB

1. Nous avons omis de parler de certains types de réseaux d'interconnexion tels que les réseaux *Crossbar*. Cette architecture permet la connexion de différents processeurs à différentes sources de mémoire de manière très rapide et surtout uniforme, *i.e.*, tous les processeurs sont à « égale distance » de toutes les sources de mémoire. Ces réseaux d'interconnexion peuvent accueillir de nombreux processeurs mais sont bien plus onéreux que les bus classiques.

2. Enfin presque ! Souvenons-nous de la remarque sur les processeurs double cœur Intel dans lesquels le cache mémoire de niveau deux se situait à l'extérieur de la puce...

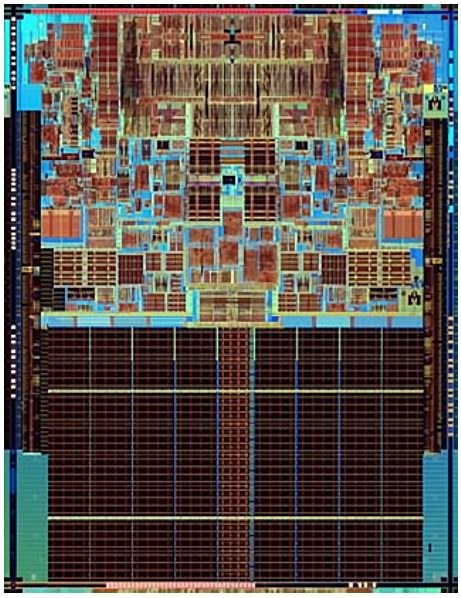


FIGURE 8.3 – *Vue d'ensemble du substrat de l'architecture multi-cœurs d'un processeur Intel Conroe (source wikimedia). On distingue clairement en haut à gauche et à droite les deux CPU, le bas du circuit étant occupé par la mémoire cache et les bus de communication.*

et grâce à la MMU). Lorsque plusieurs processeurs sont disponibles, et donc que plusieurs processus peuvent s'exécuter en même temps, le système d'exploitation doit être modifié pour gérer cette situation nouvelle :

- il faut maintenir une cohérence de cache comme cela a déjà été mentionné ;
- il semble impératif de synchroniser les accès à la mémoire ;
- il faut gérer les accès concurrents à la mémoire.

Nous aurons l'occasion de revenir dans la partie traitant des *threads* sur la cohérence de cache et les accès concurrents. Examinons tout d'abord la synchronisation des accès à la mémoire physique. L'approche traditionnelle est celle du « *Symmetric Multi-Processing* » dans laquelle chaque processeur possède son gestionnaire de mémoire, l'attribution des pages mémoires dans la mémoire physique étant globalement, et pour chaque MMU, gérée par le système d'exploitation. On voit donc qu'outre la cohérence de cache (ceux de niveau 2 ou 3 à l'intérieur d'un substrat ou d'une puce) il faut s'occuper de la cohérence des différentes tables de pages mémoire. La figure 8.4 donne une idée schématique de l'organisation de la gestion de la mémoire. Le système d'exploitation réside dans une zone mémoire et s'occupe d'organiser le découpage de la mémoire pour l'attribuer aux différents processus. Hormis le fait de contrôler et

de mettre à jour plusieurs catalogues de pages et TLB lorsque qu'interviennent des changements dans l'organisation de la mémoire physique, les mécanismes d'accès à la mémoire sont sensiblement identiques à quelques détails (très importants !) près.

Afin de faire dialoguer les processeurs entre eux, le système d'exploitation dispose d'un mécanisme d'interruptions : les `IPI` ou « *Inter Processor Interrupts* ». Lorsqu'une entrée d'un catalogue de pages change, chaque processeur doit impérativement rafraîchir son TLB ou marquer cette entrée comme invalide dans le catalogue des pages. Le processeur ayant provoqué ce changement sait comment faire cela de manière automatique, ce qui n'est pas le cas des autres processeurs. Ainsi, le processeur ayant provoqué le changement doit envoyer une `IPI` aux autres processeurs afin que ceux-ci réalisent les changements opportuns dans les catalogues de pages. Remarquons au passage qu'il faudra probablement interrompre le traitement des interruptions lorsque l'on traite une telle interruption.

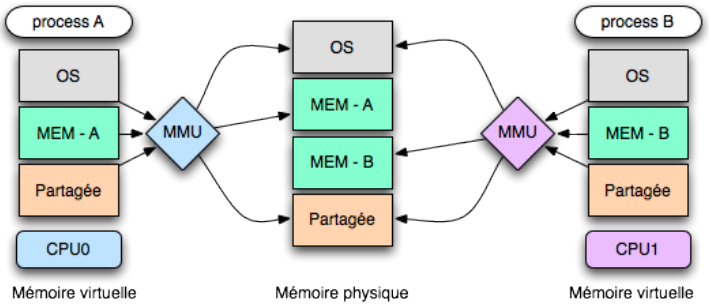


FIGURE 8.4 – *L'accès à la mémoire dans une architecture SMP.*

Pourtant le principal problème n'est pas lié au système d'exploitation lui-même mais bel et bien à l'accès physique à la mémoire au travers du bus mémoire et de son contrôleur. Il est en effet difficile de maintenir un accès correct à la mémoire en augmentant à la fois la fréquence d'horloge des processeurs ainsi que leur nombre ; la quantité de données devant circuler sur le bus de données devient alors très grande. Rapidement la limite haute du nombre envisageable de processeurs pour un accès correct à la mémoire fut atteinte : 8 processeurs saturent le NorthBridge. Si une réponse possible est l'utilisation d'un réseau d'interconnexion de type *Crossbar*, son coût reste prohibitif pour un faible nombre de processeurs et pour un usage de type ordinateur personnel.

Les fabricants de processeurs proposèrent alors de désymétriser l'accès à la mémoire, ce qui était en fait la mise en application, coté hardware, d'une idée déjà connue dans le parallélisme faiblement couplé. La mémoire fut donc découpée en banques, chacune de ces banques étant dévolue à un processeur. Ce type d'accès asymétrique est appelé `NUMA` pour « *Non Uniform Memory Access* » en opposition à l'architecture

précédente qualifiée de UMA. Afin de faire adopter cette technologie, il est impératif de la rendre aussi transparente que possible aux yeux d'un système d'exploitation. Un adressage global de la mémoire doit donc être possible, quand le système d'exploitation n'est pas conçu pour utiliser une architecture NUMA ce qui est réalisé par exemple dans la technologie HTT (« *HyperTransport Technology* ») d'AMD. Chaque processeur reconnaît la banque de mémoire qui lui est associée mais il la déclare aussi dans l'adressage global. Qui plus est la cohérence de cache est maintenue automatiquement par les processeurs eux-mêmes.

Pour tirer des bénéfices d'une architecture NUMA il est impératif de modifier le système d'exploitation afin que les processus soient exécutés sur un couple processeur / banque mémoire. L'optimum de performance est atteint lorsque toute la mémoire allouée par un processus s'exécutant sur un processeur est contenue dans la banque de mémoire de ce processeur, et encore mieux, si les *threads* issus de ce processus sont eux aussi exécutés sur le même processeur. Afin de réaliser ceci, l'ordonnanceur de tâches doit être changé.

Un ordonnanceur multi-queues

Dans l'approche traditionnelle que nous avons vue dans le chapitre 5 page 121, le système d'exploitation maintient une table de processus, avec différentes files de priorité certes, mais ceci est mal adapté à la gestion du parallélisme. Cette approche est connue sous Linux sous l'acronyme DSS ou « *Default SMP Scheduler* ». Lorsque qu'un processeur doit sélectionner un nouveau processus (il vient de rencontrer l'exécution de `sleep()` ou de `wait_for_IO()` ou il a reçu une interruption d'un autre processeur), le système d'exploitation doit accéder à la queue de processus et pour ce faire, puisqu'il est lui-même susceptible de s'exécuter en parallèle, il est impératif qu'il place un verrou (ces notions seront examinées plus en détail dans le cours sur les *threads*) afin de garantir un accès unique et sériel à cette structure. Des efforts sont toutefois consentis pour calculer la priorité des processus dans le cas d'une architecture SMP : une partie de la priorité dépend du surplus de tâches qui pourrait intervenir dans la cohérence de cache et dans le changement de la table des pages si un processus devait changer de processeur pour s'exécuter. Cette pénalité au changement de processeur est naturellement totalement dépendante de l'architecture matérielle sous-jacente. Le verrouillage est néanmoins un goulet d'étranglement pour le parallélisme (le fameux BKL ou « *Big Kernel Lock* »).

Le système d'exploitation doit donc évoluer et c'est ainsi qu'est apparu MQS ou « *Multiple Queue Scheduler* ». Au lieu d'une seule queue de processus, il existe autant de queues que de processeurs. Chaque queue possède ses verrous mais ces derniers sont locaux au processeur auquel est attachée la queue de processus. Le mécanisme d'ordonnancement intervient maintenant en deux étapes :

1. Acquisition du verrou local à la queue de processus du processeur ayant entraîné l'appel de l'ordonnanceur. La queue de processus du processeur est analysée afin

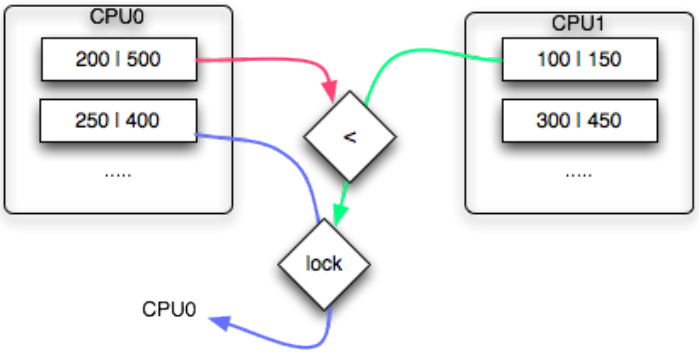


FIGURE 8.5 – *La queue locale au processeur* `CPU0` *permet de choisir le premier processus de priorité avec affinité* 200*. Il est comparé au premier processus distant du* `CPU1` *de priorité sans affinité* 150*. Si le système d'exploitation ne parvient pas à poser un verrou pour migrer ce processus distant dans la queue du processeur* `CPU0`*, c'est le deuxième processus du processeur* `CPU0` *qui sera élu.*

de déterminer le processus possédant la meilleure priorité d'exécution ainsi que celui venant immédiatement après dans ce classement.

2. Interrogation des queues non locales. Le processus candidat est comparé aux candidats des autres CPUs et le processus gagnant acquiert le CPU.

Naturellement un mécanisme permettant de renforcer ou de diminuer la priorité en fonction de la localité d'un processus est mis en œuvre, comme cela était déjà le cas dans le `DSS`. Deux valeurs de priorité sont affectées à chaque processus, une priorité avec affinité qui tient compte de la localisation de l'exécution et une priorité sans affinité. Le meilleur représentant local (tenant compte de l'affinité) sera élu s'il est plus prioritaire que le meilleur représentant distant sans tenir compte de l'affinité. Lorsque c'est un processus distant qui est élu, l'ordonnanceur peut très bien échouer à obtenir le verrou de la queue distante, et dans ce cas c'est le deuxième processus local le plus prioritaire qui sera élu comme le résume la figure 8.5.

Adaptation des programmes

Peut-on observer un gain de performance en changeant notre système d'exploitation pour qu'il utilise efficacement les multiples processeurs mis à sa disposition sans pour autant changer les programmes (et donc la façon dont ils ont été conçus) ? La réponse intuitive est oui mais il faut y apporter quelques précisions. L'exécution de deux processus ne sera plus vraiment concurrente, chacun ayant son processeur pour dérouler sa séquence d'instructions. Nous allons donc constater une amélioration dans

la gestion des tâches interactives. Sur un système Unix avec interface graphique, un calcul intensif n'affectera pas la fluidité de l'interface car automatiquement, par le biais des priorités, les processus mis en jeu dans la gestion des fenêtres trouveront un processeur libre pour les accueillir. De la même manière, l'encodage au format H.264 de deux séquences vidéo sera en moyenne presque deux fois plus rapide.

Malheureusement ce gain en performance s'arrête dès que le nombre de processus importants (à nos yeux d'utilisateur !) dépasse le nombre de processeurs. Chaque changement de contexte, même s'il prend place sur le même processeur pour limiter les changements de cache ou du moins les accélérer, consomme du temps. Il faudrait disposer d'un moyen pour rendre l'intérieur de nos programmes parallèle.

Paradoxalement ce n'est pas réellement le parallélisme qui a motivé la mise au point des fils [3] d'activité ou « *thread* ». Le fait d'éviter de passer par des mécanismes de dialogue sophistiqués (les sémaphores par exemple) pour faire dialoguer deux processus intéressait en effet nombre de personnes. C'est pourtant un moyen très efficace de tirer parti d'une architecture multi-processeurs ou multi-cœurs. Nous verrons de plus que des systèmes d'exploitation tels que Windows ou Solaris fondent leur politique d'ordonnancement sur la notion de *thread* plus que sur celle de processus.

8.2 Les *threads*

Introduction

Il fut un temps où l'interaction entre les processus était très largement simplifiée par l'utilisation de variables communes. Il était alors très simple de communiquer des données entre processus, la mémoire était partagée, au sens réel du terme. Ainsi, en 1965, sur le *Berkeley Timesharing System*, les processus partageaient réellement la mémoire et se servaient de ce partage pour dialoguer. La notion de *thread*, même si la terminologie était inconnue à l'époque, était bien là.

L'avènement d'Unix mit fin à ce partage et le dialogue entre les processus fut rendu plus complexe avec la protection de la mémoire : il fallait passer soit par des `IPC` (`signal`, `pipe`...), soit par des segments de mémoire partagés (ceux-ci vinrent plus tardivement).

Au bout d'un certain temps, les utilisateurs d'Unix à qui le partage de la mémoire entre processus manquait cruellement, furent à l'origine de « l'invention » des *threads*, des processus du même genre que les processus classiques, mais partageant la même mémoire virtuelle. Ces processus légers, comme ils furent appelés, apparurent à la fin des années 70.

Avant d'aborder les différents types de *threads* disponibles, nous allons commencer par détailler l'utilisation de la mémoire virtuelle, en pré-supposant qu'un *thread* est

3. Le terme fil est à prendre ici dans le sens ficelle. Le pluriel est particulièrement ambigu en français car il peut être confondu avec le pluriel de « fils » au sens descendant mâle en ligne directe !

avant tout attaché à un processus afin de pouvoir aborder ultérieurement le problème de l'ordonnancement.

Le partage des ressources, mémoire et fichiers

Que partager ?

Chaque processus sous Unix possède un accès virtuel à la mémoire qui lui est propre, ainsi que le montre la figure 8.6. Un *thread*, afin d'acquérir pleinement le qualificatif de « processus léger » doit partager un certain nombre de choses avec le processus dont il est issu.

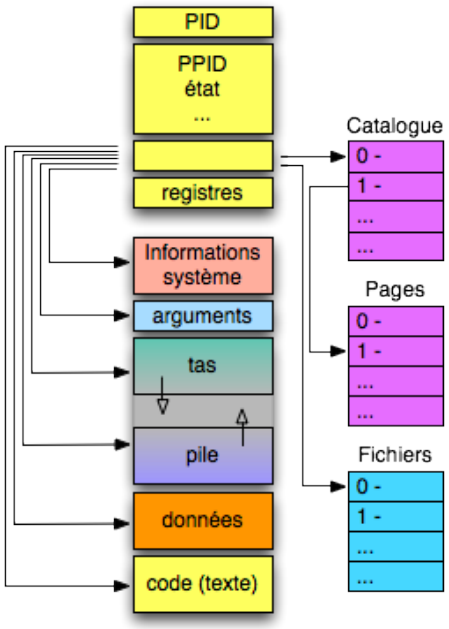


FIGURE 8.6 – *Un processus sous Unix. Une partie des informations (`PID`, `état`... `registres`) fait partie de la gestion des processus organisée par le système d'exploitation. Le reste est relatif au processus et de son accès à la mémoire virtuelle (pile, tas, catalogue...).*

Nous avons placé en premier plan le fait de partager l'accès à la mémoire (nous en verrons le détail après). Donc le catalogue des pages ainsi que les pages seront communs au processus et aux *thread*. Il en sera de même pour la table des fichiers

ouverts, ce qui nous met en alerte sur le fait que certaines précautions devront être prises afin que les accès aux fichiers ouverts ne soient pas incohérents.

Puisqu'un *thread* doit posséder une relative indépendance d'exécution vis à vis du processus dont il est issu afin de pouvoir mettre en place une véritable politique d'ordonnancement en parallèle, il est obligatoire que les informations conservées par le système d'exploitation (état, registres, pointeurs de données ou d'instructions, de pile ou de tas...) ne soient pas partagées. Il faudra donc garder cette structure privée.

Un *thread* est créé par l'appel d'une fonction système et les instructions qui seront exécutées lors de son activation sont naturellement comprises dans le code exécutable du programme dont est issu le processus. Le segment de texte (ou code) d'un *thread* est donc une partie du segment de texte du processus. Il paraît donc naturel que cette zone soit commune aux deux structures. Il en va de même pour les données statiques allouées (déclarations de variables globales, allocations statiques...).

Puisque le besoin initial est le partage de la mémoire, nous conviendrons dès lors qu'il doit être possible de partager le tas (la mémoire allouée dynamiquement). Qu'en est-il de la pile ? Cette dernière sert à conserver en mémoire les différentes variables locales à l'exécution d'une fonction du programme. Le deuxième concept à l'origine des *threads* intervient ici : l'exécution simultanée sur plusieurs processeurs. Une fonction pouvant être appelée par plusieurs *threads* s'exécutant en parallèle, il est impératif que chaque *thread* possède sa propre pile.

Nous arrivons ainsi à la structure de la figure 8.7.

Les précautions à prendre

Nous laissons toujours de côté le problème de l'ordonnancement pour aborder un point crucial qui sera repris plus en détail dans un autre module (IN203). Nous avons maintenant la possibilité d'accéder à la mémoire de manière asynchrone et parallèle. Prenons l'exemple d'une bibliothèque dont un fragment du code source est développé ci-après :

```
int byte_per_line;
int what_size(struct image_mylib *im)
{
   if (im == NULL) return -1;
   if (im->pix_len <= 8) {
      byte_per_line = im->w*im->h;
   } else if (im->pix_len >= 24) {
      byte_per_line = 3*im->w*im->h;
   }
}
...
}
```

La variable globale `byte_per_line` est déclarée de manière globale. Supposons que deux *threads* exécutent cette fonction avec un léger décalage temporel :

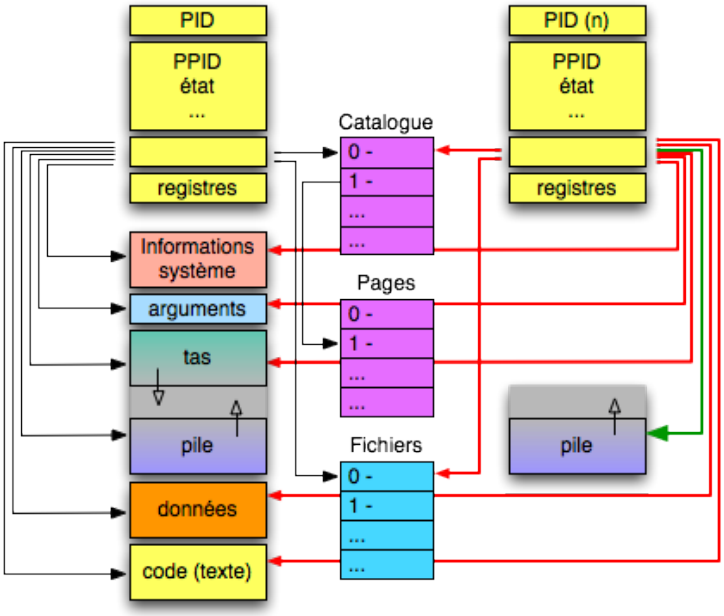


FIGURE 8.7 – *Les* threads *Unix et les accès aux ressources. Les flèches dirigées vers la gauche accèdent aux mêmes ressources. Les structures exploitées par le système d'exploitation (dans la table des processus) sont privées, ainsi que la pile afin que chaque* thread *puisse préserver les variables locales aux fonctions qu'il exécute.*

```
/* thread 0 */
what_size(&image_source);
do_it();
/* test */
if (byte_per_line == 3456) ...
```

```
/* thread 1 */
what_size(&image_dest);
do_it();
/* test */
if (byte_per_line == 3456) ...
```

Nous remarquons dans le `thread 0` qui se déroule à gauche, la variable est modifiée par l'appel de la fonction `what_size()`. Mais lorsque le test a lieu, le contenu de cette variable a été modifié par le `thread 1` !

Il existe souvent dans un code source une partie qui ne peut pas être rendue parallèle. On appelle cette portion de code la **section critique**. Nous en avons donné un exemple trivial dans un programme abominable [4]. Le noyau Unix contient un certain nombre de

4. L'utilisation de variables globales est généralement à proscrire. Les premières versions de la bibliothèque `GIF` possédaient ainsi un certain nombre de variables globales qui rendaient une exploitation

sections critiques. Il est impératif de les protéger car elles sont loin d'être atomiques (exécutables en une instruction sur le processeur).

La première solution adoptée fut de masquer toutes les interruptions matérielles lorsque l'une d'entre elles est traitée. Cela rend chaque appel système bloquant pour le reste des processus ou des *threads*.

La deuxième solution fut d'employer de nouvelles instructions atomiques telles que `CAS` (« *Compare And Swap* ») ou `TAS` (« *Test And Set* »). Ces instructions permettent, en un cycle d'horloge, de comparer deux valeurs de registres et de les échanger, ou de réaliser un test et d'affecter une valeur. Il devint possible de mettre au point des verrous infranchissables et dès lors de protéger les sections critiques.

Gardons toutefois en mémoire que ces protections par verrous sont des goulets d'étranglement pour le système d'exploitation puisque les instructions protégées par ces verrous ne peuvent être exécutées que par un seul processeur (d'où le nom de `BKL` ou « *Big Kernel Lock* » [5]).

Les différents modèles

Ce qui intéressait les programmeurs était avant tout le partage de la mémoire. Le fait de pouvoir réaliser des exécutions concurrentes sur plusieurs processeurs n'était pas encore au goût du jours. C'est pourquoi les premiers modèles de *threads* furent les *threads* utilisateurs encore appelés `ULT` (« *User-Level-Thread* »). Mais la montée en puissance du calcul parallèle nécessitait la mise à disposition des `KLT` (« *Kernel-Level-Thread* »).

Les *threads* utilisateurs

Dans ce cas de figure, le système d'exploitation n'est pas au courant de l'existence des *threads*. Il continue de gérer des processus. L'orchestration des *threads* est réalisée par l'application (le processus) au moyen d'une bibliothèque de fonctions. Le fait de changer de *thread* ne requiert aucun appel système (donc pas de changement du mode utilisateur vers le mode superviseur) et l'ordonnancement est lui aussi relégué à l'application (donc à la bibliothèque).

Ceci paraît relativement avantageux, mais il faut regarder de plus près. L'activité du système d'exploitation se résume, concernant les `ULT` aux actions suivantes :

- le noyau ne s'occupe pas des *threads* mais continue à gérer les processus ;
- quand un *thread* (appelons le `THR0`) fait un appel système, tout le processus sera bloqué en attendant le retour de l'appel système, mais la bibliothèque de gestion des *threads* gardera en mémoire que `THR0` était actif et il reprendra son activité à la sortie de l'appel système ;

multi-threadée illusoire.

5. Le noyau Linux 2.0 gérait le `SMP` mais avec peu de verrous. Le noyau était donc, de manière imagée, un `BKL` à lui tout seul ! Le 2.2 fut presque identique. Ce n'est vraiment qu'à partir du 2.4 que toutes les opérations noyaux ont été *multi-threadées* avec des verrous.

– les *threads* de niveau utilisateur sont donc indépendants de l'état du processus.

Nous pouvons donc dégager quelques avantages et inconvénients de ces `ULT` :

– le changement d'activité d'un *thread* à un autre ne requiert pas d'appel système, il n'y a donc pas de changement de contexte ce qui rend cela plus rapide ;
– c'est à l'application qui emploie la bibliothèque de réaliser l'ordonnancement de l'activité des différents *threads*, il est donc possible d'employer un algorithme parfaitement adapté à cette application précise ;
– les *threads* de niveau utilisateur sont indépendant du système d'exploitation, nous avons seulement besoin de la bibliothèque permettant leur emploi ;
– mais les appels système sont bloquants au niveau des processus et ce sont donc tous les *threads* qui sont bloqués en présence d'un appel système ;
– le système d'exploitation seul affecte chaque processus à un processeur, donc deux *threads* de niveau utilisateur s'exécutant au sein du même processus ne pourront pas être déployés sur deux processeurs différents.

Cette dernière remarque ne plaide pas pour les `ULT` en dépit des avantages que ceux-ci procurent (liberté de l'OS, liberté de l'ordonnancement. . .).

Les *threads* système

Cette fois l'intégralité de la gestion des *threads* est réalisée par le système d'exploitation. On accède aux *threads* par le biais d'un ensemble d'appels système. L'ordonnancement est réalisé non plus sur la base du processus mais sur celle du *thread*. Dressons une liste rapide des avantages et inconvénients :

– le noyau peut répartir les *threads* sur différents processeurs et l'on accède ainsi à un parallélisme d'exécution ;
– les appels système sont bloquants au niveau du *thread* et non du processus ;
– le noyau lui-même peut être rendu *multi-threadé* ;
– mais le nombre de changements de contexte est multiplié par le nombre de *threads* exécutant les mêmes instructions ce qui peut avoir des conséquences néfastes sur la rapidité globale de l'application.

Le mélange `ULT` / `KLT`

Il est naturellement tentant de mélanger les types de *thread*. C'est ce que réalise le système d'exploitation Solaris en offrant des *threads* noyau mais aussi une bibliothèque de gestion de *threads* utilisateur. Ces derniers sont créés dans l'espace utilisateur. C'est l'application (au travers de son processus) qui se charge de gérer leur ordonnancement ainsi que leur attachement à un ou plusieurs *threads* noyau. Ceci permet aux concepteurs d'optimiser l'exécution des différents *threads* utilisateur. Un ensemble de *threads* utilisateur est donc encapsulé dans un « processus léger » qui est maintenant géré par le système d'exploitation en tant que processus à part entière et qui peut donc l'affecter à un processeur particulier. C'est au développeur de faire les choix judicieux pour encapsuler le bon nombre de *threads* utilisateur dans un processus léger (ils se partageront

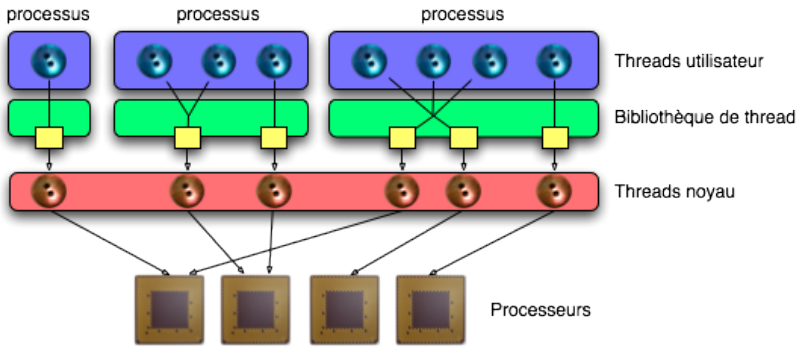


FIGURE 8.8 – *Chaque processus (rectangles du haut) emploie un ou plusieurs* threads *utilisateur par le biais de la bibliothèque. Cette dernière crée des processus légers (petits rectangles superposés au rectangle symbolisant le bibliothèque). Chaque processus léger est associé exactement à un* thread *noyau. Le système d'exploitation se charge ensuite de répartir les différents* threads *noyau sur les différents processeurs.*

le temps de ce processus léger) et de laisser un *thread* utilisateur particulier être géré comme un processus afin de lui garantir une exécution complètement concurrente des autres. La figure 8.8 résume les différents agencements possibles.

Nous ne pouvons pas terminer ce paragraphe sans citer les *threads* Java qui sont un bel exemple de migration réussie du concept de *thread* utilisateur – la première machine virtuelle Java ne possédait que ce type de *threads* – vers les *threads* noyau tout en offrant la possibilité d'employer les *threads* utilisateur. Ceci paraît quelque part normal, puisqu'il existe un point commun entre Solaris et Java : Sun Microsystems [6].

L'ordonnancement

Nous avons déjà abordé l'ordonnancement des processus sur une architecture multi-processeurs à l'aide des affinités entre processus et processeurs. On trouve cette notion d'affinité dans le noyau Linux à l'intérieur du fichier `linux/kernel/sched.c` :

6. Sun a été racheté en 2010 par Oracle.

```
static inline int goodness(struct task_struct * p,
                           int this_cpu,
                           struct mm_struct *this_mm)
{
...
#ifdef __SMP__
   /* Give a largish advantage to the same processor...  */
   /* (this is equivalent to penalizing other processors) */
   if (p->processor == this_cpu)
      weight += PROC_CHANGE_PENALTY;
      ...
```

Mais ce calcul d'affinité ne concerne que l'éligibilité d'un processus vers un CPU. Le cœur de l'ordonnancement se situe ailleurs et doit nécessairement tenir compte de la notion de *thread*.

Nous examinons dans un premier temps l'ordonnanceur Linux pour examiner par la suite d'autres politiques.

L'ordonnanceur de *threads* Linux

Les *threads* sont liés à un processus et doivent probablement se partager le temps d'occupation à l'intérieur du contexte de ce processus. En y réfléchissant un peu plus, il est d'ailleurs abusif de parler de *processus maître* et de ses *threads*. Cette distinction n'a pas lieu d'être, un processus n'utilisant pas d'appels système tel que `pthread_create()` possèdent néanmoins un *thread*, lui même. Il n'y a donc pas de notion de **maître**.

Restreindre l'utilisation d'un CPU aux *threads* pendant le quantum de temps durant lequel le processus auquel ils sont attachés est éligible paraît peu efficace... mais pas forcément ! Examinons les deux politiques qui peuvent être mises en place en nous plaçant dans un premier temps à l'intérieur d'une architecture mono-processeur. Le noyau Linux ne tient compte que des processus, chacun d'entre eux étant considéré comme un ensemble T de n *threads*.

Sous Linux les *threads* sont mis à la disposition de l'utilisateur par l'intermédiaire de l'API POSIX `libpthread`. Cette bibliothèque d'appels permet de créer des *threads* en leur affectant un contexte d'exécution parmi :

- `PTHREAD_SCOPE_PROCESS` : le *thread* est dépendant, toujours du point de vue de l'ordonnancement, du processus dont il fait partie ;
- `PTHREAD_SCOPE_SYSTEM` : le *thread* est directement « attaché » à un *thread* système et donc indépendant, du point de l'ordonnancement, du processus dont il fait partie.

Quand la création se place dans le contexte « processus », c'est le processus dans son ensemble qui rentre en compétition pour l'accès au CPU. Les *threads* qui composent ce processus rentreront en compétition **les uns avec les autres** pour accéder au CPU lors de l'élection du processus. C'est ce que montre la figure 8.9. Remarquons que peu importe la façon dont chaque *thread* du deuxième processus sera élu pour accéder au

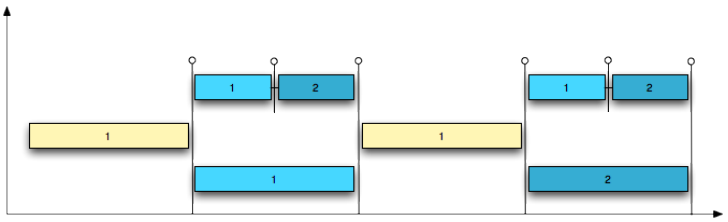


FIGURE 8.9 – *Chaque* thread *est créé avec un contexte lié au processus dont il fait partie. Le processus* 1 *ne possède qu'un seul* thread *tandis que le deuxième possède 2* threads. *L'accès au* CPU *est équiréparti et ainsi l'unique* thread *du processus 1 reçoit* 50 % *des quantums de temps tandis qu'un* thread *du processus 2 n'en reçoit que* 25 %.

CPU, soit en divisant le quantum alloué au processus 2 pour dérouler les instructions de chaque *thread*, soit en allouant le quantum à un seul *thread* alternativement. En effet, le changement de contexte entre deux *threads* du même processus est quasi négligeable. Il est toutefois beaucoup plus simple de ne pas découper chaque quantum mais plutôt d'attribuer un quantum (au sein d'un processus) à chaque *thread* à tour de rôle.

Lors d'une création dans un contexte « système », soit le deuxième cas de figure présenté, tout se passe comme si le *thread* était « placé » dans la table des processus et rentrait en compétition avec les autres processus pour obtenir un quantum de temps d'exécution. Un processus possédant de nombreux *threads* est alors largement avantagé par rapport à un processus ne possédant qu'un seul *thread* comme le montre la figure 8.10. Mais par défaut, la création de *thread* utilise le contexte du processus.

FIGURE 8.10 – *Chaque* thread *rentre en compétition pour l'accès au* CPU *indépendamment du processus auquel il appartient. L'unique* thread *du processus 1 obtient ainsi* 33% *du temps* CPU*, au même titre que les deux* threads *du second processus.*

Nous pouvons résumer ces deux modes d'ordonnancement en disant que dans le cas SCOPE_SYSTEM chaque *thread* compte pour 1 processus, alors que dans le cas SCOPE_PROCESS chaque *thread* compte pour un tantième du processus dont il dépend.

Il reste à combiner cette gestion avec plusieurs processeurs et donc utiliser l'ordonnanceur multi-queues. La figure 8.11 donne un exemple sommaire et forcément simpliste du déroulement des *threads* dans une structure à deux cœurs.

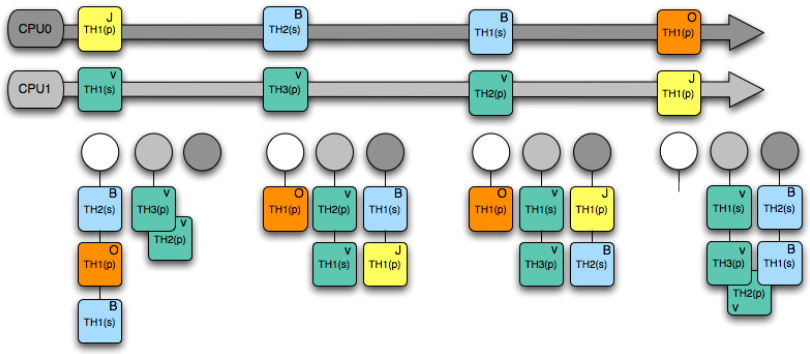


FIGURE 8.11 – *Cette figure représente de façon très simple un processus d'ordonnancement dans une architecture multi-cœurs. Chaque processeur possède sa queue d'ordonnancement (un cercle) et il existe aussi une queue générale. On remarque deux types de* thread*, ceux déclarés dans un contexte système (S) et ceux déclarés dans un contexte processus (P). Le processus bleu (marqué d'un « B ») et le processus vert (marqué d'un « V ») s'exécutent en parallèle. L'absence de changement de contexte général entre les* threads *du processus vert lui permette d'être élu assez facilement. Ceci montre bien la complexité de réalisation d'un ordonnanceur car le processus vert pourrait fort bien affamer les autres processus.*

Les autres ordonnanceurs

Le système d'exploitation Windows, du moins dans ces versions récentes (*i.e.* à partir de Windows Vista, axe sa politique d'ordonnancement autours des *threads*. Un processus est une structure complexe dont la création est réalisée *ex nihilo* et donc en général plus lentement que sous un Unix classique. La création d'un *thread* sous Windows Vista est par contre plus rapide que sous Linux.

Rappelons que la création d'un processus par la fonction `CreateProcess()` permet de créer un processus sans réelle notion de parent / fils et possédant ce que Windows Vista appelle un *thread* principal. Windows Vista ne possède aucun équivalent à la fonction `fork()` et Microsoft annonce que son émulation serait très difficile. Pourquoi ? Simplement parce que : « `fork` est une fonction difficile à utiliser dans un système Unix de type SMP, car il faut penser à tout et qu'il y a de nombreux problèmes à résoudre avant de pouvoir programmer cette fonction correctement [7] » ! Et de conclure sur le fait que la fonction `fork` n'est pas vraiment appropriée dans un environnement de type `SMP` ! Si le développement des systèmes d'exploitation devait s'arrêter à chaque fois qu'une chose est difficile, nous en serions encore à utiliser des moniteurs résidents !

7. extrait traduit de « Windows System Programming, 3rd Edition » de Johnson M. Hart.

Tout comme la plupart des systèmes d'exploitation, Windows Vista utilise un ordonnanceur multi-queues. Introduit assez récemment, ce nouvel ordonnanceur permet à l'utilisateur, mais surtout aux développeurs, de ranger certaines applications ainsi que leurs *threads* dans un niveau de priorité spécifique : les applications multimédia. Ceci se fait par l'intermédiaire de la base de registres. On trouve ainsi un service gérant ces priorités et l'appartenance de certaines applications à cette queue spéciale : le `MMCSS` ou *Multimedia Class Scheduler Service*. Le `MMCSS` permet d'augmenter la priorité de toutes les *threads* et possède donc pour sa part une priorité très élevée de 27 (les priorités des *threads* sous Windows Vista vont de 0 à 31). Seul le *thread* de gestion de la mémoire possède une priorité plus élevée de 28 ou 29. Cependant, même s'il est agréable de ne pas interrompre la lecture du dernier morceau de musique acheté légalement sur un site de vente de musique en ligne, il peut, accessoirement, être utile de lire son courrier électronique ou éventuellement d'avancer le développement d'un projet informatique. Il est donc primordial que ce système de priorité élevée pour les *threads* qualifiées de temps réel n'affecte pas ou n'affame pas les autres processus. Une partie du temps `CPU` est donc réservé par le `MMCSS` pour les autres *threads*, de l'ordre de 20 %, mais cette quantité peut être modifiée par le biais de la base de registres. Afin d'éviter aux *threads* multimédia de prendre la main pendant ce laps de temps, le `MMCSS` leur affecte une priorité variant de 1 à 7.

Le point le plus important est toutefois la mise en place d'une mesure effective du temps `CPU` consommé par un *thread*. En effet, il est courant qu'un processus soit mis en attente par une interruption logicielle ou matérielle, puis qu'il reprenne. Le quantum de temps qui lui était affecté n'aura donc pas été intégralement consommé mais le système d'exploitation le tiendra pourtant pour acquis. Les nouveaux processeurs intègrent un registre de compteur de cycles et Windows Vista tire profit de ce compteur pour réaffecter le `CPU` à un *thread* dont la fenêtre d'exécution a été interrompue. Ceci assure effectivement que chaque *thread* pourra dérouler ses instructions en bénéficiant au moins une fois de la totalité de sa fenêtre d'exécution.

Le système Solaris introduit 4 objets pour la gestion des *threads* :

- les *threads* noyau : c'est ce qui est effectivement programmé pour une exécution sur le `CPU` ;
- les *threads* utilisateur : il s'agit de l'état d'un *thread* dans un processus à l'intérieur du mode utilisateur ;
- le processus : c'est l'objet qui maintient l'environnement d'exécution d'un programme ;
- le processus « léger » : il s'agit du contexte d'exécution d'un *thread* utilisateur qui est toujours associé avec un *thread* noyau.

Les services du noyau et différentes tâches sont exécutés en tant que *threads* en mode noyau. Lorsqu'un *thread* utilisateur est créé, le processus léger (ou *lwp* pour *Light Weight Process*) et les *threads* du mode noyau associés sont créés et associés au *thread* utilisateur.

Solaris possède 170 niveaux de priorité différents. Comme sur la plupart des systèmes d'exploitation récents, les niveaux de priorité correspondent à des classes

différentes d'ordonnancement (les priorités de valeur faible sont les moins prioritaires) :

- TS (*Time Sharing*, temps partagé) : il s'agit de la classe par défaut pour les processus et les *threads* du mode noyau associés. Les priorités à l'intérieur de cette classe vont de 0 à 59 ;
- IA (*InterActive*) : il s'agit d'une extension de la classe TS permettant à certains *threads* associés à des applications graphiques (fenêtre) de réagir plus vite ;
- FSS (*Fair-sare Scheduler*) : il s'agit de l'ordonnanceur classique que l'on trouve sous Unix et qui permet d'ajuster la priorité en fonction de données statiques mais aussi dynamiques ;
- FX (*Fixed-priority*) : à la différence des *threads* précédents, ceux-ci n'ont qu'une priorité fixe ;
- SYS (système) : il s'agit des priorités des *threads* du mode noyau. Leur priorité varie de 60 à 99 ;
- RT (*real-time*) : les *threads* de cette classe possèdent un quantum fixe alloué sur le `CPU`. Leur priorité varie entre 100 et 159 ce qui permet à ces *threads* de prendre la main aux dépens des *threads* système.

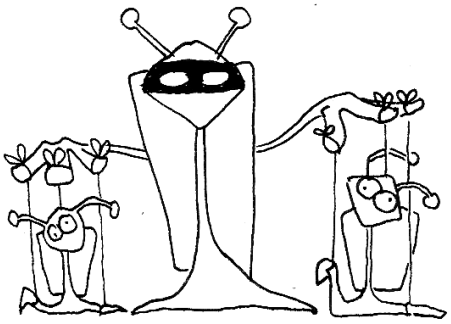
8.3 Conclusion

L'arrivée des processeurs multi-cœurs et des architectures multi-processeurs n'a pas radicalement changé le monde Unix. En effet, il existait déjà tous les prémisses du calcul distribué dans les projets comme Beowulf et les différents problèmes inhérents à ce parallélisme (mémoire, ordonnancement) étaient connus et des solutions mises en œuvre.

Il a fallu avant tout optimiser l'accès à la mémoire et réduire le temps consacré aux changements de contexte. Même si ce n'est pas de prime abord ce qui mena aux *threads*, le fait de partager la mémoire (tas, code, données) accéléra grandement le changement de contexte. L'enjeu est maintenant de réduire les allers retours entre les processeurs et de définir des affinités entre processeur et mémoire (`NUMA`).

9

La virtualisation des systèmes d'exploitation



VIROSE n. f. Infection due à un virus.
VIRTUALITÉ n. f. Caractère de ce qui est virtuel.
VIRTUEL, ELLE adj. (du latin *virtus*, force) **1.** Qui n'est qu'en puissance : potentiel, possible...
VIRTUELLEMENT adv. De façon virtuelle.

Ce rapide extrait du PETIT LAROUSSE ILLUSTRÉ nous montre l'absence dans le dictionnaire du mot *virtualisation*. Que le lecteur nous pardonne, nous allons pourtant l'employer de manière abondante dans ce chapitre et nous l'avons même déjà utilisé dans les chapitres précédents. En effet nous avons abordé la virtualisation, ne serait-ce qu'au travers des accès à la mémoire physique (cf. chapitre 6). Remarquons que l'accès au système de fichiers (aux systèmes...) est, quelque part, lui aussi virtualisé puisque du point de vue de l'utilisateur, que le fichier qu'il manipule se trouve sur le disque dur de l'ordinateur ou quelque part sur le réseau, peu importe, l'accès reste identique au travers des appels système `open()`,`read()`, `write()` et `close()`. De la même manière les clusters de systèmes (Beowulf) tels que ceux évoqués dans le chapitre précédent (cf. chapitre 8) offrent la vision d'un ordinateur virtuel cumulant la puissance des nœuds du cluster.

Qu'est-ce qu'une machine virtuelle ? Il s'agit avant tout d'un système électronique sans existence matérielle, chaque composante dudit système étant purement et simple-

ment simulée (nous apporterons plus de détails et de bémols sur cette « simulation »). Wikipédia [1] donne la définition suivante d'une machine virtuelle :

> *La virtualisation consiste à faire fonctionner sur un seul ordinateur plusieurs systèmes d'exploitation comme s'ils fonctionnaient sur des ordinateurs distincts. On appelle serveur privé virtuel (Virtual Private Server ou VPS) ou encore environnement virtuel (Virtual Environment ou VE) ces ordinateurs virtuels.*

Dans le cas d'une utilisation grand public, telle que l'offrent des solutions comme *VirtualBox*, *QEMU* ou encore *VMware*, le possesseur d'un ordinateur unique sur lequel est installé un système d'exploitation unique [2] peut se servir d'un logiciel qui fera office de machine virtuelle et démarrer ainsi un nouveau système d'exploitation totalement différent de celui qu'il emploie. Ce système d'exploitation virtualisé lui permettra ainsi d'utiliser d'anciens logiciels devenus incompatibles avec les systèmes d'exploitation récents ou simplement d'accéder à une offre logicielle différente (architecture PC *vs* Mac OS X).

Le concept de virtualisation et les travaux associés ont commencé au siècle dernier en France, à Grenoble, en 1965 dans le centre de recherche d'IBM France !

9.1 Les intérêts et les enjeux de la virtualisation

Les intérêts d'une machine virtuelle

Il peut sembler toutefois curieux de simuler un ordinateur pour des besoins autres que ceux liés à la conception. Pourtant, disposer de machines virtuelles intéresse différentes populations.

Les spécialistes, experts, en sécurité informatique peuvent disposer au travers d'une machine virtuelle d'un moyen sans danger et très efficace d'observer le comportement de logiciels malveillants. Une fois la machine corrompue il est très simple de revenir en arrière puisque tout ce qui la compose est simulé et peut donc être sauvegardé à un instant t quelconque. L'observation de logiciels malveillants est ainsi parfaitement sécurisée d'une part, et d'autre part reproductible et ce de manière très rapide et déterministe (c'est la machine dans son ensemble qui fait un retour vers le passé). Il est aussi possible de travailler par différences et de comprendre le comportement du logiciel malveillant par une analyse avant / après.

Pour une entreprise la virtualisation offre de nombreux avantages. Tout d'abord, l'utilisation de la virtualisation permet d'héberger au sein d'une seule machine physique plusieurs machines virtuelles. Lorsque l'on s'intéresse au coût d'un serveur et que l'on regarde de plus près l'utilisation du CPU, on remarque que la plupart des serveurs

1. `http://fr.wikipedia.org/wiki/Virtualisation`

2. La majeure partie des ordinateurs personnels ne comportent pas de gestionnaire de boot et n'hébergent qu'un système d'exploitation. L'utilisation de `GRUB` et de plusieurs partitions de démarrage hébergeant des systèmes d'exploitation différents est encore peu répandue.

sont largement sous-employés. La valeur moyenne d'utilisation du processeur est de l'ordre de 10 %, ce qui représente donc un gâchis énorme, tant en investissements, qu'en ressources. Ce serveur doit en effet être maintenu (les contrats de maintenances matériels sont généralement très chers), il doit être placé dans un local climatisé, il est raccordé au réseau informatique de l'entreprise par des câbles et enfin il occupe une place non négligeable dans une baie informatique.

FIGURE 9.1 – *Les baies informatiques permettent de « ranger » des serveurs physiques. Mais la place disponible n'est généralement pas extensible et les salles accueillant ces baies doivent être climatisées, placées dans un environnement électrique de confiance (onduleur). La virtualisation peut apporter une solution. (image d'une baie de serveurs à l'Université Toulouse 1 Capitole)*

Un ensemble de serveurs virtuels peut être hébergé par un seul serveur physique. La moyenne actuelle d'hébergement pour des serveurs physiques de moyenne gamme étant de l'ordre de 10 à 20 machines virtuelles par serveur, nous réalisons ainsi un gain en câblage (seules les alimentations et les interfaces réseaux de la machine physique sont connectées) et un gain de place (un emplacement là où il aurait fallu une baie entière). La climatisation reste par contre sensiblement constante puisque nous allons maintenant utiliser le CPU et les autres composants d'une façon beaucoup plus régulière. Mais nous venons de réaliser un gain conséquent sur notre marché de maintenance puisque pour la valeur d'un serveur et sa garantie pièces et main d'œuvre, nous avons 15 serveurs différents (ou identiques... mais ne brûlons pas les étapes !).

Le lecteur attentif nous objectera immédiatement que nous avons certes un seul contrat de maintenance sur une machine physique, mais que lorsque celle-ci tombe en panne ce sont 15 serveurs qui s'arrêtent subitement de fonctionner. Effectivement, la tolérance aux pannes est un sujet crucial dans les DSI actuelles et la reprise d'activité doit être sinon immédiate, tout du moins la plus rapide possible. En règle général il est très dangereux de mettre l'intégralité des machines virtuelles sur un seul serveur et le bon sens mène plutôt vers une solution ou deux serveurs physiques hébergent le parc

de serveurs virtuels. Nous arrivons en fait à la même solution que celle proposée par un environnement physique comme montre la figure 9.2. La tolérance aux pannes est donc gérée de la même manière sur un parc virtuel et sur un parc réel.

Nous pouvons enfin utiliser cette virtualisation pour réaliser des tests qu'il serait impensable de mener sur une machine de production. L'application d'une mise à jour ou d'une rustine sur un logiciel peut parfois s'accompagner de quelques surprises. Pire, le passage vers une nouvelle version du système d'exploitation peut se révéler très dangereux, certaines applications cesseront de fonctionner correctement. Dans le cas du particulier aventureux, le passage d'une Ubuntu 9.04 vers la version 10.04 est souvent vécu comme une longue suite d'interrogations métaphysiques sur le sens à donner à la phrase « `/dev/sda` - Une table de partition `GPT` a été repérée. Que voulez-vous faire : Partitionnement libre ou avancé ». La seule question qui se pose à ce moment est « vais-je perdre toutes mes données ». Notre particulier aventureux a pourtant déjà rencontré cette étape de formatage du disque dur et il a brillamment franchi ce cap, mais la présentation était différente, les questions posées différemment (voir fig. 9.3) et surtout c'était un nouvel ordinateur sans rien sur le disque, alors recommencer trente fois ne portait pas à conséquence. Maintenant, l'enjeu n'est plus le même après un an d'utilisation ! Une multitude d'autres questions viendront après avoir surmonté cette étape : « mon imprimante, mon scanner, mon réseau, mon mail, les photos de Tata… ».

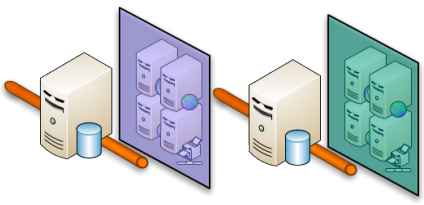


Figure (a)

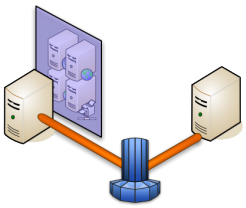


Figure (b)

FIGURE 9.2 – *[Fig. (a)] Dans un environnement de production les serveurs (Web, impression,…) sont généralement présents en double afin de permettre une reprise d'activité en cas de panne. La virtualisation permet de conserver cette redondance. Ici le parc de serveurs virtuels hébergés sur le serveur de gauche est identique à celui du serveur de droite. Les deux serveurs physiques sont interchangeables.*
[Fig. (b)] Une solution plus avancée place les fichiers liés aux différentes machines virtuelles sur un espace de stockage accessible par différents serveurs physiques. La panne d'un serveur, et le fait que les données des machines virtuelles n'ont pas à être déplacées, permet une reprise sur incident très rapide.

L'utilisation d'un logiciel de virtualisation permet de se familiariser avec les différentes étapes d'une installation ou d'un changement majeur de version de système

d'exploitation. La copie des anciens fichiers de configuration, ou leur comparaison, dans cette nouvelle machine virtuelle permettra de vérifier qu'ils sont toujours compatibles avec les services mis à jour.

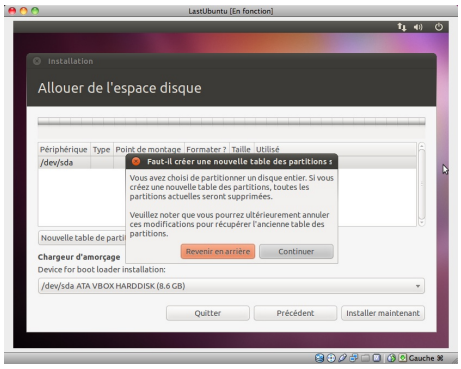


FIGURE 9.3 – *La douloureuse étape de choix des partitions devient plaisante puisque l'erreur est permise (Installation de Ubuntu-10.10amd64 sous Mac OS X à l'aide de* VirtualBox*) !*

C'est naturellement dans l'entreprise que le recours aux serveurs virtuels permet d'aborder les mises à jour sans crainte de devoir restaurer une configuration à partir de sauvegardes sur disques (ou pire d'une réinstallation complète à partir d'une « image disque »). Un logiciel de virtualisation étant avant tout un logiciel, il permet en règle générale de prendre un « instantané » d'une machine virtuelle, *i.e.* une photographie de l'état de la machine, mémoire, disque, interruptions matérielles comprises ! La mise à jour pourra être appliquée sur cet instantané dans un contexte réseau différent, ou sur une autre machine virtuelle, puis elle sera mise à l'épreuve pour enfin être validée. Dans ce cas l'interruption de service sera minime puisque c'est l'instantané qui sera mis en lieu et place du serveur initial.

Ce ne sont pas les seuls avantages de la virtualisation. Un certain nombre d'enjeux doivent être pris en considération.

Préparer l'avenir

D'un point de vue strictement économique, une entreprise (telle qu'une SSII) doit rester compétitive et décliner son offre en étant à la fois réactive et flexible. La réactivité impose de pouvoir mener des tests logiciels sur des systèmes d'exploitation divers et variés. Maintenir autant de serveurs physiques que de versions de systèmes d'exploitation serait bien trop onéreux.

Dans la même optique, s'adapter facilement et simplement à la demande, aux besoins des clients, est chose plus aisée lorsque l'on peut compter sur des environnements

de développements installés sans risque et sans grever le budget par l'acquisition d'un nouveau serveur.

Mais dans ce domaine c'est de loin la possibilité d'installer et d'administrer à distance des machines qui a un impact grandissant sur les coûts. Un serveur virtuel peut en effet être installé, administré, arrêté et redémarré depuis son socle comme le montre la figure 9.4 (la machine physique, ou du moins le logiciel de virtualisation).

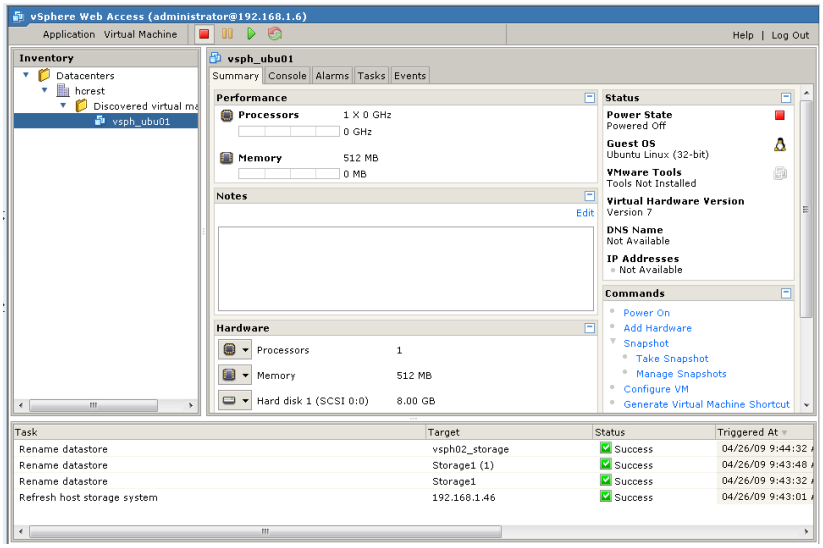


FIGURE 9.4 – *La console d'administration d'un hyperviseur (ici* Vmware ESX*) permet d'accéder aux différentes machines virtuelles hébergées.*

Toutefois il faut aussi réfléchir à la nécessaire formation qui accompagne obligatoirement le passage d'une solution totalement physique à une solution virtualisée. Les administrateurs doivent apprendre de nouvelles techniques, l'état d'esprit doit changer car la gestion d'un parc de machines virtuelles n'est pas du tout la même que celle d'un ensemble de machines physiques.

9.2 Les différentes solutions

Il existe différents moyens de réaliser une virtualisation, ceux-ci dépendant généralement de l'utilisation que l'on veut en faire, sécurisation, hébergement de systèmes d'exploitation variés et différents, simulation. Nous allons examiner ces solutions, sans entrer en profondeur dans les détails mais plutôt en analysant les modifications qu'elles imposent au système d'exploitation hôte (la machine physique) et aux sys-

tèmes invités (les machines virtuelles). Nous examinerons ainsi, le cloisonnement, la para-virtualisation, la virtualisation assistée par le matériel et la virtualisation complète.

Précisons quelques choix de vocabulaire :

– le système qui héberge les machines virtuelles sera indifféremment appelé : système hôte, système physique, socle ;
– les systèmes virtualisés seront appelés : environnement virtuel, système invité, système hébergé, instance virtuelle.

Le cloisonnement ou l'isolation

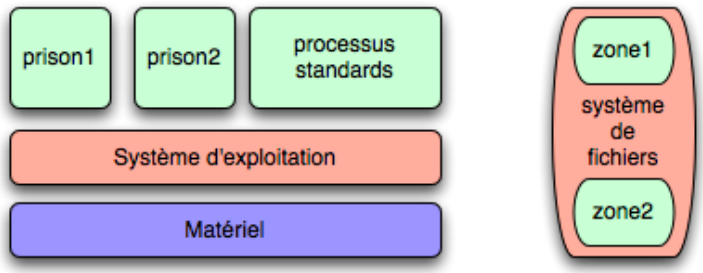


FIGURE 9.5 – *Le principe des prisons permet de cloisonner différentes applications en leur associant de plus des utilisateurs et un administrateur (`root`) qui leur sont propres.*

Une des techniques les plus simples à mettre en œuvre est le cloisonnement. Vous pouvez avoir chez vous, sur votre ordinateur, un ensemble de fichiers relatifs aux corrigés des travaux dirigés du cours d'IN201. Vous souhaiteriez les mettre à disposition de certaines personnes par une connexion sécurisée telle que `ssh`[3] mais pour autant vous ne souhaitez pas mettre l'intégralité de votre disque dur à disposition de ces amis privilégiés. L'appel système `chroot()` permet de se déplacer dans un répertoire qui devient le répertoire racine du système :

3. *Secure SHell* est un moyen de se connecter sur un ordinateur distant par l'intermédiaire d'une connexion chiffrée. À la différence d'un serveur Web, cette connexion ne sera possible que si l'utilisateur distant possède un compte sur la machine à laquelle il veut accéder et surtout elle est bien plus sécurisée qu'un simple serveur Web avec authentification.

```
$ cd /
$ /bin/ls
bin    dev   mnt       root  sys         vmlinuz
boot   etc   lib       opt   sbin        tmp
cdrom  home  selinux   usr   initrd.img  media
proc   srv   var
$ chroot /usr/local/test /bin/ls /
bin    lib    usr
$
```

Dans l'exemple qui précède, la commande `"/bin/ls /"` exécutée dans le contexte du `chroot` prend place à l'intérieur du répertoire local `/usr/local/test`. Cette commande demande la liste des fichiers et répertoires présents à la racine du système de fichiers, et nous voyons clairement que la réponse obtenue est différente de celle de la commande exécutée normalement. Nous commençons à voir comment tirer parti de cette commande pour isoler différents contextes d'utilisation.

C'est ce qui est mis en œuvre dans les isolateurs tels que les *zones* sous Solaris ou les *jails* sous *FreeBSD*. Nous ne parlerons pas de virtualisation car un processus exécuté dans un environnement *chrooté* n'accède qu'à une partie restreinte du système de fichiers, mais il partage néanmoins les notions d'utilisateurs, de groupes et bien d'autres choses avec le système d'exploitation.

Les *jails* améliorent la technique employée par `chroot` afin de fournir aux processus *chrootés* un environnement complet (utilisateurs, groupes, ressources réseaux, etc.). Ainsi une « prison » est caractérisée par quatre éléments :

- une arborescence dans le système de fichiers. C'est la racine de cette arborescence qui deviendra la racine de la prison ;
- un nom d'hôte ou `FQDN` (*Fully Qualified Domain Name*) ;
- une adresse `IP` qui viendra s'ajouter à la liste des adresses gérées par l'interface réseau ;
- une commande qui sera exécutée à l'intérieur de la prison.

Tous les processus qui seront exécutés à l'intérieur de la prison seront gérés de façon classique par l'ordonnanceur. Il en sera de même de la mémoire. Chaque processus fera donc partie de la table des processus du système.

Nous retrouvons ce principe dans les zones Solaris. Il s'agit ni plus ni moins d'une amélioration [4] du `chroot` avec une gestion des processus centralisée.

Il est toutefois possible de gérer de manière différente chaque système invité, mais il faut pour cela intervenir sur le noyau du système d'exploitation hôte. C'est ce que fait OpenVZ sans pour autant présenter une abstraction du matériel comme le feront les mécanismes que nous étudierons après.

Afin de conférer aux différentes instances virtuelles une vie propre et donc la gestion des processus, OpenVZ modifie le noyau Linux pour fournir plusieurs tables des processus, une par instances, ainsi qu'une table pour l'hôte, et donc un ordonnanceur

4. Attention, l'amélioration est toutefois très importante car les zones tout comme les *jails* permettent bien plus de choses qu'un simple `chroot` !

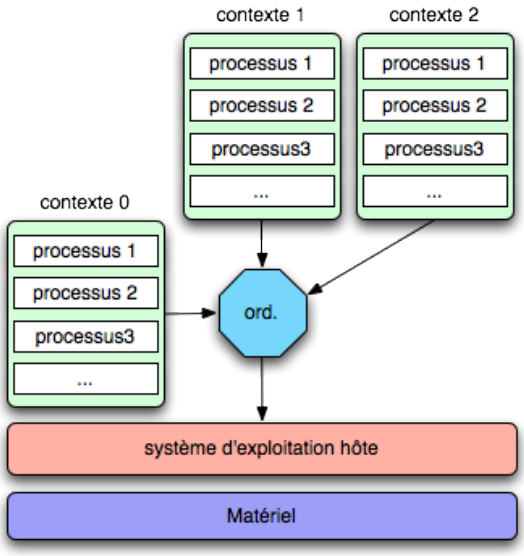


FIGURE 9.6 – *Sur un hôte OpenVZ il n'existe qu'un seul système d'exploitation, celui de l'hôte. Les instances virtuelles n'ont pas de noyau mais bénéficient de systèmes de fichiers indépendants. Leurs processus sont gérés dans différents contextes et font partie de l'ordonnancement général du noyau du socle. Chaque instance virtuelle peut donc bénéficier de tout le* `CPU` *disponible.*

qui prendra en compte ces différentes tables. Chaque instance virtuelle possède ainsi une table des processus, mais c'est le système hôte qui intègre dans son ordonnanceur cette table des processus en plus de la sienne. OpenVZ étant un isolateur, les instances virtuelles gérées par l'hôte n'ont pas de système d'exploitation (voir figure 9.6). Le système hôte est lancé dans un contexte particulier (la notion de contexte faisant partie des ajouts noyau du socle réalisés par OpenVZ), les différentes instances virtuelles exécuteront leurs processus dans un autre contexte.

Le même partage existe pour les accès à la mémoire. Chaque processus, qu'il appartienne au socle ou à une instance virtuelle, possède ses accès à la mémoire virtuelle au travers de sa table des pages. C'est donc toute la mémoire qui est partagée entre le socle et les instances. Par contre, l'utilisation de contexte (et de toutes les informations complémentaires ajoutées au noyau du socle) permet de limiter la mémoire mise à disposition d'une instance. Notons que le réseau est lui aussi virtualisé, ceci permet à chaque instance de posséder sa propre adresse `IP` ainsi que ses propres règles de pare-feu.

Les avantages d'un tel système sont naturellement liés aux performances :

– un très faible pourcentage du temps CPU est consacré à la virtualisation puisqu'il n'existe qu'un seul noyau en fonctionnement ;
– chaque instance n'est rien d'autre qu'une arborescence de fichiers telle qu'on la trouverait sur un système Unix classique. Créer une instance revient donc à copier des fichiers ;
– chaque instance est parfaitement isolée des autres et du système hôte. Ce dernier voit par contre l'intégralité des instances et peut interagir avec chacune d'entre elles.

Il est par contre tout à fait impossible d'héberger une instance d'un autre type qu'Unix puisque seul le système de fichiers est copié. Les appels système doivent donc être communs car seul le noyau de l'hôte est disponible pour les exécuter !

De nombreuses instances peuvent être placées sur un hôte :

```
[root@s-openvz ~]# /usr/sbin/vzlist -a
      VEID      NPROC STATUS  IP_ADDR         HOSTNAME
       103         48 running 10.45.10.83      v-vir01.moi.org
       104          6 running 10.45.10.161     v-vir02.moi.org
       109         45 running 10.45.10.78      v-vir03.moi.org
       110         38 running 10.45.10.127     v-vir04.moi.org
       111         18 running 10.45.10.160     v-vir05.moi.org
       112          - stopped 10.32.10.162     v-test1.moi.org
       115         30 running 10.45.10.248     v-vir07.moi.org
       116          - stopped 10.73.10.28      v-kaput.moi.org
       117         33 running 10.45.10.137     v-vir08.moi.org
       118         34 running 10.45.10.200     v-enprod.moi.org
       120          - stopped 10.44.10.43      v-entest.moi.org
```

Et l'interrogation de l'ensemble des processus s'exécutant sur l'hôte permet de voir les différents contextes d'exécution :

```
 8909 ?        Ss     0:17 init [3]
10913 ?        Ss     0:26  \_ ...
10989 ?        SN     0:00  \_ /sbin/bunch_agentd
10992 ?        SN     0:30  |   \_ /sbin/bunch_agentd
12807 ?        Ss     0:10  \_ /usr/libexec/postfix/master
12823 ?        S      0:01  |   \_ qmgr
23313 ?        S      0:00  |   \_ pickup
13123 ?        Ss     0:11  \_ /usr/sbin/httpd
23209 ?        S      0:37  |   \_ /usr/sbin/httpd
13280 ?        Ss     0:11  \_ winbindd
12871 ?        S      0:00      \_ winbindd
10276 ?        Ss     0:01 init [2]
17091 ?        Ss     0:00  \_ ...
16447 ?        SN     0:00  \_ /sbin/bunch_agentd
18569 ?        SN     0:30  |   \_ /sbin/bunch_agentd
24841 ?        S      0:00  |   \_ /usr/sbin/winbindd
18667 ?        Ss     0:02  \_ /usr/sbin/apache2
29167 ?        S      0:00  |   \_ /usr/sbin/apache2
13144 ?        S      0:00  \_ /bin/sh /usr/bin/mysqld_safe
13189 ?        Sl     0:09      \_ /usr/sbin/mysqld
```

La virtualisation complète

Afin d'héberger des systèmes d'exploitation divers et variés, il est impératif de pouvoir exécuter leur noyau. Les isolateurs montrent alors leurs limites et on doit faire appel à une solution de virtualisation complète. L'intégralité d'une machine physique est ainsi simulée.

Le noyau d'un système d'exploitation n'est rien d'autre qu'un programme et virtualiser un système d'exploitation revient donc à exécuter un programme un peu particulier, le noyau, mais en faisant singulièrement attention à tous les appels qu'il déclenche vers le matériel. En effet seul l'hôte doit avoir la main sur le matériel, sinon chaque instance pourra à sa guise réaliser des modifications relativement pénalisantes et parfois même dangereuses (un système d'exploitation permet l'accès au matériel et contrôle l'accès au matériel). Différentes solutions s'offrent à nous pour réaliser un tel « simulateur » :

- L'émulation complète du noyau et des différents appels. Il conviendra d'analyser chaque instruction pour vérifier ce qu'elle désire réaliser et quels sont ses droits puisque le système hébergés, même s'il croît fonctionner dans l'anneau de privilège 0 est en fait placé dans l'anneau 3. Cette solution apparaît comme rapide et surtout ne requiert pas de modification du noyau invité.
- La réécriture des sources des noyaux invités paraît peu envisageable même si les performances seraient alors accrues (pas de nécessité de vérifier chaque instruction). Nous examinerons cette solution plus tard.
- Utilisation de certaines technologies telles que `Intel-Vtx` ou `AMD-V` pour obtenir une accélération matérielle ! Nous reportons l'examen de cette solution pour la

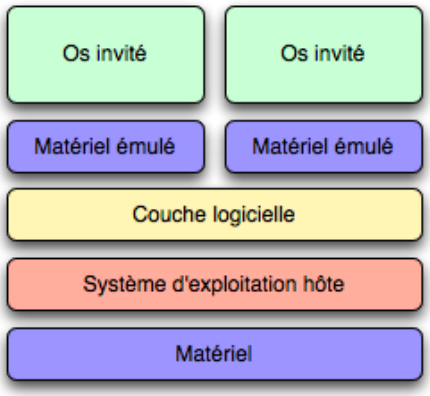


FIGURE 9.7 – *On émule le matériel afin de faire croire aux systèmes d'exploitation invités qu'ils s'exécutent sur une machine physique. On énonce par abus de language que les systèmes hébergés n'ont pas « conscience » d'être virtualisés.*

dernière partie.

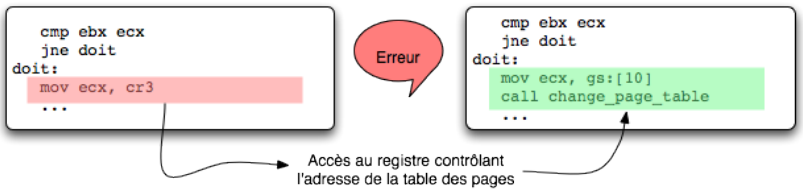


FIGURE 9.8 – *Dans cet exemple, le logiciel de virtualisation remarque l'utilisation par un noyau invité du registre `CR3` qui contrôle l'adresse de la table des pages mémoire et intervient afin de réaliser une traduction en appelant une fonction prédéfinie qui se chargera de réaliser, mais de manière sécurisée, ce changement.*

Un logiciel de virtualisation n'est pas un simulateur [5], il est en effet beaucoup plus restrictif puisqu'il ne peut héberger que des systèmes d'exploitation s'exécutant sur la

5. Nous avons déjà parlé de *Rosetta* qui permet de traduire des instructions pour PowerPC en instruction pour processeur Intel. Un simulateur doit réaliser la traduction (par bloc d'instructions) du code que désire exécuter le noyau invité afin de lui faire correspondre les instructions du processeur physique sous-jacent. Ce mécanisme introduit nécessairement une perte de performances liée au temps consommé par les efforts de traduction.

même architecture matérielle que lui : l'ISA (*Instruction Set Architecture*) et donc les instructions qui seront exécutées sur le CPU doivent être identiques entre le système hôte et les systèmes invités.

Une fois ce pré-requis obtenu, le logiciel de virtualisation peut exécuter, sans avoir à les traduire, les instructions en provenance des systèmes invités. Nous savons toutefois que certaines instructions, dangereuses car intervenant sur le matériel, ne peuvent s'exécuter qu'en mode noyau. Leur demande d'exécution par le système invité entraîne donc une interruption puisque ce dernier tourne dans l'anneau de niveau 3. Le rôle du logiciel de virtualisation consiste à détecter ces interruptions et à réaliser les traductions ou les remplacements adéquats pour permettre à ces appels d'aboutir sans pour autant les exécuter sur le processeur. La figure 9.8 montre un exemple d'interception d'une interruption et de la traduction assortie.

Tous les accès au matériel virtuel, qu'il s'agisse naturellement du CPU mais aussi des différents périphériques présents, sont signalés par des erreurs d'instructions illégales dans l'anneau 3 et traduites pour être répercutées sur le matériel physique. Un logiciel de virtualisation doit donc être capable d'émuler (de simuler) un certain nombre de matériels, tels que contrôleurs SCSI, cartes graphiques, lecteur de DVD, disques durs, ... Il sera difficile de choisir par vous-même les « composants » de votre machine virtuelle.

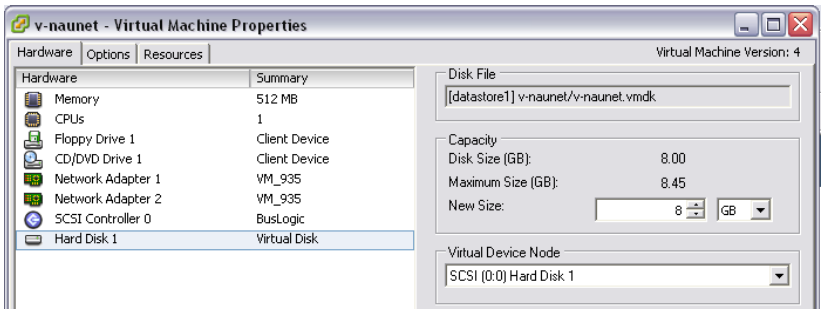


FIGURE 9.9 – *Un certain nombre de logiciels simulent la présence de matériel pour l'instance hébergé. Dans ce panneau de configuration de VmWare ESX, on note la présence d'un contrôleur SCSI sur lequel se situe le disque dur virtuel. Seuls deux types de contrôleur sont proposés.*

Tout serait relativement « simple » si toutes les instructions privilégiées s'exécutaient exclusivement dans l'anneau 0. Le logiciel de virtualisation réaliserait les actions adéquates pour chaque erreur. Hélas l'ISA des processeurs Intel révèle 17 instructions ne s'exécutant pas en mode noyau, mais en mode utilisateur (donc l'anneau 3), qui sont néanmoins critiques du point de vue d'un système d'exploitation (elles affectent des ressources matérielles). Ces instructions doivent donc être interceptées ce qui

signifie que le logiciel de virtualisation doit analyser l'intégralité des instructions. C'est une perte de performance. Nous verrons que l'`ISA` a su évoluer pour conduire à une virtualisation assistée par le matériel.

Outre cette analyse au coup par coup des 17 instructions ne déclenchant par d'erreur [6], le logiciel de virtualisation doit impérativement observer toutes les tentatives d'accès à la mémoire afin de les réinterpréter pour gérer correctement le partage de la mémoire entre les différents systèmes invités. Nous trouverons donc les pages mémoires doublement virtuelles des systèmes invités, puis la mémoire virtuelle du système hôte et enfin la mémoire physique. Une solution simple de gestion de la table des pages invitée est de la marquer en lecture seule afin d'intercepter les erreurs générées par tous les accès et de les réinterpréter. Nous avons là encore une baisse de performance.

Enfin il est important de « faire croire » aux systèmes invités que leur noyau s'exécute dans l'anneau de niveau 0. Toutes les instructions permettant de révéler l'anneau sous-jacent doivent donc être capturées et réinterprétées. C'est une nouvelle baisse de performance.

La virtualisation assistée par le matériel

Pour supprimer la nécessité d'une surveillance continuelle des instructions par le logiciel de virtualisation, Intel et AMD ont proposé dans leurs nouveaux processeurs une extension de l'`ISA` [7]. Dans ces instructions on trouve, chez Intel, `VMENTRY` et `VMLOAD, VMCALL` chez AMD. Un nouvel anneau est ajouté, ce qui porte leur nombre à 5. Le logiciel de virtualisation (le système d'exploitation de l'hôte) s'exécute dans ce nouvel anneau particulier, puis à l'aide des nouvelles instructions, charge le contexte d'exécution d'un système invité, bascule dans le mode virtuel et présente les quatre anneaux classiques d'exécution à ce système invité. La figure 9.10 présente cette démarche.

Le gain que procure cette technique est très conséquent puisque le système de contrôle de l'hôte n'a plus besoin d'examiner chaque instruction pour vérifier qu'elle ne fait pas appel au mode privilégié. Qui plus est, les systèmes d'exploitation invités ont l'illusion de s'exécuter en mode noyau, *i.e.* dans l'anneau de niveau 0.

La gestion de la mémoire virtuelle est elle aussi adressée par ce nouvel `ISA`. En règle générale, un système d'exploitation utilise la virtualisation de la mémoire (voir le chapitre 6). Ceci devrait *a priori* nous rassurer et nous pourrions penser de prime abord que le logiciel de virtualisation n'aura aucun effort supplémentaire à fournir pour que les systèmes invités accèdent à la mémoire. Cependant nous avons déjà évoqué le fait qu'un système d'exploitation charge généralement, voire toujours, son noyau à partir de l'adresse virtuelle la plus basse, *i.e.* celle d'adresse nulle. Cela signifie que

6. Le mécanisme permettant d'intercepter les instructions fautives est souvent noté « *trap and emulate virtualization* ».

7. Ils ont aussi proposé une nouvelle structure de contrôle, appelée `VMCS` pour *Virtual Machine Control Structure* chez Intel et `VMCB` pour *Virtual Machine Control Block* chez AMD.

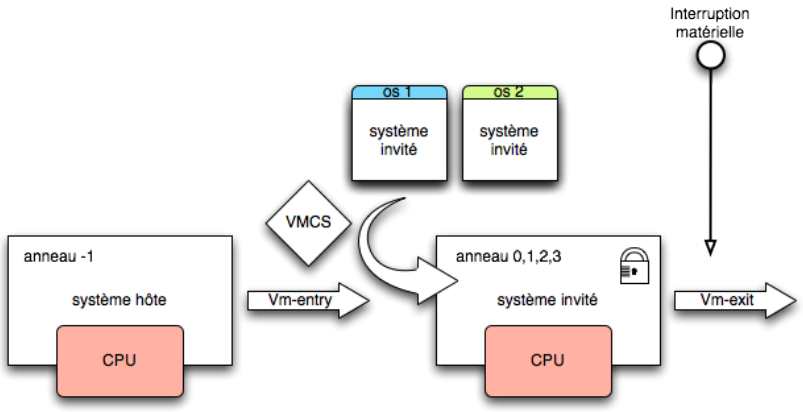


FIGURE 9.10 – *Un mode spécial, parfois noté anneau -1, permet l'exécution du système hôte. Par le biais d'instructions spécifiques, on entre dans un mode moins privilégié et on charge les différentes composantes liées au système invité qui va s'exécuter. Ce mode moins privilégié présente toutefois au système invité quatre anneaux de virtualisation ce qui évite le mécanisme du* trap and emulate. *La demande d'exécution d'une instruction de l'anneau 0 déclenchera alors une interruption matérielle qui sera gérée par le système hôte après avoir provoqué la sortie du mode virtuel.*

si rien n'est fait, tous les systèmes invités vont adresser directement la `MMU` de l'hôte en demandant la correspondance de l'adresse `0x0` dans la mémoire physique et cette correspondance étant bijective, les problèmes vont commencer...

Il est donc nécessaire que le logiciel de virtualisation bloque tous les appels des systèmes invités vers la `MMU`, réinterprète ces appels pour finalement interroger la `MMU` et retourner les adresses correctes. Cela se réalise généralement par le biais d'une table « ombre » des pages (traduction très approximative de `SPT` ou « *Shadow Page Table* » !). La figure 9.11 décrit ce fonctionnement.

Ce type de fonctionnement est complexe puisqu'il nécessite une double indirection et donc plus de calculs. De plus, lorsque le logiciel de virtualisation donne accès au `CPU` à un système invité il doit aussi charger la table ombre associée. Tout cela peut ralentir fortement les systèmes invités si ces derniers font très régulièrement des accès à la mémoire.

Afin de palier cette gestion complexe par le logiciel de virtualisation, Intel et AMD ont mis respectivement en place les `EPT` (« Extended Page Tables ») et les `NPT` (« Nested Page Tables »). Il s'agit ni plus ni moins de faire réaliser par le matériel une gestion qui était jusqu'à présent dévolue au logiciel de virtualisation. La mémoire dédiée auparavant pour le stockage des `SPT` est ainsi disponible pour le système. Lorsqu'un système invité requiert un accès à la mémoire, celui-ci est géré au travers de ces tables

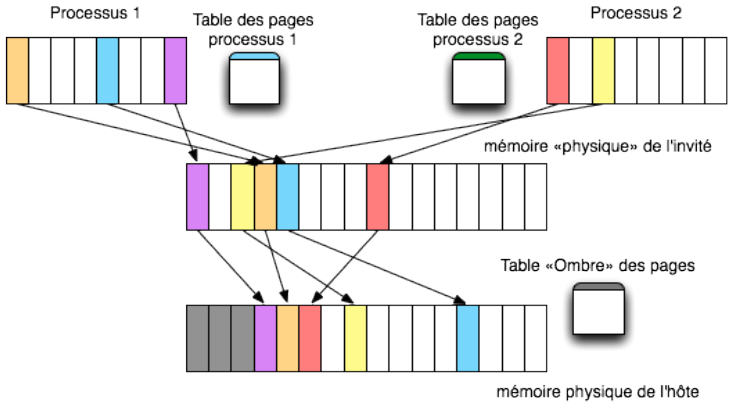


FIGURE 9.11 – *Chaque système invité place son noyau à l'adresse* `0x0`*. Il est donc indispensable que l'hôte détecte les accès à la mémoire « physique » d'un invité et traduise au moyen d'une table « ombre » des pages (associé à cet invité) ces accès vers la mémoire physique de la machine réelle.*

et tous les défauts de page [8] levés par l'hôte pourront être transmis au système invité. Ceci est une conséquence très bénéfique puisqu'elle permet de se départir d'un nombre conséquent de `VmEnter` / `VMExit` liés à la gestion de la mémoire.

D'autres améliorations matérielles ont permis une meilleure isolation des différentes machines virtuelles, notamment sur la gestion des périphériques et le `DMA` (« Direct Memory Access »). Le lecteur curieux trouvera sur la toile de nombreux articles traitant de ces techniques que nous n'aborderons pas dans le cadre de ce cours.

Toutes ces améliorations signent-elles le déclin de la para-virtualisation ? Examinons cette technique avant de nous prononcer !

La para-virtualisation

La para-virtualisation essaye de répondre à la virtualisation en modifiant les systèmes d'exploitation invités pour les faire dialoguer avec un hyperviseur qui règlera les accès aux matériels. Ce faisant, cette technique présente l'avantage majeure d'optimiser totalement le dialogue entre les systèmes invités (modifiés) et le matériel et doit donc offrir d'excellentes performances. L'inconvénient de cette technique réside essentiellement dans les modifications qu'il est nécessaire d'apporter aux systèmes d'exploitations que l'on désire inviter. Cela signifie que les différents appels système

8. Rappel : les défauts de page sont des interruptions qui signalent que l'accès à la mémoire a échoué et que le système d'exploitation doit intervenir pour traiter cette interruption, soit pour fournir de nouvelles pages, soit pour tuer un processus réalisant une lecture / écriture interdite.

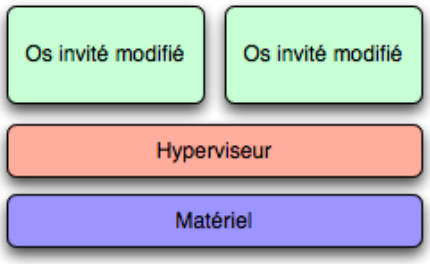


FIGURE 9.12 – *On parle très souvent d'hyperviseur de machines virtuelles pour qualifier la couche intercalée entre les systèmes para-virtualisés et le matériel. Remarquons qu'à la différence des autres techniques de virtualisation il n'y a ni système d'exploitation ni logiciel de virtualisation sur le socle. C'est un des atouts majeurs de la para-virtualisation*

doivent être remplacés par ceux intégrés dans une `API` (« *Advanced Program Interface* » ou interface de programmation avancée). Ces nouveaux appels sont parfois dénommés `hypercall`.

Deux acteurs dominent le marché de la para-virtualisation, Citrix avec Xen (XenServer est une solution libre et gratuite, mais il existe des solutions payantes telles que XenServer Advanced Edition. . .) et Microsoft avec Hyper-V (solution fermée et payante). Notons au passage que l'`API` d'Hyper-V nécessaire aux systèmes invités est passée sous licence GPL [9] afin de pouvoir être intégrée dans des systèmes de type Unix / Linux.

Un des inconvénients majeures de la para-virtualisation est la mise au point de différents drivers matériels permettant de construire la bibliothèque d'appels d'un matériel spécifique tel qu'un contrôleur `SCSI` ou une carte vidéo, et son intégration dans le système d'exploitation invité.

Mais ce pré-requis tend naturellement à disparaître avec les nouveaux `ISA` puisque les appels au système des invités sont maintenant associés à un mode particulier du processeur à la différence des instructions réalisées par l'hyperviseur. Il n'est plus nécessaire de les traduire, que ce soit dans le logiciel de virtualisation (comme nous l'avons vu précédemment) ou dans l'hyperviseur. La différence entre para-virtualisation et virtualisation complète se réduit donc peu à peu.

9. Il s'agit de la *GNU Public Licence* très connue du logiciel libre.

9.3 Conclusion

Nous conclurons en élargissant ce que nous venons de voir au domaine de la sécurité des techniques de virtualisation. L'intérêt de l'utilisateur (particulier, entreprise) est d'éviter qu'une attaque unique sur l'hôte mette à mal l'intégralité des systèmes invités et de façon duale c'est sur ce point que se concentrera un pirate informatique afin de réduire d'une part la masse de systèmes à analyser pour réaliser une compromission (les systèmes invités peuvent être relativement disparates ce qui dans le cas présent est une qualité !). Les différentes techniques ne se comportent pas de la même façon vis à vis de ces attaques (nous évoquerons principalement le `DoS` ou *Deny of Service*).

Un isolateur est avant tout un système d'exploitation classique et possède nécessairement un point d'entrée pour permettre l'administration des systèmes emprisonnés. À ce titre il est particulièrement vulnérable à une attaque par saturation de ses connexions réseau ainsi qu'aux failles de sécurité du système d'exploitation de l'hôte et des services qui s'y trouvent (`ssh`, `dns`, ...).

La virtualisation complète, la virtualisation assistée par le matériel et la para-virtualisation n'offrent pas d'accès direct au socle. Ceci est particulièrement vrai pour la para-virtualisation puisque l'hyperviseur n'est pas accessible, ce n'est qu'une instance virtuelle particulère (appelé le domaine 0) qui permet la gestion des autres instances. De la même façon la virtualisation complète, même si elle s'appuie sur un système d'exploitation, n'est pas directement accessible. Provoquer un déni de service du domaine 0 (dans le cas de Xen) ou de l'interface d'administration (pour Vmware) ne permet pas de compromettre les machines virtuelles.

L'autre point crucial est l'isolement entre les différentes machines virtuelles. Si l'on ne prend pas certaines précautions, sortir d'une prison est relativement simple (il suffit de créer un périphérique disque dur identique au disque de l'hôte et de monter ce disque dur pour accéder à l'intégralité du monde extérieur !). Un certain nombre de failles de sécurité ont été trouvées tant dans le domaine de la para-virtualisation (possibilité de s'échapper d'une machine virtuelle pour aller sur l'hyperviseur ou l'unité de gestion des systèmes invités) que de la virtualisation (utilisation de certains outils installés dans les systèmes invités ou contournement des sécurités des outils d'administration).

Mais ces failles tendent à devenir de moins en moins nombreuses avec l'utilisation des technologies matérielles fournies avec les nouveaux processeurs. Il faut cependant garder à l'esprit que la compromission d'un socle donne la main sur toutes les machines virtuelles et qu'il semble donc naturel d'adopter les pratiques usuelles de séparation des services (web d'un côté, messagerie de l'autre, ...) – utilisées anciennement sur les machines physiques – sur les machines virtuelles.

Deuxième partie

Programmation système en C sous Unix

10

Les processus sous Unix

10.1 Introduction

Les ordinateurs dotés de systèmes d'exploitation modernes peuvent exécuter « en même temps » plusieurs programmes pour plusieurs utilisateurs différents : ces machines sont dites multi-tâches (voir chapitre sur les processus) et multi-utilisateurs. Un processus (ou *process* en anglais) est un ensemble d'instructions se déroulant séquentiellement sur le processeur : par exemple, un de vos programmes en `C` ou le terminal de commande (*shell* en anglais) qui interprète les commandes entrées au clavier.

Sous Unix, la commande **`ps`** permet de voir la liste des processus existant sur une machine : **`ps -xu`** donne la liste des processus que vous avez lancé sur la machine, avec un certain nombre d'informations sur ces processus. En particulier, la première colonne donne le nom de l'utilisateur ayant lancé le processus et la dernière le contenu du tableau `argv` du processus. **`ps -axu`** donne la liste de tout les processus lancés sur la machine.

Chaque processus, à sa création, se voit attribuer un numéro d'identification (le *pid*). C'est ce numéro qui est utilisé ensuite pour désigner un processus. Ce numéro se trouve dans la deuxième colonne de la sortie de la commande **`ps -axu`**.

Un processus ne peut être créé qu'à partir d'un autre processus (sauf le premier, *init*, qui est créé par le système au démarrage de la machine). Chaque processus a donc un ou plusieurs fils, et un père, ce qui crée une structure arborescente. À noter que certains systèmes d'exploitation peuvent utiliser des processus pour leur gestion interne, dans ce cas le *pid* du processus *init* (le premier processus en mode utilisateur) sera supérieur à 1.

10.2 Les entrées / sorties en ligne

La façon la plus simple de communiquer avec un processus consiste à passer des paramètres sur la ligne de commande au moment où l'on exécute le programme associé. Par exemple, lorsque l'on tape des commandes comme **`ps -aux`** ou **`ls -alF`**, les paramètres en ligne « -aux » et « -alF » sont respectivement transmis aux programmes

ps et **ls**. Ce principe est très général et peut s'étendre à tout type et tout nombre de paramètres : **xeyes -fg red -bg gray -center yellow -geometry 300x300**

À la fin de son exécution, chaque processus renvoie une valeur de retour qui permet notamment de déterminer si le programme a été exécuté correctement ou non. Cette valeur de retour, que l'on nomme parfois pas abus de langage le code d'erreur, est nécessairement un entier et, par convention, une valeur nulle de retour indique que l'exécution du programme a été conforme à ce qui était attendu.

La valeur de retour d'un programme donné (par exemple **ls**) est transmise au programme appelant qui a déclenché l'exécution du programme appelé (**ls**). En général, le programme appelant est un terminal de commande (un *shell* en anglais), mais cela peut être un autre programme comme nous le verrons plus tard. Dans le cas du terminal de commande, il est très simple de visualiser la valeur de retour grâce à la variable [1] `status` (via la commande qui est notée **echo $?** en `bash` et **echo $status** en `csh`), comme le montre l'exemple ci-dessous :

```
menthe22> cd /usr/games/
menthe22> ls
banner         netrek.paradise        runzcode        trojka
christminster  paradise.sndsrv.linux  sample.xtrekrc  xzip
fortune        pinfo                  scottfree
menthe22> ls banner
banner
menthe22> echo $?
0
menthe22> ls fichier_qui_n_existe_pas
ls: fichier_qui_n_existe_pas: No such file or directory
menthe22> echo $?
1
```

Lorsque les programmes concernés sont écrits en `C`, les entrées / sorties en ligne s'effectuent respectivement via les fonctions `main()` et `exit()` que nous allons décrire dans la suite du chapitre.

Attention, ce type d'entrées / sorties peut passer au premier abord comme une simple fonctionnalité du langage `C`, mais nous verrons dans le chapitre suivant qu'elles reflètent la façon dont le système d'exploitation Unix crée des processus.

La fonction `main()`

La fonction `main()` est la fonction appelée par le système [2] après qu'il a chargé le programme en mémoire : c'est le point d'entrée du programme. Elle se déclare de la manière suivante :

1. Un terminal de commande utilise pour son fonctionnement un certain nombre de variables et il est capable d'interpréter un grand nombre de commande qui lui sont propres (comme **history**, par exemple), c'est-à-dire qui ne sont pas des programmes indépendants. Ceci permet notamment de programmer les terminaux de commande à partir de scripts appelés *shell scripts*. L'utilisation en détail de ces commandes et de ces variables peut être trouvée dans le manuel en ligne du terminal (**man csh** ou **man bash** par exemple) et n'a que peu d'intérêt ici. Il faut simplement retenir qu'il est possible d'accéder à la valeur de retour du programme exécuté.
2. Le système appelle en fait une autre fonction qui elle-même appellera la fonction `main()`.

```
int main(int argc, char *argv[])
```

La ligne ci-dessus représente le prototype de la fonction `main()`, c'est-à-dire la déclaration :

– du nombre d'arguments qu'admet la fonction ;
– du type de ces arguments ;
– du type de la valeur de retour que renvoie la fonction.

En l'occurrence, le prototype de la fonction `main` est imposé par le système (et par le langage `C`) et il n'est donc pas possible d'en utiliser un autre : la fonction `main` prend nécessairement deux arguments, un de type `int` et un de type `char **`, et elle retourne nécessairement un entier.

La valeur de retour de la fonction `main` est renvoyée grâce à un appel à la fonction `exit()` (qui est abordée plus loin). Même s'il est possible d'utiliser l'instruction `return` à cet effet, il est demandé par convention de n'utiliser que la fonction `exit()` (l'instruction `return` étant utilisée pour transmettre les valeurs de retour des fonctions autre que la fonction `main()`).

La fonction `main()` accepte deux prototypes différents, selon que l'on souhaite ou non passer des arguments :

```
int main(int argc, char **argv)
...
int main(void)
```

Les noms des variables utilisées dans le premier cas ne sont pas imposées par le système ou le langage. Il est donc possible de choisir d'autres noms que `argc` et `argv`. Néanmoins, ces noms sont systématiquement utilisés par convention et ils permettent de lire facilement un programme en `C` en sachant à quoi ils font référence (en l'occurrence aux paramètres de la ligne de commande).

La variable `argc` contient le nombre de paramètres passés sur la ligne de commande, sachant que le nom du programme compte comme un paramètre. Par exemple, si l'on tape la commande **`./a.out param`** dans un terminal de commande, la variable `argc` du programme `a.out` vaudra 2. Si on tape la commande **`./a.out param1 param2`**, la variable `argc` vaudra 3.

La variable `argv` est un tableau de chaînes de caractères contenant les paramètres passés au programme. La taille du tableau `argv` est donnée par la variable `argc`. Par exemple, si l'on tape la commande **`./a.out param`** dans un terminal de commande, la variable `argc` vaudra 2, `argv[0]` contiendra la chaîne de caractères `"./a.out"` (notez la présence des guillemets signifiant qu'il s'agit d'une chaîne de caractères) et `argv[1]` contiendra la chaîne de caractères `"param"`. Si on tape la commande **`./a.out param1 param2`**, la variable `argc` vaudra 3, `argv[0]` contiendra la chaîne de caractères `"./a.out"`, `argv[1]` contiendra la chaîne de caractères `"param1"` et `argv[2]` contiendra la chaîne de caractères `"param2"`.

Lors du passage de paramètres en ligne, l'utilisateur (i.e. le code du programme appelé) n'a rien à faire avant d'utiliser les variables `argc` et `argv` : c'est le programme

appelant (en général, le terminal de commande) qui va recopier chaque paramètre de la ligne de commande dans un élément du tableau `argv` avant d'exécuter le programme. Par exemple, lorsqu'on tape **`./a.out param`** dans un terminal de commande, c'est ce terminal qui va recopier les paramètres `"./a.out"` et `"param"` dans la variable `argv` du programme `a.out`.

Insistons sur le fait que les paramètres passés sur la ligne de commande sont de type `char *`, ce sont donc des chaînes de caractères (puisque `argv` est un tableau de chaînes de caractères). Cela signifie en particulier qu'il n'est pas possible de passer directement un paramètre numérique sur la ligne de commande : celui-ci sera transformé en chaîne de caractères et il faudra effectuer la transformation inverse dans le programme (différence entre la chaîne de caractères "2", le caractère '2' et l'entier 2).

Pour le terminal de commande, les paramètres sont des mots, c'est-à-dire des groupes de caractères séparés par des espaces. Il est cependant possible d'avoir des espaces dans un mot : en utilisant les caractères `\` ou `"`. Par exemple, dans `./a.out param1 param2`, `a.out` verra trois paramètres alors qu'il n'en verra que deux dans `./a.out "param1 param2"`.

Voici un petit programme d'exemple (que nous appellerons `ex11`) qui se contente d'afficher les valeurs de `argc` et `argv` :

Listing 10.1 – *Affichage des arguments et de leur nombre*

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;
        printf("nombre d'arguments: %d\n", argc);
        for (i=0; i<argc; i++) {
                printf("argument %d: <%s>\n", i, argv[i]);
        }
        printf("\n");
        exit(EXIT_SUCCESS);
}
```

L'exécution donne :

```
menthe22> ./ex11
nombre d'arguments: 1
argument 0: <./ex11>

menthe22> ./ex11 arg1
nombre d'arguments: 2
argument 0: <./ex11>
argument 1: <arg1>

menthe22> ./ex11 arg1 arg2
nombre d'arguments: 3
argument 0: <./ex11>
argument 1: <arg1>
argument 2: <arg2>

menthe22> ./ex11 plein de parametres en plus
```

```
nombre d'arguments: 6
argument 0: <./ex11>
argument 1: <plein>
argument 2: <de>
argument 3: <parametres>
argument 4: <en>
argument 5: <plus>

menthe22> ./ex11 "plein de parametres en plus"
nombre d'arguments: 2
argument 0: <./ex11>
argument 1: <plein de parametres en plus>

menthe22> ./ex11 plein de "parametres en" plus
nombre d'arguments: 5
argument 0: <./ex11>
argument 1: <plein>
argument 2: <de>
argument 3: <parametres en>
argument 4: <plus>
```

La manière dont les paramètres sont passés du programme appelant (ici le *shell*) au programme appelé sera vu plus en détail au cours du TP Recouvrement de processus sous Unix.

La fonction `exit()`

Les paramètres de la fonction `main()` permettent au programme appelant de passer des paramètres au programme appelé. La fonction `exit()` permet au programme appelé de retourner un paramètre au programme appelant. La fonction `exit()` termine le programme et prend comme paramètre un entier signé qui pourra être lu par le programme appelant. La fonction `exit()` ne retourne donc jamais et toutes les lignes du programme situées après la fonction `exit()` ne servent donc à rien.

Traditionnellement, un `exit(0)` signifie que le programme s'est exécuté sans erreur, alors qu'une valeur non-nulle signifie autre chose (par exemple qu'une erreur est survenue). Deux valeurs sont définies dans `/usr/include/stdlib.h` :

- `EXIT_SUCCESS` qui vaut 0 ;
- `EXIT_FAILURE` qui vaut 1.

Différentes valeurs peuvent désigner différents types d'erreurs (voir le fichier `/usr/include/sysexits.h`).

10.3 Les fonctions d'identification des processus

Ces fonctions sont au nombre de deux :

- `pid_t getpid(void)`
  Cette fonction retourne le *pid* du processus.
- `pid_t getppid(void)`
  Cette fonction retourne le *pid* du processus père. Si le processus père n'existe plus (parce qu'il s'est terminé avant le processus fils, par exemple), la valeur

retournée est celle qui correspond au processus *init* (en général 1), ancêtre de tous les autres et qui ne se termine jamais.

Le type `pid_t` est en fait équivalent à un type `int` et a été défini dans un fichier d'en-tête par un appel à `typedef`.

Le programme suivant affiche le *pid* du processus créé et celui de son père :

Listing 10.2 – *`pid` du processus et de son père*

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    printf("mon pid est : %d\n", (int)getpid());
    printf("le pid de mon pere est : %d\n", (int)getppid());
    exit(EXIT_SUCCESS);
}
```

La commande **`echo $$`** tapée au clavier retourne le *pid* du *shell* interprétant cette commande. Le programme suivant peut donc donner le résultat suivant (les numéros de processus étant uniques, deux exécutions successives ne donneront pas le même résultat) :

```
menthe22>echo $$
189
menthe22>./ex21
mon pid est : 3162
le pid de mon pere est : 189
```

10.4 Exercices

Question 1

Écrire une application qui affiche les paramètres passés sur la ligne de commandes et qui retourne le nombre de ces paramètres. Vérifier le résultat grâce à la variable du *shell* qui stocke la valeur de retour (`status` en **`csh`**, `?` en **`bash`**).

Question 2

Écrire une application qui additionne les nombres placés sur la ligne de commande et qui affiche le résultat. On nommera ce programme **`addition`**.

Question 3

Écrire un programme affichant son *pid* et celui de son père. On nommera ce programme **`identite`**.

Question 4

Reprendre le programme **`identite`** et ajouter un appel à la fonction `sleep()` pour attendre 10 secondes avant d'exécuter les appels `getpid()` et `getppid()`.

– `unsigned int sleep(unsigned int s)` suspend l'exécution du programme pendant `s` secondes.

Vérifier que le terminal de commande où le programme est lancé est bien le père du processus correspondant. Relancer ensuite le programme, mais en tâche de fond et en redirigeant la sortie dans un fichier :

```
./exo >sortie &
```

et fermer la fenêtre en tapant **`exit`**. Attendre 10 secondes et regarder le contenu du fichier contenant la sortie du programme. Remarques ?

10.5 Corrigés

Listing 10.3 – *Solution de la question 1*

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        int i, s;
        s=0;
        for (i=1; i<argc; i++) {
                s = s + atoi(argv[i]);
        }
        printf("la somme est: %d\n", s);
        exit(EXIT_SUCCESS);
}
```

Listing 10.4 – *Solution de la question 2*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    sleep(10);
    printf("mon pid est: %d\n", getpid());
    printf("le pid de mon pere est: %d\n", getppid());
    exit(EXIT_SUCCESS);
}
```

Lorsque le père n'existe plus (après avoir tapé `exit()` dans le *shell*), la fonction `getppid()` retourne 1.

10.6 Corrections détaillées

Comme cela est rappelé au début du chapitre, les entrées les plus simples que l'on puisse faire dans un programme consistent à passer des arguments sur la « ligne de commandes ».

La ligne de commande est tout simplement l'ensemble des caractères que l'on tape dans une console, *i.e.* un terminal ou encore une `xterm`. La ligne de commandes est gérée par le *shell*, c'est-à-dire l'interprète de commandes. Comme son nom l'indique il interprète les caractères tapés avant de les transmettre à la commande invoquée. Vous avez tous déjà remarqué que lorsque l'on tape :

```
menthe22> ls *.tif
 toto1.tiff toto2.tif toto3.tif
menthe22>
```

l'interprète s'est chargé d'interpréter le caractère $*$ pour ensuite appeler la commande **`ls`** avec les arguments `toto1.tif`, `toto2.tif` et `toto3.tif`. Nous allons donc essayer de constater cela avec le premier programme.

Arguments transmis au programme

Le programme fait appel à deux fonctions `printf()` dont le prototype est dans `stdio.h` et `exit()` dont le prototype est dans `stdlib.h`, ce qui explique les deux directives d'inclusions.

Listing 10.5 – *Affichage des arguments passés en ligne de commandes*

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i ;
    printf("nombre d'arguments : %d\n", argc) ;
    for (i=0 ; i<argc ; i++)
        printf("argument %d : <%s>\n", i, argv[i]) ;

    printf("\n") ;
    exit(argc) ;
}
```

Nous compilons ce programme pour obtenir l'exécutable **`showarg`** que nous allons essayer sur le champ :

```
menthe22> showarg *.tif
 nombre d'arguments : 4
 argument 0 : showarg
 argument 1 : toto1.tif
 argument 2 : toto2.tif
 argument 3 : toto3.tif
menthe22>
```

L'interprète de commandes a remplacé le métacaractère par la liste des fichiers correspondant et ce n'est pas 2 arguments qu'a reçus le programme (`showarg` et `*.tif`) mais bel et bien 4.

Nous pouvons alors demander à l'interprète de ne pas jouer son rôle et de ne rien interpréter en protégeant le caractère spécial * grâce à l'antislash :

```
menthe22> showarg \*.tif
 nombre d'arguments : 2
 argument 0 : showarg
 argument 1 : *.tif
menthe22>
```

Si l'on veut maintenant regarder comment sont construits les arguments, nous voyons que c'est l'espace qui sert de séparateur entre les mots de la ligne de commandes. Prenons de nouveau un exemple :

```
menthe22> showarg le\ premier puis le\ deuxieme
 nombre d'arguments : 4
 argument 0 : showarg
 argument 1 : le premier
 argument 2 : puis
 argument 3 : le deuxieme
menthe22>
```

Comme on le remarque facilement, l'espace séparant le mot `le` du mot `premier` est précédé d'un antislash. Cet espace est donc protégé et l'interprète de commandes n'essaye donc pas de l'interpréter comme un séparateur. Ainsi pour lui, le premier argument après le nom du programme est bel et bien `le premier`.

Nous pouvons aussi protéger les différents arguments de l'interprétation du *shell* en les « entourant » par des guillemets, (*i.e.* des « double quote ») :

```
menthe22> showarg "le premier" puis "le deuxieme"
 nombre d'arguments : 4
 argument 0 : showarg
 argument 1 : le premier
 argument 2 : puis
 argument 3 : le deuxieme
menthe22>
```

Le fait de passer comme premier argument de la liste le nom du programme peut sembler un peu futile, et pourtant cela permet parfois de réunir plusieurs programmes en un seul !

Prenons l'exemple suivant :

Listing 10.6 – *Le premier argument est utile !*

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int i ;
    if (strstr(argv[0],"showarg")!=NULL) {
```

```
      for (i=0 ; i<argc ; i++)
        printf("argument %d : <%s>\n", i, argv[i]) ;

      printf("\n") ;
    } else if (strstr(argv[0],"countarg")!=NULL) {
      printf("nombre d'arguments : %d\n", argc) ;
    }
    exit(EXIT_SUCCESS) ;
}
```

Dans ce programme on utilise la fonction `strstr()` pour savoir si `argv[0]` contient **`showarg`** ou **`countarg`** et ainsi déterminer sous quel nom le programme a été exécuté. Nous allons compiler ce programme deux fois avec deux sorties différentes (il existe une façon plus élégante [3]) :

```
menthe22> gcc -Wall -o countarg multiprog.c
menthe22> gcc -Wall -o showarg multiprog.c
```

Nous avons maintenant à notre disposition deux programmes, issus du même code source, dont le fonctionnement n'est pas le même.

```
menthe22> countarg "le premier" puis "le deuxieme"
 nombre d'arguments : 4
menthe22> showarg "le premier" puis "le deuxieme"
 argument 0 : showarg
 argument 1 : le premier
 argument 2 : puis
 argument 3 : le deuxieme
menthe22>
```

Ce genre de procédé est couramment utilisé, sinon comment expliquer qu'un programme de compression de fichier ait exactement la même taille qu'un programme de décompression :

```
menthe22> ls -l /bin/gzip /bin/gunzip
-rwxr-xr-x  3 root root 47760 dec 7 1954 /bin/gunzip
-rwxr-xr-x  3 root root 47760 dec 7 1954 /bin/gzip
```

Argument renvoyé à l'interprète

De la même manière que l'interprète de commandes passe des arguments au programme, le programme annonce à l'interprète comment s'est déroulée son exécution. C'est le rôle de l'appel à la fonction `exit()` qui permet de transmettre un nombre à l'interprète de commandes.

Par convention sous Unix, lorsqu'un programme se termine de façon satisfaisante, *i.e.* il a réussi à faire ce qu'on lui demandait, ce programme transmet une valeur nulle, **`0`**, à l'interprète. Selon l'interprète, cette valeur est placée dans une variable spéciale appelée `status` (`csh`, `bash`,...) ou `?` sous `bash`. Reprenons notre programme, il transmet à l'interprète le nombre d'arguments qu'il a reçus (`exit(argc)`) :

3. La solution élégante est de créer un lien symbolique : **`gcc -Wall -o countarg multiprog.c`** puis **`ln -s countarg showarg`**

```
menthe22> showarg *.tif
 nombre d'arguments : 4
 argument 0 : showarg
 argument 1 : toto1.tif
 argument 2 : toto2.tif
 argument 3 : toto3.tif
menthe22>echo $?
 4
menthe22>
```

Essayons maintenant avec un autre programme bien connu, **ls**, en lui demandant de lister un fichier inexistant :

```
menthe22> ls ~/corrige/pasmoi.c
ls: ~/corrige/pasmoi.c:  No such file or directory
menthe22>echo $?
 1
menthe22>
```

On constate que la commande **ls** a retourné la valeur **1** à l'interprète, ce qui signifie qu'il y a eu une erreur. Ainsi même si je suis incapable de lire ce qu'une commande affiche, je peux toujours contrôler par l'intermédiaire de son code de retour si elle s'est bien déroulée.

Deux valeurs sont définies par défaut dans le fichier `stdlib.h` :

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Ceci étant rien ne vous empêche de définir vos propres codes d'erreur, mais gardez en tête qu'une exécution sans problème doit retourner une valeur nulle à l'interprète. Il convient toutefois de faire attention ce que nous allons voir en examinant les quelques résultats qui suivent.

Prenons le programme très simple suivant :

Listing 10.7 – *Examiner le comportement de sortie*

```
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  exit(argc>1?atoi(argv[1]):0) ;
}
```

Ce programme retourne la valeur `0` si on ne lui donne aucun autre argument que le nom du programme et il retourne le deuxième argument converti en entier si cet argument existe. Une version moins dense pourrait être :

Listing 10.8 – *Version complète pour examiner la sortie*

```
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
```

```
  int val;
  if (argc > 1) {
    val = atoi(argv[1]);
  } else {
    val = 0;
  }
  exit(val) ;
}
```

Examinons ce que l'interprète réceptionne comme valeur de retour :

```
menthe22>./myexit
menthe22>echo $?
0
menthe22>./myexit 234
menthe22>echo $?
234
menthe22>./myexit 256
menthe22>echo $?
0
menthe22>./myexit 260
menthe22>echo $?
4
menthe22>./myexit -6
menthe22>echo $?
250
```

Le code de retour d'un programme est en fait codé sur un seul octet et ne peut donc prendre que des valeurs comprises entre `0` et `255`. Un code retour de `-1` est donc équivalent à un code de `255`.

Addition de nombres en ligne

Le meilleur moyen de transmettre n'importe quel style d'arguments à un programme quand on est un interprète, aussi bien des mots que des nombres entiers ou des réels, c'est bel et bien de tout transmettre sous la forme de mots, à savoir de chaîne de caractères. Rappelons brièvement par un petit dessin (fig. 10.6) les principales différences entre un caractère, un tableau de caractères représentant un mot et un nombre entier.

Donc pour additionner les arguments de la ligne de commandes (sauf le premier qui sera le nom du programme) il est impératif de convertir ces mots en entiers. Cela nous oblige donc à faire appel à la fonction `atoi()` en remarquant toutefois dans la page de **`man`** que cette fonction ne détecte pas les erreurs. Le programme aura la structure suivante :

Listing 10.9 – *Addition des arguments*

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i ;
    int tmp;
    int s ;
```

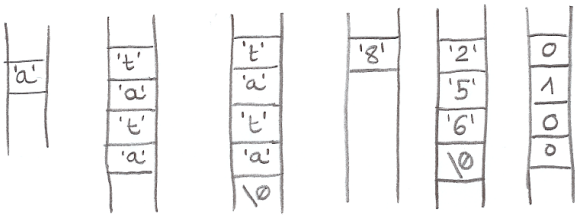


FIGURE 10.1 – *Un caractère, un tableau de caractères, un mot, le caractère* `'8'`*, le mot* `'256'` *et un entier valant 256. Le caractère consiste en un seul octet dans la mémoire, peu importe ce qu'il y a après ou avant. Il en va de même pour le tableau de caractères, peu importe ce qui précède les 4 caractères* `'tata'` *ou ce qui suit. Le troisième schéma correspond par contre à un tableau représentant un mot et on distingue clairement la différence : la présence d'un caractère un peu spécial,* `\0`*, qui permet de faire savoir à des fonctions comme* `printf()` *ou encore* `strlen()` *où s'arrête le mot. Les trois derniers schémas insistent sur la différence entre le caractère* `'8'` *qui est un symbole de notre alphabet, le mot* `'256'` *et le nombre 256 qui est égal à* 2^8 *et qui est donc représenté sur 4 octets par la suite* `0100` *(à lire avec l'octet de poids faible à gauche !).*

```
    for (i=1,s=0;i<argc;i++) {
        s += (tmp = atoi(argv[i]));
        if (i<argc-1) printf("%d + ",tmp) ;
        else printf("%d = ",tmp) ;
    }
    printf("%d\n",s) ;
    exit(EXIT_SUCCESS) ;
}
```

Son exécution dans un contexte normal donne :

```
menthe22>addarg 34 56 78
34 + 56 + 78 = 168
```

Dans un contexte moins glorieux nous pourrions avoir des surprises :

```
menthe22>addarg 10 10.4 10.1
10 + 10 + 10 = 30
```

Enfin si nous poussons le programme dans ses retranchements :

```
menthe22>addarg azeze erer 45
0 + 0 + 45 = 45
```

11

Les entrées / sorties sous Unix

Dans cette partie, nous décrivons les outils que le noyau du système d'exploitation met à la disposition de l'utilisateur pour manipuler des fichiers. Pour utiliser un service pris en charge par le noyau du système d'exploitation, l'utilisateur doit utiliser un « appel système » et il est capital de bien distinguer les appels système des fonctions de bibliothèques.

Nous parlerons ainsi des appels systèmes associés à l'utilisation de fichiers dans un premier temps. Puis les fonctions de la bibliothèque standard d'entrées / sorties ayant des fonctionnalités comparables seront décrites plus loin.

Si l'on se restreint aux appels système, toutes les opérations détaillées sont des opérations « bas niveau », adaptées à certains types de manipulation de fichiers, comme par exemple des écritures brutales (i.e. sans formatage) de données stockées de façon contiguë en mémoire.

En marge de cette introduction, il est bon de mentionner une fonction qui nous sera particulièrement utile dans l'intégralité des travaux dirigés : `perror()`. La documentation peut être consultée de la manière habituelle par `man perror`, mais en voici les grandes lignes.

```
void perror(const char *s);
```

Cette fonction affiche sur la sortie standard d'erreur le message d'erreur consigné par le système dans une variable globale. Lorsqu'un appel au système produit une erreur, le système consigne un numéro d'erreur dans une variable globale. Il existe un tableau associant ces numéros à un message (ce qui permet un affichage plus explicite que « l'erreur 12345 est survenue »). Si la chaîne de caractère « `s` » passée en paramètre à `perror()` n'est pas vide, `perror()` affiche tout d'abord la chaîne passée en paramètre, puis le symbole « `:` » et enfin le message d'erreur consigné par le système. En considérant Le fragment de code qui suit :

```
...
if ((fd = open(...)) == -1) {
        perror("Mon prog s'arrete la!");
        exit(EXIT_FAILURE);
}
...
```

un programme l'utilisant et révélant une erreur donnerait l'affichage suivant :

```
moi@moi> ./monprogramme
Mon prog s'arrete la: file not found
moi@moi>
```

L'utilisation de `perror()` est utile pour révéler la raison d'un échec lors d'un appel système et bien souvent les pages de manuel offriront une ligne consacrée à son utilisation comme ici l'extrait du manuel de `open()` :

> *If successful, open() returns a non-negative integer, termed a file descriptor.*
> *It returns -1 on failure, and sets errno to indicate the error.*

La variable globale consignant le numéro de l'erreur est `errno`. On comprend donc que toute erreur provoquée par une mauvaise utilisation de la fonction `open()` pourra être affichée sur la sortie standard d'erreur en utilisant `perror()`.

11.1 Les descripteurs de fichiers

Historiquement, les appels système associés aux fichiers étaient dédiés à la communication de l'ordinateur avec ses périphériques (écran, clavier, disque, imprimante, etc.). Sur les systèmes modernes, les différents périphériques sont gérés par le système lui-même, qui fournit à l'utilisateur une interface abstraite, un **descripteur de fichier** (*file descriptor* en anglais) pour accéder aux fichiers.

Cette interface se présente pour l'utilisateur sous la forme d'un simple entier : le système d'exploitation tient en fait à jour une table [1], appelée table des fichiers ouverts, où sont référencés tous les fichiers utilisés, c'est-à-dire tous les fichiers en train d'être manipulés par un processus (création, écriture, lecture), et l'entier mis à disposition de l'utilisateur correspond au numéro de la ligne de la table faisant référence au fichier concerné.

Le mot fichier ne doit pas être compris ici au sens « fichier sur le disque dur », mais comme une entité pouvant contenir ou transmettre des données. Un descripteur de fichier peut aussi bien faire référence à un fichier du disque dur, à un terminal, à une connexion réseau ou à un lecteur de bande magnétique.

Un certain nombre d'opérations génériques sont définies sur les descripteurs de fichier, qui sont ensuite traduites par le système en fonction du périphérique auquel se rapporte ce descripteur. Ainsi, écrire une chaîne de caractères à l'écran ou dans un fichier se fera – pour l'utilisateur – de la même manière. D'autres opérations sont spécifiques au type de descripteur de fichier.

11.2 Les appels système associés aux descripteurs de fichier

Les déclarations des appels système décrits dans cette section se trouvent dans les fichiers suivants :

1. En pratique, le système utilise une table par processus.

```
/usr/include/unistd.h
/usr/include/sys/types.h
/usr/include/sys/stat.h
/usr/include/fcntl.h
```

Pour utiliser ces appels système dans un programme `C`, il est donc nécessaire d'inclure ces fichiers en utilisant la directive `#include` (notez l'utilisation des caractères $<$ et $>$ qui indique au compilateur que ces fichiers se trouvent dans `/usr/include`) :

Listing 11.1 – *Inclusion des fichiers de déclaration*

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
        ...

        exit(EXIT_SUCCESS);
}
```

L'appel système `open()`

L'appel système `open()` permet d'associer un descripteur de fichier à un fichier que l'on souhaite manipuler. En cas de succès, le système d'exploitation va créer une référence dans la table des fichiers et va indiquer l'entier correspondant. Une fois référencé dans la table des fichiers, le fichier est dit « ouvert ».

L'appel système `open()` s'utilise de la manière suivante :

```
int open(const char *path, int flags, mode_t mode)
```

En cas de succès, l'appel `open()` retourne un descripteur de fichier qui pourra être ensuite utilisé dans le programme afin de désigner le fichier fraîchement ouvert. En cas d'erreur, par exemple lorsque le fichier désigné n'existe pas, `open()` retourne -1.

Le paramètre `path` est une chaîne de caractères donnant le chemin d'accès du fichier. Le mot clé `const` précise que la valeur pointée par le pointeur `path` ne sera pas modifiée par la fonction `open()` (ce qui serait possible, puisque c'est un pointeur !). Pour nos travaux pratiques, `path` désignera le plus souvent un nom de fichier du disque dur, par exemple `"/etc/motd"` ou `"/home/h82/in201/ex11.c"`.

Le paramètre `flags` détermine de quelle façon le fichier va être ouvert, c'est-à-dire quels types d'opérations vont être appliquées à ce fichier. À chaque type d'opération correspond un entier (`flags` est de type `int`) et, afin de rendre les programmes plus

lisibles et plus faciles à porter d'un système à l'autre, des constantes [2] sont utilisées pour symboliser ces valeurs entières :

- `O_RDONLY` pour ouvrir un fichier en lecture seule ;
- `O_WRONLY` pour ouvrir un fichier en écriture seule ;
- `O_RDWR` pour ouvrir un fichier en lecture/écriture ;
- `O_APPEND` pour ne pas écraser un fichier existant et placer les données qui seront ajoutées seront placées à la fin de celui-ci ;
- `O_CREAT` pour créer le fichier s'il n'existe pas.

Seules les principales valeurs sont décrites ci-dessus et un utilisateur désireux d'en savoir plus doit se référer à la 2e section du manuel en ligne (**`man 2 open`**). La valeur de `flags` peut aussi être une combinaison des valeurs ci-dessus (par la fonction « ou bit à bit » `|`). Par exemple, `O_RDWR | O_CREAT` qui permet d'ouvrir un fichier en écriture et en lecture en demandant sa création s'il n'existe pas.

Le paramètre `mode` n'est utilisé que si le drapeau `O_CREAT` est présent. Dans ce cas-là, `mode` indique les permissions [3] du fichier créé : les 3 derniers chiffres représentent les permissions pour (de gauche à droite) l'utilisateur, le groupe et les autres. 4 représente un fichier en mode lecture, 6 en lecture-écriture et 7 en lecture-écriture-exécution. Le mode 0644 indique donc que le fichier sera accessible en lecture-écriture pour le propriétaire et en lecture seule pour le groupe et les autres. Cette façon de désigner les permissions des fichiers est à rapprocher de l'utilisation de la commande **`chmod`** et des indications données par la commande **`ls -l`**. Notons aussi que les permissions attribuées par défaut à un fichier nouvellement créé peuvent être directement modifiées par un masque (`umask`), lui-même défini par l'appel système `umask()`.

L'exemple ci-dessous montre comment utiliser l'appel système `open()`. Précisons qu'il est toujours très important de tester les valeurs de retour des appels systèmes avant de continuer !

Listing 11.2 – *Ouverture de fichiers par `open()`*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
  int f;
```

2. Ces constantes sont définies dans un des fichiers d'en-tête du système (dans le fichier `/usr/include/asm/fcntl.h` pour les stations Unix) grâce à la commande `#define`.

3. Les permissions sont représentées par une valeur en octal (en base huit) codée sur 4 chiffres. Le premier chiffre permet, dans le cas d'un fichier exécutable (ou d'un répertoire), de placer l'identité de l'utilisateur qui exécute ce fichier à celle du fichier (4), ou encore de changer son groupe (2). Il sert aussi à signifier au système de garder le fichier exécutable en mémoire après son exécution. Cette dernière ressource n'est plus très employée sur les fichiers. Toutefois elle présente un grand intérêt sur les répertoires puisqu'elle permet par exemple d'autoriser quiconque à écrire dans le répertoire `/tmp` mais de ne pouvoir effacer que ses propres fichiers.

```
  f = open("/etc/passwd",O_RDONLY, NULL);
  if (f==-1) {
        printf("le fichier n'a pu etre ouvert\n");
        exit(EXIT_FAILURE);
  }
  printf("le fichier a ete correctement ouvert\n");
  /*
   * on utilise le fichier
   * .....
   * puis ferme le fichier
   * (cf exemples suivants)
   */
  exit(EXIT_SUCCESS);
}
```

L'appel système `read()`

Cet appel système permet de lire des données dans un fichier préalablement ouvert via l'appel système `open()`, c'est-à-dire dans un fichier représenté par un descripteur. L'appel système `read()` s'utilise de la manière suivante :

```
int read(int fd, void *buf, size_t nbytes)
```

Le paramètre `fd` indique le descripteur de fichier concerné, c'est-à-dire le fichier dans lequel les données vont être lues. Ce descripteur de fichier a été en général renvoyé par `open()` lorsque l'on a procédé à l'ouverture du fichier.

Le paramètre `buf` désigne une zone mémoire dans laquelle les données lues vont être stockées. Cette zone mémoire doit être au préalable allouée soit de façon statique (via un tableau par exemple), soit de façon dynamique (via un appel à la fonction `malloc()` par exemple). Le fait que `buf` soit un pointeur de type `void *` n'implique pas obligatoirement que les données soient sans type : cela signifie simplement que les données seront lues octet par octet, sans interprétation ni formatage. Cela permet également de relire des données (typiquement binaires) directement dans un conteneur (structure ou tableau) de même type sans avoir à faire de typage explicite (*cast*).

Le paramètre `nbytes` indique le nombre d'octets (et non le nombre de données) que l'appel `read()` va essayer de lire (il se peut qu'il y ait moins de données que le nombre spécifié par `nbytes`). Le type `size_t` est en fait équivalent à un type `int` et a été défini dans un fichier d'en-tête par un appel à `typedef`.

La valeur retournée par `read()` est le nombre d'octets effectivement lus. À la fin du fichier (c'est-à-dire quand il n'y a plus de données disponibles), 0 est retourné. En cas d'erreur, -1 est retourné.

Voici un exemple d'utilisation l'appel système `read()` :

Listing 11.3 – *Lecture avec `read()`*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>

#define NB_CAR 50

int main(int argc, char *argv[])
{
  int f;
  int n;
  char *texte;

  f = open("/etc/passwd",O_RDONLY, NULL);
  if(f==-1) {
        printf("le fichier n'a pu etre ouvert\n");
        exit(EXIT_FAILURE);
  }

  printf("le fichier a ete correctement ouvert\n");

  texte = (char *)malloc(NB_CAR*sizeof(char));
  if(texte == NULL) {
        printf("erreur d'allocation memoire\n");
        exit(EXIT_FAILURE);
  }

  n = read(f,texte, NB_CAR - 1);
  /* NB_CAR - 1 car il faut laisser de la place pour le caractere \0
         ajoute en fin de chaine de caracteres */

  if(n == -1) {
        printf("erreur de lecture du fichier\n");
        exit(EXIT_FAILURE);
  }
  else if(n == 0)
        printf("tout le fichier a ete lu\n");
  else
        printf("%d caracteres ont pu etre lus\n",n);

  exit(EXIT_SUCCESS);
}
```

L'appel système `write()`

```
int write(int fd, void *buf, size_t nbytes)
```

L'appel système `write()` permet d'écrire des données dans un fichier représenté par le descripteur de fichier `fd`. Il s'utilise de la même façon que l'appel système `read()`. Le descripteur de fichier `fd` a été en général renvoyé par `open()` lorsque l'on a procédé à l'ouverture du fichier.

La valeur retournée est le nombre d'octets effectivement écrits. En cas d'erreur, -1 est retourné.

L'appel système `close()`

```
int close(int fd)
```

Cet appel permet de fermer un fichier précédemment ouvert par l'appel système `open()`, c'est-à-dire de supprimer la référence à ce fichier dans la table maintenue par

le système (table des fichiers ouverts). Cela indique ainsi au système que le descripteur `fd` ne sera plus utilisé par le programme et qu'il peut libérer les ressources associées.

En cas d'oubli, le système détruira de toutes façons en fin de programme les ressources allouées pour ce fichier. Il est néanmoins préférable de fermer explicitement les fichiers qui ne sont plus utilisés.

Le programme ci-dessous présente un exemple complet de programme utilisant les appels système `open()`, `read()`, `write()` et `close()`. Ce programme est une version (très) simplifiée de la commande **`cp`** qui permet de copier des fichiers.

Listing 11.4 – *Copie simplifiée de fichiers*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define NB_CAR 200

int main(int argc, char *argv[])
{
  int f;
  int n, m;
  char *texte;

  /* verification du nombre d'arguments */
  if(argc < 3) {
    printf("pas assez d'arguments en ligne\n");
    printf("usage : %s <fichier1> <fichier2>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  /* ouverture en lecture du premier fichier */
  f = open(argv[1],O_RDONLY, NULL);
  if(f == -1) {
    printf("le fichier n'a pu etre ouvert\n");
    exit(EXIT_FAILURE);
  }

  printf("le fichier a ete correctement ouvert\n");

  /* allocation d'une zone memoire pour la lecture */
  texte = (char *)malloc(NB_CAR*sizeof(char));
  if(texte == NULL) {
    printf("erreur d'allocation memoire\n");
    exit(EXIT_FAILURE);
  }

  /* lecture des caracteres */
  n = read(f,texte, NB_CAR - 1);
  /* NB_CAR - 1 car il faut laisser de la place pour
     le caractere \0 ajoute en fin de chaine de
     caracteres */

  if(n == -1) {
    printf("erreur de lecture du fichier\n");
    exit(EXIT_FAILURE);
  }
```

```
  else if(n == 0)
    printf("tout le fichier a ete lu\n");
  else
    printf("%d caracteres ont pu etre lus\n",n);

  /* fermeture du premier fichier */
  close(f);

  /* ouverture du second fichier */
  f = open(argv[2], O_WRONLY | O_CREAT, 0644);
  if(f == -1) {
    printf("le fichier n'a pu etre ouvert\n");
    exit(EXIT_FAILURE);
  }

  printf("le fichier a ete correctement ouvert\n");

  /* ecriture des caracteres */
  m = write(f, texte, strlen(texte));

  if(m != n)
    {
      printf("Je n'ai pas ecrit le bon nombre de caracteres\n");
      perror("Erreur ");
    }
  printf("%d caracteres ecrits\n",m);

  /* fermeture du second fichier */
  close(f);

  exit(EXIT_SUCCESS);
}
```

Les descripteurs de fichier particuliers

Les descripteurs de fichiers 0, 1 et 2 sont spéciaux : ils représentent respectivement l'entrée standard, la sortie standard et la sortie d'erreur. Si le programme est lancé depuis un terminal de commande sans redirection, 0 est associé au clavier, 1 et 2 à l'écran. Ces descripteurs sont définis de manière plus lisible dans le fichier `<unistd.h>` :

Listing 11.5 – *Définition des descripteurs classiques de fichiers*

```
#define   STDIN_FILENO    0    /* standard input file descriptor */
#define   STDOUT_FILENO   1    /* standard output file descriptor */
#define   STDERR_FILENO   2    /* standard error file descriptor */
```

Nous verrons dans les exercice sur les tuyaux qu'il est possible de rediriger les entrées / sorties standard vers d'autres fichiers.

Le programme suivant affiche la chaîne « bonjour » à l'écran :

Listing 11.6 – *Afficher « bonjour » !*

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

```
int main(int argc,char *argv[])
{
    char *s = "bonjour\n";
    if(write(STDOUT_FILENO, s, strlen(s))==-1)
      exit(EXIT_FAILURE);
    else
      exit(EXIT_SUCCESS);
}
```

11.3 La bibliothèque standard d'entrées / sorties

Appel système ou fonction de bibliothèque ?

Les appels système présentés précédemment sont relativement lourds à utiliser et, s'ils ne sont pas utilisés avec précaution, ils peuvent conduire à des programmes très lents. Par exemple, un programme demandant 1000 fois de suite l'écriture d'un caractère à l'écran en utilisant l'appel système `write()` va provoquer 1000 interventions du noyau (donc 1000 passages en mode noyau et 1000 retours en mode utilisateur) et 1000 accès effectifs à l'écran...

Des fonctions ont donc été écrites en `C` afin de pallier les inconvénients des appels système. Ces fonctions, qui ont été standardisées et regroupées dans la bibliothèque standard d'entrées / sorties, utilisent bien entendu les appels système décrits ci-dessus pour accéder aux fichiers [4] et représentent, en quelque sorte, une surcouche plus fonctionnelle.

Les fonctions d'entrées / sorties de la bibliothèque standard ont les caractéristiques suivantes :

- Les accès aux fichiers sont asynchrones et « bufferisés ». Cela signifie que les lectures ou les écritures de données dans un fichier n'ont pas lieu au moment où elles sont demandées, mais qu'elles sont déclenchées ultérieurement. En fait, les fonctions de la bibliothèque standard utilisent des zones de mémoire tampon (des *buffers* en anglais) pour éviter de faire trop souvent appel au système et elles vident ces tampons soit lorsque cela leur est demandé explicitement, soit lorsqu'ils sont pleins.
- Les accès aux fichiers sont formatés, ce qui signifie que les données lues ou écrites sont interprétées conformément à un type de données (`int`, `char`, etc.).

Ces fonctions sont particulièrement indiquées pour traiter un flot de données de type texte.

Pour utiliser ces fonctions, il est indispensable d'inclure le fichier `stdio.h` dans le programme.

4. Il n'y a de toutes façons pas d'autres moyens !

Les descripteurs de fichier

Les fonctions d'entrées / sortie de la bibliothèque standard utilisent aussi des descripteurs de fichier pour manipuler les fichiers. Néanmoins, le descripteur de fichier employé ici n'est plus un entier faisant directement référence à la table des fichiers ouverts, mais un pointeur sur une structure de type `FILE` qui contient, d'une part, le descripteur entier utilisé par les appels système et, d'autre part, des informations spécifiques à l'utilisation de la bibliothèque (comme par exemple la désignation des zones de mémoire qui seront utilisées comme tampon).

En pratique, le détail de la structure `FILE` importe peu [5] et le programmeur n'a pas à modifier, ni même à connaître ces données. Son emploi est tout-à-fait similaire à celui des descripteurs de fichier entiers décrits précédemment.

En langage « informaticien », les fichiers sont souvent vus comme des flots de données (*stream* en anglais), c'est-à-dire comme une suite de données arrivant les unes derrière les autres, et les descripteurs de fichier de type `FILE *` sont alors souvent appelés descripteurs de flot. Cette pratique verbale ne doit pas dérouter le novice...

Les principales fonctions de la bibliothèque standard des entrées / sorties

La fonction `fopen()`

```
FILE *fopen(char *path, char *mode)
```

Elle ouvre le fichier indiqué par la chaîne de caractères `path` et retourne soit un pointeur sur une structure de type `FILE` décrivant le fichier, soit la valeur symbolique `NULL` si une erreur est survenue. Cette fonction utilise naturellement l'appel système `open()` de manière sous-jacente. La chaîne `mode` indique le type d'ouverture :

- `"r"` (pour *read*) ouvre le fichier en lecture seule ;
- `"r+"` comme `"r"` mais ouvre le fichier en lecture/écriture ;
- `"w"` (pour *write*) crée le fichier s'il n'existe pas ou le tronque à 0 et l'ouvre en écriture ;
- `"w+"` comme `"w"` mais ouvre le fichier en lecture/écriture ;
- `"a"` (pour *append*) crée le fichier s'il n'existe pas ou l'ouvre en écriture, en positionnant le descripteur à la fin du fichier ;
- `"a+"` comme `"a"` mais ouvre le fichier en lecture/écriture.

Listing 11.7 – *Ouverture d'un fichier*

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *f;
```

5. Les curieux peuvent regarder dans `/usr/include/stdio.h` la structure complète !

```
  f = fopen("/tmp/test","w");
  if(f == NULL) {
        printf("erreur d'ouverture du fichier\n");
        exit(EXIT_FAILURE);
  }

  printf("le fichier a ete correctement ouvert\n");
  exit(EXIT_SUCCESS);
}
```

La fonction `fprintf()`

```
fprintf(FILE *stream, const char *format, ...)
```

Elle écrit dans le fichier décrit par `stream` les données spécifiées par les paramètres suivants. La chaîne de caractères `format` peut contenir des données à écrire ainsi que la spécification d'un format d'écriture à appliquer. Cette fonction s'utilise de la même manière que la fonction `printf()`. La valeur retournée est le nombre de caractères écrits.

Exemple : `fprintf(f, "la valeur de i est : %d\n", i);`

La fonction `fscanf()`

```
int fscanf(FILE *stream, const char *format, ...)
```

Cette fonction lit le fichier décrit par `stream`, selon le format spécifié par la chaîne `format` et stocke les valeurs correspondantes aux adresses spécifiées par les paramètres suivants. Attention, les adresse spécifiées doivent être valides, c'est-à-dire qu'elles doivent correspondre à des zones mémoires préalablement allouées (soit de façon statique, soit de façon dynamique). Cette fonction fonctionne comme `scanf()` et il est impératif de ne pas oublier les & quand cela est nécessaire.

La valeur retournée est soit le nombre de conversions effectuées, soit la valeur symbolique `EOF` si aucune conversion n'a eu lieu.

Exemple : `fscanf(f, "%d", &i);`

La fonction `fgets()`

```
char *fgets(char *str, int size, FILE *stream)
```

Cette fonction lit des caractères depuis le fichier `stream` et les stocke dans la chaîne de caractères `str`. La lecture s'arrête soit après `size` caractères, soit lorsqu'un caractère de fin de ligne est rencontré, soit lorsque la fin du fichier est atteinte. La fonction `fgets()` retourne soit le pointeur sur `str`, soit NULL si la fin du fichier est atteinte ou si une erreur est survenue. Attention, dans les cas où cette fonction retourne NULL, il est fréquent que la chaîne de caractère `str` ne soit pas modifiée par rapport à sa valeur avant l'appel.

Les fonctions `feof()` et `ferror()` peuvent être utilisées pour savoir si la fin du fichier a été atteinte ou si une erreur est survenue. Elles seront présentées dans les paragraphes qui suivent.

Exemple :

```
char buf[255];
...
fgets(buf, 255, f);
...
```

La fonction `feof()`

```
int feof(FILE *stream)
```

Cette fonction retourne 0 si le fichier décrit par `stream` contient encore des données à lire, une valeur différente de 0 sinon.

La fonction `ferror()`

```
int ferror(FILE *stream)
```

Cette fonction retourne une valeur différente de 0 si une erreur est survenue lors de la dernière opération sur le fichier `stream`. Exemple :

```
...
if (ferror(f) != 0) {
  printf("erreur en ecriture\n");
  exit(EXIT_FAILURE);
}
```

La fonction `fclose()`

```
int fclose(FILE *stream)
```

Cette fonction indique au système que le fichier `stream` ne sera plus utilisé et que les ressources associées peuvent être libérées. La fonction retourne `EOF` en cas d'erreur ou 0 sinon.

Exemple : `fclose(f);`

La fonction `fdopen()`

```
FILE *fdopen(int fildes, const char *mode)
```

Il est parfois utile de pouvoir transformer un descripteur entier retourné par la fonction `open()` en un pointeur vers une structure de type `FILE`. Cela nous servira particulièrement lorsque nous manipulerons les tuyaux (cf 15).

Elle prend comme argument le descripteur de fichier entier `fildes` ainsi que le mode avec lequel le fichier avait été ouvert, soit `"w"` pour une ouverture en écriture et `"r"` lorsque le fichier a été ouvert en lecture. Cette fonction retourne alors un pointeur vers une structure `FILE` qui nous permettra d'utiliser les fonctions de la bibliothèque. L'exemple qui suit donne une illustration.

```
...
int fd;
FILE *fp=NULL;
...
if ((fd = open(myfile,O_WRONLY|O_CREAT,NULL)) == -1) {
  perror("Impossible d'ouvrir le fichier");
  exit(EXIT_FAILURE);
}
...
if ((fp = fdopen(fd,"w")) == NULL) {
  perror("Impossible d'associer un descripteur");
  exit(EXIT_FAILURE);
}
...
/* On utilise soit fclose()  */
/* soit close() mais pas les */
/* deux !! *
fclose(fp);
/* pas besoin de close(fd)*/
exit(EXIT_SUCCESS);
```

Les fonctions `sprintf()` et `sscanf()`

Les fonctions :
`int sprintf(char *str, const char *format, ...)`
`int sscanf(const char *str, const char *format, ...)`
fonctionnent de la même manière que les fonctions `fprintf()` et `fscanf()`, mais prennent ou stockent leurs données dans une chaîne de caractères (par exemple un tableau stocké en mémoire) et non dans un fichier. `sscanf()` est souvent utilisée en combinaison avec `fgets()`, car `fscanf()` ne permet pas de détecter simplement une fin de ligne.

Les descripteurs de fichier particuliers

Trois descipteurs de fichiers particuliers sont prédéfinis :
- `stdin` qui correspond au descripteur de fichier entier 0 (cf. appels système), c'est-à-dire à l'entrée standard ;
- `stdout` qui correspond au descripteur de fichier entier 1, c'est-à-dire à la sortie standard ;
- `stderr` qui correspond au descripteur de fichier entier 2, c'est-à-dire à la sortie d'erreur standard.

Les fonctions `scanf()` et `printf()` travaillent respectivement sur `stdin` et `stdout`. Ainsi,
`fprintf(stdout, "%d\n", i)` est équivalent à `printf("%d\n", i)`.

Voici un exemple de programmation de la commande (simplifiée) **`cat`** : elle affiche à l'écran le contenu du fichier dont le nom est passé en paramètre.

Listing 11.8 – *Ouverture et lecture d'un fichier avec affichage*

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  FILE *f;
  char line[255];

  if (argc != 2) {
    fprintf(stderr,"il me faut uniquement le nom du ");
    fprintf(stderr,"fichier en argument\n");
    exit(EXIT_FAILURE);
  }
  f = fopen(argv[1], "r");
  if (f == NULL) {
    fprintf(stderr,"impossible d'ouvrir le fichier\n");
    exit(EXIT_FAILURE);
  }
  while (fgets(line, 255, f) != 0) {
    printf("%s", line); /* le caractere '\n' est deja dans line */
  }
  if (ferror(f) != 0) {
    fprintf(stderr, "erreur de lecture sur le fichier\n");
    exit(EXIT_FAILURE);
  }
  fclose(f);
  exit(EXIT_SUCCESS);
}
```

11.4 Exercices

Question 1

Modifier l'application **`addition`** (voir le TD Les processus sous Unix) pour qu'elle prenne les nombres non plus sur la ligne de commande, mais dans un fichier dont le nom est précisé sur la ligne de commande. Le fichier contiendra un nombre par ligne.

Question 2

Modifier l'application précédente pour qu'elle prenne l'entrée standard si aucun nom de fichier n'a été donné sur la ligne de commande.

11.5 Corrigés

Listing 11.9 – *Correction du premier exercice*

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int s,i;
  char buf[255];
  FILE *f;

  if (argc != 2) { /* un argument et un seul */
    fprintf(stderr,"arguments incorrects\n");
    exit(EXIT_FAILURE);
  }
  f = fopen(argv[1], "r");
  if (f == NULL) {
    fprintf(stderr, "impossible d'ouvrir le fichier en lecture\n");
    exit(EXIT_FAILURE);
  }
  s = 0;
  /* fgets fait une lecture ligne par ligne, et stocke */
  /* la ligne lue (y compris le \n) dans buf */
  /* il faut que la ligne fasse moins de 255 caracteres */
  /* sinon seuls les 255 premiers caracteres sont lus */
  while (fgets(buf, 255, f) != NULL) {
    /* fgets() retourne le pointeur NULL lorsqu'il */
    /* ne peut plus lire. Ici on considere qu'il */
    /* s'agit dans ce cas de la fin du fichier */
    if (sscanf(buf,"%d", &i) != 1) {
      fprintf(stderr, "erreur de conversion sur %s\n", buf);
      fclose(f);
      exit(EXIT_FAILURE);
    }
    /* une solution plus simple (mais sans verification) */
    /* serait de faire : */
    /* i = atoi(buf); */
    s += i;
  }
  /* Un habitue du C aurait pu ecrire : */
  /* while (fgets(buf,255,f)) s += atoi(buf); */
  /* mais c'est moins lisible */
  if (ferror(f)) {
    fprintf(stderr, "erreur de lecture\n");
    fclose(f);
    exit(EXIT_FAILURE);
  }
  printf("la somme est: %d\n", s);
  fclose(f);
  exit(EXIT_SUCCESS);
}
```

Listing 11.10 – *Correction du deuxième exercice*

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int s,i;
  char buf[255];
```

```
  FILE *f;

  if (argc != 2 && argc != 1) { /* zero ou un argument */
    fprintf(stderr,"arguments incorrects\n");
    exit(EXIT_FAILURE);
  }
  if (argc == 2) { /* un fichier est precise en parametre */
    f = fopen(argv[1], "r");
    if (f == NULL) {
      fprintf(stderr, "impossible d'ouvrir le fichier en lecture\n");
      exit(EXIT_FAILURE);
    }
  } else { /* pas de fichier en parametre, */
    /* on utilise l'entree standard */
    f = stdin;
    /* Dans le cas de stdin, la fin du fichier s'obtient */
    /* en tapant ^d (Ctrl-d) qui est le marqueur de fin */
  }

  s = 0;
  /* meme code que l'exercice precedent */
  while (fgets(buf, 255, f) != NULL) {
    if (sscanf(buf,"%d", &i) != 1) {
      fprintf(stderr, "erreur de conversion sur %s\n", buf);
      fclose(f);
      exit(EXIT_FAILURE);
    }
    s += i;
  }
  if (ferror(f)) {
    fprintf(stderr, "erreur de lecture\n");
    fclose(f);
    exit(EXIT_FAILURE);
  }
  printf("la somme est: %d\n", s);
  fclose(f);
  exit(EXIT_SUCCESS);
}
```

11.6 Corrections détaillées

Lire de façon basique

Si nous voulons lire dans un fichier les nombres que nous rentrions en ligne de commandes, il nous faut manipuler les appels systèmes `open()`, `read()` et `close()`. Ces fonctions vont nous permettre d'associer un descripteur de fichier (un simple entier) et un fichier présent sur le disque. Ceci se produit lorsque nous faisons appel à la fonction `open()`. Le système associe de manière unique dans notre programme en cours d'exécution un numéro et un fichier. Ainsi lorsque nous voulons travailler sur le même fichier au long du programme, nous n'avons plus à nous demander si le chemin est le bon, si le fichier est sur le disque ou si le système l'a placé dans une zone tampon en mémoire.

Il est essentiel de remarquer que les appels système `read()` et `write()` sont très basiques. Ils ne savent que lire ou écrire des octets, il n'est donc pas question d'oublier ce que l'on a écrit dans un fichier (un entier, deux flottants puis trois entiers) sous peine de relire n'importe quoi. Reste aussi à savoir comment organiser le fichier dont nous allons nous servir pour le programme d'addition. Va-t-on y mettre des valeurs écrites sous la même forme que leur représentation dans la mémoire (écriture sous forme d'octets) ou choisira-t-on une représentation sous forme de mot (écriture ASCII). Il y a en effet une différence entre l'entier 234 qui sera écrit ae00 sur quatre octets et le mot « 234 » qui s'écrit à l'aide de trois symboles. Remarquons alors que la première méthode nous impose d'écrire un programme capable d'écrire les différents nombres sous forme d'octets dans un fichier ! Ainsi pour apprendre à lire, nous devons déjà savoir écrire. Nous allons donc commencer par la deuxième méthode, lire des mots et les traduire en nombres. Nous supposerons que sur chaque ligne du fichier nous trouvons un nombre, *i.e.* chaque nombre est séparé du suivant un caractère de saut à la ligne. Nous allons donc lire le fichier caractère par caractère et mettre le caractère lu dans un tableau (donc on avancera l'index) jusqu'à la lecture d'un retour chariot [6]. Nous remplacerons alors le retour chariot par le caractère de terminaison de chaîne afin de faire croire à la fonction de conversion `atoi()` que la chaîne s'arrête là. Le programme est le suivant :

Listing 11.11 – *Lecture de nombres dans un fichier*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int descr_file;
  int res;
```

6. Attention c'est charrette qui prend 2 *r*

```
  char texte[256];
  int index;
  int somme;

  /* Un peu de prudence ne nuit pas ! */
  if (argc !=2) {
    printf("J'ai besoin d'un nom de fichier\n");
    exit(EXIT_FAILURE);
  }

  descr_file = open(argv[1],O_RDONLY,NULL);
  /* Un peu plus de prudence fait du bien */
  if (descr_file == -1) {
    perror("Impossible d'ouvrir le fichier");
    exit(EXIT_FAILURE);
  }

  somme = 0; index = 0;
  while ((res = read(descr_file,texte+index,1)) >0) {
    if (texte[index] == '\n') {
      /* on est sur une fin de ligne, on peut lire  */
      /* un entier. On remplace le caractere de fin */
      /* de ligne par '\0' pour signaler a atoi()   */
      /* que le mot se termine.                     */
      texte[index] = '\0';
      somme += atoi(texte);
      index = 0;
    } else index++;
  }
  close(descr_file);
  printf("%d\n",somme);
  exit(EXIT_SUCCESS);
}
```

Nous aurions pu pour des questions de rapidité lire l'intégralité du fichier. Mais dans ce cas il faut connaître sa taille, car tous les caractères lus devront trouver une place. On peut à cette fin utiliser la fonction `stat()` et ensuite procéder à une lecture du tableau de caractères chargés en mémoire. La fonction `stat()` est déclarée comme suit :

```
int stat(const char *restrict path, struct stat *restrict buf);
```

Elle permet de recueillir dans une structure (`struct stat`) des informations sur un fichier dont :

- le propriétaire : `st_uid` ;
- la taille en octet : `st_size`.

Le programme ressemblera alors à ceci :

Listing 11.12 – *Lecture en bloc*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
```

```
  int descr_file;
  int res;
  char *texte=NULL;
  int cur,endc;
  int somme;

  struct stat buf;

  /* Un peu de prudence ne nuit pas ! */
  if (argc !=2) {
    printf("J'ai besoin d'un nom de fichier\n");
    exit(EXIT_FAILURE);
  }

  descr_file = open(argv[1],O_RDONLY,NULL);
  /* Un peu plus de prudence fait du bien */
  if (descr_file == -1) {
    perror("Impossible d'ouvrir le fichier");
    exit(EXIT_FAILURE);
  }

  /* On demande la taille, contenue */
  /* dans la structure buf          */
  fstat(descr_file,&buf);

  /* On alloue la place necessaire  */
  /* pour lire le texte             */
  texte = malloc((size_t)buf.st_size + 1);

  /* On lit le texte, en verifiant  */
  /* que tout s'est bien passe      */
  if ((res = read(descr_file,texte,(int)buf.st_size)) !=
      (int)buf.st_size) {
    perror("impossible de lire tout le fichier");
    exit(EXIT_FAILURE);
  }
  /* On peut fermer le fichier */
  close(descr_file);

  /* On place un caractere de fin dans le tableau */
  texte[(int)buf.st_size] = '\0';
  somme = 0; cur = 0;endc = 0;
  while (texte[cur] != '\0') {
    if (texte[cur] == '\n') {
 /* on est sur une fin de ligne, on peut lire un entier */
 /* on remplace le caractere de fin de ligne par '\0'   */
 /* pour signaler a atoi() que le mot se termine.       */
      texte[cur] = '\0';
      somme += atoi(texte+endc);
      endc = ++cur;
    } else cur++;
  }
  free(texte);
  printf("%d\n",somme);
  exit(EXIT_SUCCESS);
}
```

À la places des appels système nous pouvons aussi utiliser les fonctions de la bibliothèque `fopen()`, `fscanf()` et `fclose`. Si la première et la dernière sont sensiblement identiques aux appels système d'un point de vue rédactionnel, la fonction `fscanf()` possède un degré de structuration bien plus grand que la fonction `read()`. En effet la

fonction `fscanf()` accepte un argument de type format qui décrit ce que l'on cherche à lire et un nombre variable d'arguments afin de placer chaque lecture demandée à un endroit précis.

Le programme précédent devient tout de suite beaucoup plus simple :

Listing 11.13 – *Un grand pas vers la simplicité*

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  FILE *fp=NULL;
  int somme, nombre;

    /* Un peu de prudence ne nuit pas ! */
  if (argc !=2) {
    printf("J'ai besoin d'un nom de fichier\n");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen(argv[1],"r")) == NULL) {
    perror("Impossible d'ouvrir le fichier");
    exit(EXIT_FAILURE);
  }

  somme = 0;
  while(fscanf(fp,"%d",&nombre) == 1) {
    somme += nombre;
  }
  fclose(fp);
  printf("%d\n",somme);
  exit(EXIT_SUCCESS);
}
```

Choisir où lire

Afin d'autoriser la saisie des nombres après le lancement du programme, nous allons choisir où lire ceux-ci, *i.e.* sur l'entrée standard ou dans un fichier. Ceci se fait très simplement en regardant combien nous avons d'arguments sur la ligne de commande et en plaçant le descripteur de fichier sur l'entrée standard si aucun fichier ne peut être ouvert.

Listing 11.14 – *Choix de l'endroit de lecture*

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  FILE *fp=NULL;
  int somme, nombre;

    /* Un peu de prudence ne nuit pas ! */
  if (argc !=2) {
    fp = stdin;
  } else {
```

```
    if ((fp = fopen(argv[1],"r")) == NULL) {
      perror("Impossible d'ouvrir le fichier");
      exit(EXIT_FAILURE);
    }
  }

  somme = 0;
  while(fscanf(fp,"%d",&nombre) == 1) {
    somme += nombre;
  }
  if (fp != stdin)  fclose(fp);
  printf("%d\n",somme);
  exit(EXIT_SUCCESS);
}
```

L'exécution donne ceci :

```
menthe22>addargbis
12 [return]
14 [return]
16 [^D]
42
menthe22>
```

Vous remarquez que l'on signifie au programme que l'entrée standard ne lui donnera plus rien en tapant le caractère de fin de fichier `^D`.

12

Création de processus sous Unix

Dans le chapitre précédent, la notion de processus a été abordée, de même que les appels système permettant d'identifier des processus. Ce chapitre étudie la création des processus, sans aller jusqu'au bout de la démarche : celle-ci sera complétée dans le chapitre suivant.

12.1 La création de processus

La bibliothèque `C` propose plusieurs fonctions pour créer des processus avec des interfaces plus ou moins perfectionnées. Cependant toute ces fonctions utilisent l'appel système `fork()` qui est la seule et unique façon de demander au système d'exploitation de créer un nouveau processus.

– `pid_t fork(void)`
  Cet appel système crée un nouveau processus. La valeur retournée est le *pid* du fils pour le processus père, ou 0 pour le processus fils. La valeur -1 est retournée en cas d'erreur.

`fork()` se contente de dupliquer un processus en mémoire, c'est-à-dire que la zone mémoire du processus père est recopiée dans la zone mémoire du processus fils. On obtient alors deux processus identiques, déroulant le même code en concurrence. Ces deux processus ont les mêmes allocations mémoire et les mêmes descripteurs de fichiers. Seuls le *pid*, le *pid* du père et la valeur retournée par `fork()` sont différents. La valeur retournée par `fork()` est en général utilisée pour déterminer si on est le processus père ou le processus fils et permet d'agir en conséquence.

L'exemple suivant montre l'effet de `fork()` :

Listing 12.1 – *L'effet `fork()`*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int i;

  printf("[processus %d] je suis avant le fork\n", getpid());
```

```
  i = fork();
  printf("[processus %d] je suis apres le fork, il a retourne %d\n",
         getpid(), i);

  exit(EXIT_SUCCESS);
}
```

Le résultat de l'exécution est :

```
menthe22>./ex22
[processus 1197] je suis avant le fork
[processus 1197] je suis apres le fork, il a retourne 1198
[processus 1198] je suis apres le fork, il a retourne 0
```

Le premier `printf()` est avant l'appel à `fork()`. À ce moment-là, il n'y a qu'un seul processus, donc un seul message « **`je suis avant le fork`** ». En revanche, le second `printf()` se trouve après le `fork()` et il est donc exécuté deux fois par deux processus différents. Notons que la variable `i`, qui contient la valeur de retour du `fork()` contient deux valeurs différentes.

Lors de l'exécution, l'ordre dans lequel le fils et le père affichent leurs informations n'est pas toujours le même. Cela est dû à l'ordonnancement des processus par le système d'exploitation.

12.2 L'appel système `wait()`

Il est souvent très pratique de pouvoir attendre la fin de l'exécution des processus fils avant de continuer l'exécution du processus père (afin d'éviter que celui-ci se termine avant ses fils, par exemple). La fonction : `pid_t wait(int *status)` permet de suspendre l'exécution du père jusqu'à ce que l'exécution d'un des fils soit terminée. La valeur retournée est le *pid* du processus qui vient de se terminer ou -1 en cas d'erreur. Si le pointeur `status` est différent de NULL, les données retournées contiennent des informations sur la manière dont ce processus s'est terminé, comme par exemple la valeur passée à `exit()` (voir le manuel en ligne pour plus d'informations, **`man 2 wait`**).

Dans l'exemple suivant, nous complétons le programme décrit précédemment en utilisant `wait()` : le père attend alors que le fils soit terminé pour afficher les informations sur le `fork()` et ainsi s'affranchir de l'ordonnancement aléatoire des tâches.

Listing 12.2 – *Attente passive de la terminaison du fils*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
  int i,j;
```

```
  printf("[processus %d] je suis avant le fork\n", getpid());
  i = fork();
  if (i != 0) { /* i != 0 seulement pour le pere */
    j = wait(NULL);
    printf("[processus %d] wait a retourne %d\n", getpid(), j);
  }
  printf("[processus %d] je suis apres le fork, il a retourne %d\n",
         getpid(), i);

  exit(EXIT_SUCCESS);
}
```

L'exécution donne :

```
[processus 1203] je suis avant le fork
[processus 1204] je suis apres le fork, il a retourne 0
[processus 1203] wait a retourne 1204
[processus 1203] je suis apres le fork, il a retourne 1204
```

Notons que cette fois-ci, l'ordre dans lequel les informations s'affichent est toujours le même.

12.3 Exercices

Question 1

Écrire un programme créant un processus fils. Le processus père affichera son identité ainsi que celle de son fils, le processus fils affichera son identité ainsi que celle de son père.

Question 2

Nous allons essayer de reproduire le phénomène d' « adoption » des processus vu précédemment. Pour cela, il faut ajouter un appel à la fonction `sleep()`, de façon à ce que le père ait terminé son exécution avant que le fils n'ait appelé `getppid()`.

Question 3

Ajouter un appel à la fonction `wait(NULL)` pour éviter que le père ne se termine avant le fils.

Question 4

Dans l'exercice précédent, utiliser `wait()` pour obtenir le code de retour du fils et l'afficher.

Question 5

Écrire un programme créant 3 fils, faire en sorte que ceux-ci se terminent dans un autre ordre que l'ordre dans lequel ils ont été créés, puis demander un père d'attendre la fin des 3 fils et d'indiquer l'ordre dans lequel ils se terminent.

12.4 Corrigés

Listing 12.3 – *Corrigé du premier exercice*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int s;

  /* avant le fork(), un seul processus */
  s = fork();
  /* apres le fork(), deux processus */
  /* dans le pere, continuation du processus de depart, s>0 */
  /* dans le fils, s=0 */
  if (s < 0) { /* erreur */
    fprintf(stderr,"erreur dans fork\n");
    exit(EXIT_FAILURE);
  }
  if (s == 0) { /* on est le fils */
    printf("Je suis le fils, mon pid est %d,",getpid());
    printf(" celui de mon pere est %d\n",getppid());
  } else { /* on est le pere */
    printf("Je suis le pere, mon pid est %d,"getpid());
    printf(" celui de mon fils est %d\n",s);
  }
  /* cette ligne est executee par le fils et le pere */
  exit(EXIT_SUCCESS);
}
```

Listing 12.4 – *Corrigé du deuxième exercice*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int s;

  s = fork();
  if (s < 0) { /* erreur */
    fprintf(stderr,"erreur dans fork\n");
    exit(EXIT_FAILURE);
  }
  if (s == 0) { /* on est le fils */
    sleep(10); /* pendant ce temps, le pere se termine */
    printf("Je suis le fils, mon pid est %d,",getpid());
    printf(" celui de mon pere est %d\n",getppid());
  } else { /* on est le pere */
    printf("Je suis le pere, mon pid est %d,",getpid());
    printf(" celui de mon fils est %d\n",s);
    /* Le pere se termine rapidement */
  }
  /* cette ligne est executee par le fils et le pere */
  exit(EXIT_SUCCESS);
}
```

Listing 12.5 – *Corrigé du troisième exercice*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int s;

  s = fork();
  if (s < 0) { /* erreur */
    fprintf(stderr,"erreur dans fork\n");
    exit(EXIT_FAILURE);
  }
  if (s == 0) { /* on est le fils */
    sleep(10);
    printf(" Je suis le fils, mon pid est %d,",getpid());
    printf("celui de mon pere est %d\n",getppid());
  } else { /* on est le pere */
    printf("Je suis le pere, mon pid est %d,",getpid());
    printf(" celui de mon fils est %d\n",s);
    /* Le pere arrive ici pendant que le fils fait le sleep() */
    if (wait(NULL) < 0) { /* bloque en attendant la fin du fils */
      fprintf(stderr, "erreur dans wait\n");
      exit(EXIT_FAILURE);
    }
  }
  /* cette ligne est executee par le fils et le pere */
  exit(EXIT_SUCCESS);
}
```

Listing 12.6 – *Corrigé du quatrième exercice*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int s;
  int status;

  s = fork();
  if (s < 0) { /* erreur */
    fprintf(stderr,"erreur dans fork\n");
    exit(EXIT_FAILURE);
  }
  if (s == 0) { /* on est le fils */
    sleep(10);
    printf("Je suis le fils, mon pid est %d,",getpid());
    printf(" celui de mon pere est %d\n",getppid());
    /* le fils retourne les deux derniers chiffres de son pid */
    /* en guise de valeur de retour */
    /* (ce n'est qu'un exemple) */
    exit(getpid() % 100);
  } else { /* on est le pere */
    printf("Je suis le pere, mon pid est %d,",getpid());
    printf(" celui de mon fils est %d\n",s);
    /* Le pere arrive ici pendant que le fils fait le sleep() */
    if (wait(&status) < 0) { /* bloque en attendant la fin du fils */
      fprintf(stderr, "erreur dans wait\n");
      exit(EXIT_FAILURE);
```

```
    }
    printf("Mon fils a retourne la valeur %d\n",WEXITSTATUS(status));
  }

  exit(EXIT_SUCCESS); /* cette ligne est executee le pere seulement */
}
```

12.5 Corrections détaillées

Une des choses essentielles à comprendre est le fait que la fonction `fork()` réalise véritablement une opération de clonage. Elle permet donc de faire une copie exacte du processus qui l'invoque et si le cours est bien présent à votre esprit, il s'agit de reproduire à la fois le code, la pile, le tas mais aussi et c'est important le marqueur indiquant où l'on se trouve dans le processus.

La deuxième remarque importante concerne la filiation. Il s'agit ici de raisonner de manière logique et naturelle. Si l'on détruit le père d'un processus, ce dernier va naturellement devenir un des nombreux fils du processus `init`. C'est ce que propose de faire le deuxième exercice.

Le numéro de processus

Le programme affichant son `pid` est très simple. Comme nous avons pris soin, ainsi qu'on nous l'a appris à l'école primaire, de lire un énoncé jusqu'à la fin, nous allons directement écrire le programme dans sa version fichier. Nous préférons cela à une redirection de la sortie standard afin de mettre en pratique certaines choses vues concernant les fichiers. Mais afin de ne pas se montrer extrémiste, nous allons quand même laisser à l'utilisateur le choix de la sortie standard ! Nous aboutissons au code suivant :

Listing 12.7 – *Affichage du numéro de processus*

```
/* getpid() */
#include <sys/types.h>
#include <unistd.h>
/* exit()  */
#include <stdlib.h>
/* fopen() fclose() fprintf() ... */
#include <stdio.h>

int main(int argc, char **argv)
{
  FILE *logfile;

  if (argc<2) {
    logfile = stdout;
  } else {
    if ((logfile = fopen(argv[1],"a+")) == NULL) {
      fprintf(stderr,"Impossible d'ouvrir %s\n",v[1]);
      exit(EXIT_FAILURE);
    }
  }
  fprintf(logfile,"---\n[processus %5d]\n",getpid());
  fprintf(logfile,"\tprocessus pere: %5d\n",getppid());
  fclose(logfile);
  exit(EXIT_SUCCESS);
}
```

Si le programme est exécuté sans argument autre que son nom, les résultats s'affichent directement à l'écran, sinon ils seront placés dans le fichier dont le nom est donné comme deuxième argument :

```
menthe22>./showpid
[processus  3270]
        processus pere: 17603
```

Afin de trouver le processus père de ce programme dont nous ne connaissons que le numéro d'identité, nous allons utiliser la commande **`ps -ux`** et la commande **`grep`** afin de réduire l'affichage des résultats. Voici ce que nous obtenons :

```
menthe22>ps ux | grep 17603
yak  17603  pts4  S  Oct02   0:00 -tcsh
yak   3278  pts4  S  16:17   0:00 grep 17603
menthe22>
```

Nous observons que le père de notre processus n'est rien d'autre que l'interprète de commandes, ici `tcsh`.

Nous allons maintenant nous placer dans un contexte assez meurtrier et nous allons voir comment se passe la filiation si le père meurt avant le fils. Pour cela il nous faut disposer d'un court instant pour tuer le père (l'interprète de commandes) de notre processus.

Paricide

La fonction `sleep()` permet de suspendre l'exécution d'un processus pendant un certain temps. Nous allons donc reprendre notre programme, faire en sorte qu'il affiche les mêmes données que précédemment, le faire dormir, puis en profiter pour tuer son père et lui demander d'afficher de nouveau les informations concernant son identité et celui de son père. Le code qui en découle est très simple :

Listing 12.8 – *Affichage après la mort du père*

```
/* getpid() sleep() */
#include <sys/types.h>
#include <unistd.h>
/* exit()  */
#include <stdlib.h>
/* fopen() fclose() fprintf() ... */
#include <stdio.h>

int main(int argc, char **argv)
{
  FILE *logfile;

  if (argc<2) {
    logfile = stdout;
  } else {
    if ((logfile = fopen(argv[1],"a+")) == NULL) {
      fprintf(stderr,"Impossible d'ouvrir %s\n",v[1]);
      exit(EXIT_FAILURE);
    }
  }
  fprintf(logfile,"---\n[processus %5d]\n",getpid());
  fprintf(logfile,"\tprocessus pere: %5d\n",getppid());
  sleep(10);
  fprintf(logfile,"Je me reveille, Papa ou es-tu?\n");
```

```
  fprintf(logfile,"\tprocessus pere: %5d\n",getppid());
  fclose(logfile);
  exit(EXIT_SUCCESS);
}
```

Pour lancer le programme nous allons déjà ouvrir une nouvelle fenêtre afin de disposer d'un interprète de commandes tout neuf prêt à être tué ! Nous lançons notre commande en tâche de fond, et oui il faut que l'interprète puisse être tué et donc ne pas être dans une situation d'attente de la fin de notre programme ! L'interprète de commandes est tué par la commande `exit`.

```
menthe22>showpidbis > /tmp/mylog &
menthe22>exit
... argh!!!
```

Nous allons voir le résultat dans le fichier des événements :

```
menthe22>more /tmp/mylog
[processus  3270]
        processus pere: 17603
Je me reveille, Papa ou es-tu?
        processus pere:     1
menthe22>
```

Avant de tuer l'interprète de commandes, l'affichage ne change pas vraiment si ce n'est que les numéros d'identité ont augmenté, d'autres processus ont été lancés entre temps tout cela est bien normal [1]. Puis le processus s'endort et nous tuons son père. Nous remarquons qu'à son réveil, notre processus est maintenant devenu un descendant direct du processus père de tous les autres `init` (ou presque, selon les systèmes voir le cours). Ce choix paraît très logique dans la mesure où la création de processus (hormis `init`) se réalise par clonage et filiation. De plus affilier ce processus orphelin à `init` lui assure d'avoir un père tant que l'ordinateur est en marche ! Nous allons maintenant examiner la création de processus.

Création de processus

Créer un processus peut être réalisé par l'utilisation de la commande système `fork()`. Afin que ce qui suit soit clair nous allons nous remettre en mémoire de façon schématique comment sont gérés les processus sous Unix (figure 12.1).

La commande `fork()` permet donc de « recopier » l'intégralité de ce qui représente le processus qui y fait appel afin de créer un nouveau processus en tous poins identique. Une fois l'appel réalisé, nous avons donc bel et bien deux processus qui vont accomplir la même chose (bon nous verrons comment les différencier afin de les particulariser après). La seule chose qui va nous permettre de les différencier dans un premier temps sera leur numéro de processus. Nous pouvons d'ailleurs citer la page de manuel :

1. Si le correcteur avait travaillé sans cesse sur la correction les numéros auraient probablement été moins éloignés les uns des autres mais bon...

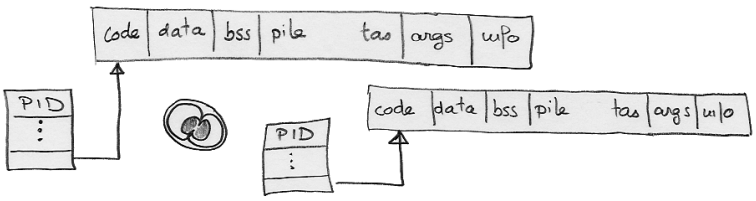


FIGURE 12.1 – *Les deux processus représentés sont des clones l'un de l'autre. S'il existe bien un lien de filiation entre eux, ils ne partagent aucune zone mémoire et chacun vit sa vie dans la plus grande indifférence de ce que peut faire l'autre. Il est fondamental de bien comprendre que chaque processus possède son propre espace mémoire, son propre tas, sa propre pile, etc.*

> `fork` crée un processus fils qui diffère du processus parent uniquement par ses valeurs `PID` et `PPID` et par le fait que toutes les statistiques d'utilisation des ressources sont remises à zéro. Les verrouillages de fichiers, et les signaux en attente ne sont pas hérités.

Le programme utilisant `fork()` est très simple à faire à partir de ce que nous avions déjà fait. Nous allons simplement compliquer un peu la résolution de l'exercice pour montrer que le père et le fils sont deux processus s'exécutant dans deux contextes différents.

Listing 12.9 – *Deux contextes bien différents*

```
/* man getpid(), fork() */
#include <sys/types.h>
#include <unistd.h>
/* man exit(), malloc() */
#include <stdlib.h>
/* man printf */
#include <stdio.h>

/* Cette variable est globale */
/* a tout le programme. C'est */
/* sale mais c'est pour la    */
/* bonne cause !!*/
int tab_stat[2];


int main(int argc, char **argv)
{
  int *tab_dyn;
  int i,le_numero_au_debut;

  le_numero_au_debut = getpid();
  printf("[processus %5d] mon pere est : %d.\n",
         getpid(),getppid());
  printf("[processus %5d] j'alloue\n",getpid());
  tab_dyn = malloc(2*sizeof(int));
```

```
  printf("[processus %5d] je remplis\n",getpid());

  tab_stat[0] = 10;
  tab_stat[1] = 11;
  tab_dyn[0] = 20;
  tab_dyn[1] = 21;

  printf("[processus %5d] je me clone\n",getpid());

  /* A partir de maintenant le code doit etre */
  /* execute deux fois */
  if (fork()==-1) {
    perror("Impossible de me cloner");
    exit(EXIT_FAILURE);
  }

  printf("[processus %5d] Mon pere est : %d\n",getpid(),
         getppid());
  printf("[processus %5d] Le numero enregistre : %d\n",
         getpid(), le_numero_au_debut);
  for(i=0;i<2;i++){
    printf("[processus %5d] tab_stat : %d\n",
           getpid(),tab_stat[i]);
    printf("[processus %5d] tab_dyn : %d\n",
           getpid(),tab_dyn[i]);
  }
  exit(EXIT_SUCCESS);
}
```

Lorsqu'on regarde l'affichage résultat de l'exécution on aboutit à ce qui suit. Attention nous avons volontairement accentué la séparation entre les lignes de code exécutées deux fois et les lignes de code exécutées une seule fois.

```
menthe22>clone1
[processus 18797] mon pere est : 18711.
[processus 18797] j'alloue
[processus 18797] je remplis
[processus 18797] je me clone

[processus 18798] Mon pere est : 18797
[processus 18798] Le numero enregistre : 18797
[processus 18798] tab_stat : 10
[processus 18798] tab_dyn : 20
[processus 18798] tab_stat : 11
[processus 18798] tab_dyn : 21

[processus 18797] Mon pere est : 18711
[processus 18797] Le numero enregistre : 18797
[processus 18797] tab_stat : 10
[processus 18797] tab_dyn : 20
[processus 18797] tab_stat : 11
[processus 18797] tab_dyn : 21
menthe22>
```

Nous remarquons bien que tant que `fork()` n'a pas été appelé, un seul processus existe, celui dont le `pid` est `18797`. Ensuite, deux processus s'exécutent de manière indépendante et nous voyons alors apparaître deux fois chaque sortie sur l'écran, mais avec quelques différences et notamment le numéro de processus qui n'est pas le même.

Jusque là rien de très exceptionnel. Maintenant nous allons compliquer un peu les choses et écrire dans un fichier au lieu d'écrire sur la sortie standard, et comme nous sommes vraiment curieux, nous allons procéder à l'ouverture et la fermeture de ce fichier **après** l'appel à `fork()`. Le programme est le suivant, remarquez bien l'ouverture de fichier qui utilise le drapeau `"w+"`.

Listing 12.10 – *Ecriture dans un fichier « partagé »*

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

/* Cette variable est globale */
/* a tout le programme. C'est */
/* sale mais c'est pour la    */
/* bonne cause !!*/
int tab_stat[2];


int main(int argc, char **argv)
{
  FILE *fp=NULL;
  int *tab_dyn;
  int i,le_numero_au_debut;

  le_numero_au_debut = getpid();
  printf("[processus %5d] mon pere est : %d.\n",
         getpid(),getppid());
  printf("[processus %5d] j'alloue\n",getpid());
  tab_dyn = malloc(2*sizeof(int));
  printf("[processus %5d] je remplis\n",getpid());

  tab_stat[0] = 10;
  tab_stat[1] = 11;
  tab_dyn[0] = 20;
  tab_dyn[1] = 21;

  printf("[processus %5d] je me clone\n",getpid());

  /* A partir de maintenant le code doit etre */
  /* execute deux fois */
  if (fork()==-1) {
    perror("Impossible de me cloner");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen("curieux.txt","a+")) == NULL) {
    perror("Impossible d'ouvrir curieux.txt");
    exit(EXIT_FAILURE);
  }
  fprintf(fp,"[processus %5d] Mon pere est : %d\n",
          getpid(),getppid());
  fprintf(fp,"[processus %5d] Le numero enregistre : %d\n",
         getpid(), le_numero_au_debut);
  for(i=0;i<2;i++){
    fprintf(fp,"[processus %5d] tab_stat : %d\n",
            getpid(),tab_stat[i]);
    fprintf(fp,"[processus %5d] tab_dyn : %d\n",
            getpid(),tab_dyn[i]);
  }
  fclose(fp);
```

```
  exit(EXIT_SUCCESS);
}
```

La sortie donne ceci :

```
menthe22>clone2
[processus 18859] mon pere est : 18711.
[processus 18859] j'alloue
[processus 18859] je remplis
[processus 18859] je me clone
menthe22>more curieux.txt
[processus 18859] Mon pere est : 18711
[processus 18859] Le numero enregistre : 18859
[processus 18859] tab_stat : 10
[processus 18859] tab_dyn : 20
[processus 18859] tab_stat : 11
[processus 18859] tab_dyn : 21
menthe22>
```

On remarque que dans le fichier, un seul affichage subsiste, comme s'il n'y avait eu qu'un processus exécuté, le processus père. Que s'est-il passé ? En fait, c'est à la fois le hasard et la logique. Le hasard a voulu que l'ordonnanceur de tâche fasse s'exécuter les instructions du processus père après celles du processus fils. Donc le processus fils ouvre le fichier, écrit dedans et le ferme. Puis le processus père ouvre le même fichier, avec un drapeau qui va placer le descripteur de fichier au début et donc vide le fichier de son contenu, le père écrit puis ferme le fichier. Toute trace du processus fils a disparu. Si notre logique est bonne, un drapeau tel que `"a+"` devrait remédier à cette situation. Voici la sortie lorsque l'on change le drapeau (n'oubliez pas d'effacer le fichier avant de lancer ce nouveau programme ! !) :

```
menthe22>rm -f curieux.txt
menthe22>clone2
[processus 18905] mon pere est : 18711.
[processus 18905] j'alloue
[processus 18905] je remplis
[processus 18905] je me clone.
menthe22>more curieux.txt
[processus 18906] Mon pere est : 18905
[processus 18906] Le numero enregistre : 18905
[processus 18906] tab_stat : 10
[processus 18906] tab_dyn : 20
[processus 18906] tab_stat : 11
[processus 18906] tab_dyn : 21
[processus 18905] Mon pere est : 18711
[processus 18905] Le numero enregistre : 18905
[processus 18905] tab_stat : 10
[processus 18905] tab_dyn : 20
[processus 18905] tab_stat : 11
[processus 18905] tab_dyn : 21
menthe22>
```

On retrouve bien le fait que nos deux processus écrivent dans le fichier [2].

Nous allons maintenant tenter une autre expérience, celle de suivre un peu les pas de Fibonacci ! Nous allons donc faire une boucle à l'intérieur de laquelle nous trouverons

2. Au fait à quoi servent les tableaux ? Un peu de patience nous allons les utiliser bientôt !

l'appel à `fork()`. Avant d'écrire ce programme nous allons tout d'abord réfléchir, car l'ordinateur étant avant tout un outil de travail, inutile de le détruire prématurément !

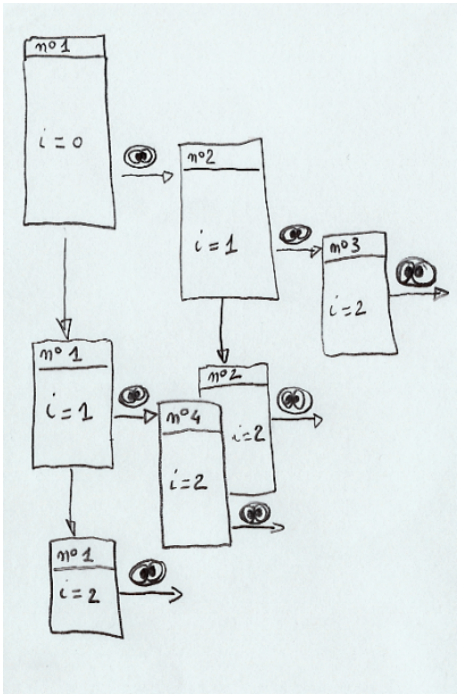


FIGURE 12.2 – *Chaque boîte représente un processus avec un numéro. Ainsi la première boîte représente le processus père qui crée un fils à la première étape de la boucle. Puis le père continue son exécution avec la deuxième étape et une nouvelle création de processus fils, et de son côté son premier fils crée son propre fils pour la deuxième étape de la boucle. Nous voyons que lors de la deuxième étape dans la boucle de chaque processus créé, quatre clones vont prendre naissance.*

Le programme ressemble singulièrement aux précédents, si ce n'est que les instructions d'écriture dans le fichier et surtout l'appel à `fork()` se trouvent maintenant dans une boucle de 3 itérations.

Listing 12.11 – *Les conséquences d'une boucle et d'un `fork`*

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
```

```
  int i;

  printf("[processus %5d] mon pere est : %d\n",
         getpid(),getppid());
  printf("[processus %5d] je vais me cloner\n\n",
         getpid());

  for(i=0;i<3;i++) {
    if (fork()==-1) {
      perror("Impossible de me cloner");
      exit(EXIT_FAILURE);
    }
    printf("[processus %5d] Mon pere est : %d\n",
           getpid(),getppid());
  }
  exit(EXIT_SUCCESS);
}
```

Voici la sortie :

```
menthe22>clone2b
[processus 18966] mon pere est : 18711.
[processus 18966] je vais me cloner

[processus 18967] Mon pere est : 18966
[processus 18968] Mon pere est : 18967
[processus 18969] Mon pere est : 18968
[processus 18968] Mon pere est : 18967
[processus 18967] Mon pere est : 18966
[processus 18970] Mon pere est : 18967
[processus 18967] Mon pere est : 18966
[processus 18966] Mon pere est : 18711
[processus 18971] Mon pere est : 18966
[processus 18972] Mon pere est : 18971
[processus 18971] Mon pere est : 18966
[processus 18966] Mon pere est : 18711
[processus 18973] Mon pere est : 18966
[processus 18966] Mon pere est : 18711
menthe22>
```

Au total 8 processus comme prévu, le père et ses trois fils, puis ses trois petits fils et enfin son petit petit fils. Conclusion, attention avant de mettre un appel à `fork()` dans une boucle, le nombre de processus devient très vite grand. Il y a bien sûr des limites, le système Linux par exemple, n'autorise pas plus de 256 processus fils créés, mais remarquons que le père n'a jamais créé que 3 fils, qui dès leur majorité atteinte ne se sont pas faits prier pour créer à leur tour des fils, donc la limite était loin d'être atteinte pour le père. Nous verrons à la fin de ce corrigé une façon de créer plusieurs fils.

Nous allons maintenant nous servir des tableaux par l'intermédiaire de la valeur retournée par `fork()`.

Nous savons que la fonction `fork()` duplique le processus courant, mais nous savons aussi qu'elle retourne une valeur, et les choses étant bien faites, la valeur retournée dépend du processus dans lequel cette valeur est prise en compte : s'il s'agit du processus père, la fonction renvoie une valeur non nulle qui est le `pid` du fils nouvellement créé, sinon, quand on se trouve dans le processus fils, la fonction renvoie une valeur nulle. Nous allons ainsi pouvoir gérer différentes exécutions selon le contexte.

Nous reprenons les programmes précédents pour bien voir que les zones mémoires sont totalement séparées. Dans le programme qui suit, remarquez que l'écriture dans le fichier des variables globales, statiques ou allouées dynamiquement est réalisée à la fin du programme par les deux processus. Par contre les modifications sur les valeurs de ces variables sont différentes selon que l'on se trouve dans le processus père (`fork()>0`) ou le fils (`fork()==0`).

Listing 12.12 – *Deux espaces totalement différents*

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

/* Cette variable est globale */
/* a tout le programme. C'est */
/* sale mais c'est pour la    */
/* bonne cause !!*/
int tab_stat[2];


int main(int argc, char **argv)
{
  FILE *fp;
  int *tab_dyn;
  int le_numero_au_debut;
  int i;

  le_numero_au_debut = getpid();
  printf("[processus %5d] mon pere est : %d.\n",
         getpid(),getppid());
  printf("[processus %5d] j'alloue\n",getpid());
  tab_dyn = malloc(2*sizeof(int));
  printf("[processus %5d] je remplis\n",getpid());

  tab_stat[0] = 10;
  tab_stat[1] = 11;

  tab_dyn[0] = 20;
  tab_dyn[1] = 21;

  printf("[processus %5d] je me clone\n",getpid());

  /* A partir de maintenant le code doit */
  /* etre execute plusieurs fois */
  if ((i=fork())==-1) {
    perror("Impossible de me cloner");
    exit(EXIT_FAILURE);
  }

  if (i==0) { /* je suis le fils */
    if ((fp = fopen("curieux.txt","a+")) == NULL) {
      perror("Impossible d'ouvrir curieux.txt");
      exit(EXIT_FAILURE);
    }
    le_numero_au_debut = 11;
    tab_stat[0] = 12;
    tab_stat[1] = 13;
    tab_dyn[0] = 22;
    tab_dyn[1] = 23;
```

```
    fprintf(fp,"[processus %5d] Fils de : %d\n",
            getpid(),getppid());
  } else { /* je suis le pere */
    if ((fp = fopen("curieux.txt","a+")) == NULL) {
      perror("Impossible d'ouvrir curieux.txt");
      exit(EXIT_FAILURE);
    }
    le_numero_au_debut = 101;
    tab_stat[0] = 102;
    tab_stat[1] = 103;
    tab_dyn[0] = 202;
    tab_dyn[1] = 203;

  }
  /* Ceci est execute par tout le monde */
  fprintf(fp,"[processus %5d] : %d %d %d %d %d\n",
          getpid(), le_numero_au_debut,
          tab_stat[0],tab_stat[1],
          tab_dyn[0],tab_dyn[1]);
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```

La sortie donne ceci :

```
menthe22>clone4
[processus 19086] mon pere est : 18711.
[processus 19086] j'alloue
[processus 19086] je remplis
[processus 19086] je me clone
menthe22>more curieux.txt
[processus 19087] Fils de : 19086
[processus 19087] : 11 12 13 22 23
[processus 19086] : 101 102 103 202 203
menthe22>
```

Nous voyons donc que les variables sont reproduites dans chaque processus et qu'elles appartiennent à des espaces mémoire différents.

Attendre ses fils

Nous allons finir en utilisant la fonction `wait()` qui permet au processus père d'attendre la fin de son fils avant de continuer son exécution. Le schéma du programme est simple, on reprend le code précédent et on ajoute, **dans la partie exécutée par le père**, l'instruction `wait()` avant toute autre instruction. On est alors certain que toutes les instructions spécifiques au père seront effectuées après que le fils ait terminé sa tâche. La sortie du programme donne la même chose que précédemment.

Vous avez remarqué que la fonction `wait()` prend une adresse d'entier comme argument. La fonction viendra placer à cet endroit le code de retour du processus fils, à savoir la valeur retournée par la fonction `main()` du fils.

Voici un petit programme avec un père… de famille.

Listing 12.13 – *Père de famille*

```
#include <sys/types.h>
```

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char **argv)
{
  int i,spid[2];
  int status;

  fprintf(stdout,"[%5d] je suis le pere\n",getpid());
  /* A partir de maintenant le code doit */
  /* etre execute plusieurs fois */
  if ((i=fork())==-1) {
    perror("Impossible de me cloner");
    exit(EXIT_FAILURE);
  }

  if (i==0) { /* je suis le fils */
    fprintf(stdout,"[F1] je suis le premier fils (%d)\n",
            getpid());
    fprintf(stdout,"[F1] je dors\n");
    sleep(4);
    exit(21);
  } else { /* je suis le pere */
    spid[0] = i;
    /* je cree un second fils */
    if ((i=fork())==-1) {
      perror("Impossible de me double-cloner");
      exit(EXIT_FAILURE);
    }
    if (i==0) { /* je suis le second fils */
      fprintf(stdout,"[F2] je suis le second fils (%d)\n",
              getpid());
      fprintf(stdout,"[F2] je dors\n");
      sleep(2);
      exit(31);
    } else {
      spid[1] = i;
      status = 0;
      /* je suis le pere */
      /* j'attends mes fils */

      i=wait(&status);
      fprintf(stdout,
              "[P0] mon fils F%c a retourne %d\n",
              i==spid[0]?'1':'2',WEXITSTATUS(status));
      i=wait(&status);
      fprintf(stdout,
              "[P0] mon fils F%c a retourne %d\n",
              i==spid[0]?'1':'2',WEXITSTATUS(status));
    }
  }
  fprintf(stdout,"[%5d] suis-je le pere?\n",getpid());
  exit(EXIT_SUCCESS);
}
```

La sortie donne ceci, sachant que `F1` est le premier fils, `F2` le deuxième fils et `P0` le père :

```
menthe22>clone5
[19273] je suis le pere
```

```
[F1] je suis le premier fils (19274)
[F1] je dors
[F2] je suis le second fils (19275)
[F2] je dors
[P0] mon fils F1 a retourne 21
[P0] mon fils F2 a retourne 31
[19273] suis-je le pere?
menthe22>
```

Nous voyons que le premier fils dort pendant 2 secondes. Le deuxième fils, créé après le premier `fork()`, dort quant à lui 4 secondes. Donc le premier appel `wait()` va sortir avec la valeur de retour du premier fils qui se sera réveillé en premier. Si l'on échange les deux valeurs (le premier fils dort 4 secondes et le deuxième 2 secondes) la sortie devient :

```
menthe22>clone5
[19273] je suis le pere
[F1] je suis le premier fils (19274)
[F1] je dors
[F2] je suis le second fils (19275)
[F2] je dors
[P0] mon fils F2 a retourne 31
[P0] mon fils F1 a retourne 21
[19273] suis-je le pere?
menthe22>
```

Regardons maintenant comment créer un nombre inconnu de fils à l'avance et que nous communiquerons au programme par la ligne de commandes. Nous allons observer une première construction dans laquelle nous voulons que chaque fils puisse dormir pendant un temps aléatoire. Voici le premier programme :

Listing 12.14 – *Attente* a priori *aléatoire...*

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_SON 12
int son(int num)
{
  int retf,t_s;
  if ((retf=fork())==-1) {
    perror("Impossible de me cloner");
    exit(EXIT_FAILURE);
  }
  if(retf) {
    return retf;
  }
  t_s = random()%10;
  fprintf(stdout,
    "[F%02d] je suis le fils de (%d) et je dors %ds\n",
    num,getppid(),t_s);
  sleep(t_s);
  exit(0);

}
```

```
int main(int argc, char **argv)
{
  int numf,i,*spid;
  int retf,status;
  srand(getpid());
  fprintf(stdout,"[%5d] je suis le pere\n",getpid());
  numf = (argc>1?atoi(argv[1]):2);
  numf = numf<=0?2:(numf>MAX_SON?MAX_SON:numf);
  spid = malloc(numf*sizeof(int));
  i=0;
  while(i<numf) {
    spid[i] = son(i);i++;
  }

  while((retf=wait(&status))!=-1) {
    for(i=0;i<numf;i++) {
      if (retf == spid[i]) {
        fprintf(stdout,
                "[P0] mon fils (F%02d) s'est termine\n",i);
      }
    }
  }
  fprintf(stdout,"[%5d] suis-je le pere?\n",getpid());
  exit(EXIT_SUCCESS);
}
```

Nous obtenons la sortie suivante :

```
menthe22>clone6 4
[20516] je suis le pere
[F00] je suis le fils de (20516) et je dors 6s
[F01] je suis le fils de (20516) et je dors 6s
[F02] je suis le fils de (20516) et je dors 6s
[F03] je suis le fils de (20516) et je dors 6s
[P0] mon fils (F00) s'est termine
[P0] mon fils (F01) s'est termine
[P0] mon fils (F02) s'est termine
[P0] mon fils (F03) s'est termine
[20516] suis-je le pere?
```

Quelle admirable médiocrité de la part de `random()`. Tous les fils ont exactement le même temps de sommeil, franchement, impossible de faire confiance à un générateur aléatoire ! À moins que notre programme soit mal écrit ?

En fait c'est la deuxième solution qui convient ! Le programme est mal écrit ! Un processus de départ, le père et donc une seule initialisation de la graine par l'appel `srand(getpid())`. Ensuite chaque fils fait appel à la fonction `random()` une et une seule fois, donc cette fonction est exécutée de manière totalement identique par chaque fils, rappelons-nous que chaque fils est une copie « à l'identique », comme si la boule du loto avait été duppliquée de façon exacte à un instant du brassage. Chaque copie nous donnera donc le même tirage !

Changeons tout cela à l'aide du programme suivant :

Listing 12.15 – *Attente vraiment aléatoire*

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SON 12
int son(int num, int t_s)
{
  int retf;
  if ((retf=fork())==-1) {
    perror("Impossible de me cloner");
    exit(EXIT_FAILURE);
  }
  if(retf) {
    return retf;
  }
  fprintf(stdout,
     "[F%02d] je suis le fils de (%d) et je dors %ds\n",
     num,getppid(),t_s);
  sleep(t_s);
  exit(0);

}

int main(int argc, char **argv)
{
  int numf,i,*spid;
  int retf,status;
  int t_s;
  srand(getpid());
  fprintf(stdout,"[%5d] je suis le pere\n",getpid());
  numf = (argc>1?atoi(argv[1]):2);
  numf = numf<=0?2:(numf>MAX_SON?MAX_SON:numf);
  spid = malloc(numf*sizeof(int));
  i=0;
  while(i<numf) {
    t_s = random()%5;
    spid[i] = son(i,t_s);i++;
  }

  while((retf=wait(&status))!=-1) {
    for(i=0;i<numf;i++) {
      if (retf == spid[i]) {
        fprintf(stdout,
                "[P0] mon fils (F%02d) s'est termine\n",i);
      }
    }
  }
  fprintf(stdout,"[%5d] suis-je le pere?\n",getpid());
  exit(EXIT_SUCCESS);
}
```

Nous obtenons la sortie suivante :

```
menthe22>clone7 4
[20671] je suis le pere
[F00] je suis le fils de (20671) et je dors 0s
[F01] je suis le fils de (20671) et je dors 1s
[F02] je suis le fils de (20671) et je dors 4s
[F03] je suis le fils de (20671) et je dors 2s
[P0] mon fils (F00) s'est termine
[P0] mon fils (F01) s'est termine
[P0] mon fils (F03) s'est termine
[P0] mon fils (F02) s'est termine
[20671] suis-je le pere?
menthe22>
```

13

Recouvrement de processus sous Unix

Dans le chapitre précédent, nous avons vu comment créer de nouveaux processus grâce à l'appel système `fork()`. Cet appel système effectue une duplication de processus, c'est-à-dire que le processus nouvellement créé contient les mêmes données et exécute le même code que le processus père. Le recouvrement de processus permet de remplacer par un autre code le code exécuté par un processus. Le programme et les données du processus sont alors différents, mais celui-ci garde le même `pid`, le même père et les même descripteurs de fichiers.

C'est ce mécanisme qui est utilisé lorsque, par exemple, nous tapons la commande **`ls`**, ou que nous lançons un programme après l'avoir compilé en tapant **`./a.out`** :

- le terminal de commande dans lequel cette commande est tapée fait un `fork()` ;
- le processus ainsi créé (le fils du terminal de commande) est recouvert par l'exécutable désiré ;
- pendant ce temps, le terminal de commande (le père) attend que le fils se termine grâce à `wait()` si la commande n'a pas été lancée en tâche de fond.

13.1 Les appels système de recouvrement de processus

Ces appels système sont au nombre de 6 :

- `int execl(char *path, char *arg0, char *arg1, ... , char *argn, (char *)0)`
- `int execv(char *path, char *argv[])`
- `int execle(char *path, char *arg0, char *arg1, ... , char *argn, (char *)0, char **envp)`
- `int execlp(char *file, char *arg0, char *arg1, ... , char *argn, (char *)0)`
- `int execvp(char *file, char *argv[])`
- `int execve(char *path, char *argv[], char **envp)`

Notons que le code présent après l'appel à une de ces fonctions ne sera jamais exécuté, sauf en cas d'erreur. Nous décrirons ici uniquement les appels système `execv()` et `execlp()` :

– `int execv(char *path, char *argv[])`
  Cette fonction recouvre le processus avec l'exécutable indiqué par la chaîne de caractère `path` (`path` est le chemin d'accès à l'exécutable). `argv` est un tableau de chaînes de caractères contenant les arguments à passer à l'exécutable. Le dernier élément du tableau doit contenir `NULL`. Le premier est la chaîne de caractères qui sera affichée par **`ps`** et, par convention, on y met le nom de l'exécutable.

  Exemple :

```
...
argv[0] = "ls";
argv[1] = "-l";
argv[2] = NULL;
execv("/bin/ls", argv);
```

– `int execlp(char *file, char *arg0, char *arg1, ... , char *argn, (char *)0)`
  Cette fonction recouvre le processus avec l'exécutable spécifié par `file` (le fichier utilisé est cherché dans les répertoires contenus dans la variable d'environnement `$PATH` si la chaîne `file` n'est pas un chemin d'accès absolu ou relatif). Les arguments suivants sont les chaînes de caractères passées en argument à l'exécutable. Comme pour `execv()`, le premier argument doit être le nom de l'exécutable et le dernier `NULL`.

```
execlp("ls","ls","-l",NULL);
```

  Notons qu'il n'est pas nécessaire ici de spécifier le chemin d'accès au fichier (**`/bin/ls`**) pour l'exécutable.

Pour recouvrir un processus avec la commande **`ps -aux`**, nous pouvons utiliser le code suivant :

Listing 13.1 – *Un premier recouvrement*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    char *arg[3];
    arg[0] = "ps";
    arg[1] = "-aux";
    arg[2] = NULL;
    execv("/bin/ps", arg); /* l'executable ps se trouve dans /bin */
    fprintf(stderr, "erreur dans execv\n");
    exit(EXIT_FAILURE);
}
```

ou alors :

Listing 13.2 – *Un recouvrement plus simple*

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    execlp("ps", "ps", "-aux", NULL);
    fprintf(stderr, "erreur dans execlp\n");
    exit(EXIT_FAILURE);
}
```

Dans le premier chapitre consacré à la programmation système, nous utilisions les variables passées à la fonction `int main(int argc, char **argv)` pour récupérer les arguments de la ligne de commande. Ces valeurs sont en fait déterminées par le terminal de commande et c'est lui qui les « passe » à la commande exécutée, grâce à un appel à `exec()`. Ainsi, lorsque nous tapons une commande comme **`ps -aux`**, le terminal de commande exécute (de manière simplifiée) :

```c
int res;
...
res = fork();

if (res == 0) {/* on est le fils */
   execlp("ps", "ps", "-aux", NULL);
   fprintf(stderr, "erreur\n");
   /* si execlp retourne, quelque chose ne va pas */
   exit(EXIT_FAILURE);
}
```

13.2 Exercices

Question 1

Créer un processus et le recouvrir par la commande **`du -s -h`**. Attention, ce programme n'a pas besoin d'utiliser l'appel système `fork()`.

Question 2

Reprendre le premier exemple de la série d'exercices précédente (l'application affichant sont `pid` et celui de son père). Nous appellerons ce programme `identite`. Écrire ensuite une application (semblable à celle de la question 2 de la série d'exercices précédente) créant un processus fils. Le père affichera son identité, le processus fils sera recouvert par le programme `identite`.

Question 3

Écrire une application `myshell` reproduisant le comportement d'un terminal de commande. Cette application doit lire les commandes tapées au clavier, ou contenue dans un fichier, créer un nouveau processus et le recouvrir par la commande lue. Par exemple :

```
menthe22> ./myshell
ls
extp3_1.c       extp3_2.c       identite        myshell
soltp3_1.c      soltp3_2.c      soltp3_3.c
ps
```

```
  PID  TT  STAT      TIME COMMAND
  521 std  S      0:00.95 bash
  874  p3  S      0:00.02 bash
 1119  p3  S+     0:00.56 vi soltp3_3.c
 1280  p2  S      0:00.03 tcsh
 1283  p2  S+     0:00.04 vi
 1284 std  S+     0:00.01 ./myshell
exit
menthe22>
```

Pour la commodité d'utilisation, on pourra faire en sorte que le terminal de commande affiche une invite (un *prompt*) avant de lire une commande :

```
menthe22>./myshell
myshell>ls
extp3_1.c       extp3_2.c       identite        myshell
soltp3_1.c      soltp3_2.c      soltp3_3.c
myshell>exit
menthe22>
```

On pensera à utiliser la fonction `wait()` afin d'attendre que l'exécution d'une commande soit terminée avant de lire et de lancer la suivante.

Que se passe-t-il si l'on tape à l'invite **`ls -l`** ?

Question 4

Ajouter au terminal de commande de la question précédente la gestion des paramètres pour permettre l'exécution de **`ls -l`** par exemple. Est-ce que la ligne **`ls *.c`** se comporte comme un vrai terminal de commande ?

Question 5

Ajouter au terminal de commande de la question précédente la gestion des exécutions en tâche de fond, caractérisée par la présence du caractère & en fin de ligne de commande.

13.3 Corrigés

Listing 13.3 – *Correction du premier exercice*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
        execlp("du", "du", "-s", "-h", NULL);
        fprintf(stderr, "erreur dans execlp\n");
        exit(EXIT_FAILURE);
}
```

Listing 13.4 – *Correction du deuxième exercice*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int s;

  s = fork();
  if (s < 0) { /* erreur */
    fprintf(stderr,"erreur dans fork\n");
    exit(EXIT_FAILURE);
  }
  if (s == 0) { /* on est le fils */
    execlp("./identite", "identite", NULL);
    fprintf(stderr, "erreur dans le execlp\n");
    exit(EXIT_FAILURE);
  } else { /* on est le pere */
    printf(" Je suis le pere, mon pid est %d,");
    printf("celui de mon fils est %d\n",
           getpid(), s);
    exit(EXIT_SUCCESS);
  }

}
```

Listing 13.5 – *Correction du troisième exercice*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc , char *argv[]) {
   char ligne[80], *arg[2];

   printf("myshell> ") ;
   while (fgets(ligne,sizeof(ligne),stdin) != NULL) {
      /* fgets lit egalement le caractere de fin de ligne */
      if (strcmp(ligne,"exit\n") == 0) {
         exit(EXIT_SUCCESS);
      }
      /* on supprime le caractere de fin de ligne */
      ligne[strlen(ligne)-1]='\0';
```

```
      arg[0] = ligne; /* nom de l'executable */
      arg[1] = NULL; /* fin des parametres */

      /* creation du processus qui execute la commande */
      switch (fork()) {
      case -1:   /* erreur */
         fprintf(stderr,"Erreur dans fork()\n");
         exit(EXIT_FAILURE);
      case 0:   /* fils */
         /* on recouvre par la commande tapee */
         execvp(arg[0],arg);
         /* on n'arrivera jamais ici, sauf en cas d'erreur */
         fprintf(stderr,"Erreur dans execvp(\"%s\")\n",arg[0]);
         exit(EXIT_FAILURE);
      default:   /* pere */
         /* on attend la fin du fils avant de representer */
         /* l'invite de commande */
         wait(NULL);
      }
      printf("myshell> ");
   } /* fin du while */
   exit(EXIT_SUCCESS);
}
```

En utilisant la fonction `strtok()` on peut écrire le *shell* comme ceci :

Listing 13.6 – *Construction d'un* shell

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc , char *argv[]) {
   char ligne[80], *arg[10], *tmp;
   int i;

   printf("myshell> ") ;
   /* lecture de l'entree standard ligne par ligne */
   while (fgets(ligne,sizeof(ligne),stdin) != NULL) {
      /* fgets lit egalement le caractere de fin de ligne */
      if (strcmp(ligne,"exit\n") == 0) {
         exit(EXIT_SUCCESS);
      }
      /* decoupage de la ligne aux espaces et tabulations */
      /* un premier appel strtok() retourne le premier parametre */
      /* chaque appel suivant se fait avec NULL comme premier */
      /* argument, et retourne une chaine, ou NULL lorsque */
      /* c'est la fin */
      for (tmp=strtok(ligne," \t\n"), i=0;
            tmp != NULL ;
            tmp=strtok(NULL," \t\n"), i++) {
         /* on remplit le tableau arg[] */
         /* une simple affectation de pointeurs */
         /* suffit car strtok() coupe la chaine ligne */
         /* sans copie et sans ecraser les arg[] precedents */
         arg[i] = tmp;
      }
      arg[i] = NULL; /* fin des parametres */

      /* creation du processus qui execute la commande */
      switch (fork()) {
```

```
      case -1:   /* erreur */
        fprintf(stderr,"Erreur dans fork()\n");
        exit(EXIT_FAILURE);
      case 0:   /* fils */
        /* on recouvre par la commande tapee */
        execvp(arg[0],arg);
        /* on n'arrivera jamais ici, sauf en cas d'erreur */
        fprintf(stderr,"Erreur dans execvp(\"%s\")\n",arg[0]);
        exit(EXIT_FAILURE);
      default:   /* pere */
        /* on attend la fin du fils avant de representer */
        /* l'invite de commande */
        wait(NULL);
      }
      printf("myshell> ");
   } /* fin du while */
   exit(EXIT_SUCCESS);
}
```

Voici un exemple d'une autre solution n'utilisant pas `strtok` :

Listing 13.7 – *Construction plus longue d'un* shell

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <ctype.h>

void decoupe_commande(char *ligne, char *arg[]);

int main (int argc , char *argv[]) {
   char ligne[80], *arg[10];

   printf("myshell> ") ;
   /* lecture de l'entree standard ligne par ligne */
   while (fgets(ligne,sizeof(ligne),stdin) != NULL) {
      /* fgets lit egalement le caractere de fin de ligne */
      if (strcmp(ligne,"exit\n") == 0) {
         exit(EXIT_SUCCESS);
      }

      /* La fonction qui decoupe la ligne de texte */
      decoupe_commande(ligne, arg);
      /* creation du processus qui execute la commande */
      switch (fork()) {
       case -1:   /* erreur */
         fprintf(stderr,"Erreur dans fork()\n");
         exit(EXIT_FAILURE);
       case 0:   /* fils */
         /* on recouvre par la commande tapee */
         execvp(arg[0],arg);
         /* on n'arrivera jamais ici, sauf en cas d'erreur */
         fprintf(stderr,"Erreur dans execvp(\"%s\")\n",arg[0]);
         exit(EXIT_FAILURE);
       default:   /* pere */
         /* on attend la fin du fils avant de representer */
         /* l'invite de commande */
         wait(NULL);
      }
      printf("myshell> ");
   } /* fin du while */
```

```
   exit(EXIT_SUCCESS);
}

/* Decoupage de la ligne en place (sans recopie) */
void decoupe_commande(char *ligne, char *arg[]) {
    int i=0;
    /* on avance jusqu'au premier argument */
    while (isspace(*ligne)) ligne++;
    /* on traite les arguments tant que ce n'est */
    /* pas la fin de la ligne */
    while (*ligne != '\0' && *ligne != '\n') {
        arg[i++]=ligne;
        /* on avance jusqu'au prochain espace */
        while (!isspace(*ligne) && *ligne!='\0')
            ligne++;
        /* on remplace les espaces par '\0' ce qui marque */
        /* la fin du parametre precedent */
        while (isspace(*ligne) && *ligne!='\0')
            *ligne++='\0';
    }
    arg[i]=NULL;
}
```

14

Manipulation des signaux sous Unix

Les signaux constituent la forme la plus simple de communication entre processus [1].

Un signal est une information atomique envoyée à un processus ou à un groupe de processus par le système d'exploitation ou par un autre processus. Lorsqu'un processus reçoit un signal, le système d'exploitation l'informe : « tu as reçu tel signal », sans plus. Un signal ne transporte donc aucune autre information utile.

Lorsqu'il reçoit un signal, un processus peut réagir de trois façons :

– Il est immédiatement dérouté vers une fonction spécifique, qui réagit au signal (en modifiant la valeur de certaines variables ou en effectuant certaines actions, par exemple). Une fois cette fonction terminée, on reprend le cours normal de l'exécution du programme, comme si rien ne s'était passé.
– Le signal est tout simplement ignoré.
– Le signal provoque l'arrêt du processus (avec ou sans génération d'un fichier `core`).

Lorsqu'un processus reçoit un signal pour lequel il n'a pas indiqué de fonction de traitement, le système d'exploitation adopte une réaction par défaut qui varie suivant les signaux :

– soit il ignore le signal ;
– soit il termine le processus (avec ou sans génération d'un fichier `core`).

Vous avez certainement toutes et tous déjà utilisé des signaux, consciemment, en tapant Control-C ou en employant la commande **`kill`**, ou inconsciemment, lorsqu'un de vos programme a affiché

```
segmentation fault (core dumped)
```

14.1 Liste et signification des différents signaux

La liste des signaux dépend du type d'Unix. La norme POSIX.1 en spécifie un certain nombre, parmi les plus répandus. On peut néanmoins dégager un grand nombre de signaux communs à toutes les versions d'Unix :

1. Au niveau du microprocesseur, un signal correspond à une interruption logicielle.

`SIGHUP` : rupture de ligne téléphonique. Du temps où certains terminaux étaient reliés par ligne téléphonique à un ordinateur distant, ce signal était envoyé aux processus en cours d'exécution sur l'ordinateur lorsque la liaison vers le terminal était coupée. Ce signal est maintenant utilisé pour demander à des démons (processus lancés au démarrage du système et tournant en tâche de fond) de relire leur fichier de configuration.

`SIGINT` : interruption (c'est le signal qui est envoyé à un processus quand on tape Control-C au clavier).

`SIGFPE` : erreur de calcul en virgule flottante, le plus souvent une division par zéro.

`SIGKILL` : tue le processus.

`SIGBUS` : erreur de bus.

`SIGSEGV` : violation de segment, généralement à cause d'un pointeur nul.

`SIGPIPE` : tentative d'écriture dans un tuyau qui n'a plus de lecteurs.

`SIGALRM` : alarme (chronomètre).

`SIGTERM` : demande au processus de se terminer proprement.

`SIGCHLD` : indique au processus père qu'un de ses fils vient de se terminer.

`SIGWINCH` : indique que la fenêtre dans lequel tourne un programme a changé de taille.

`SIGUSR1` : signal utilisateur 1.

`SIGUSR2` : signal utilisateur 2.

Il n'est pas possible de dérouter le programme vers une fonction de traitement sur réception du signal `SIGKILL`, celui-ci provoque toujours la fin du processus. Ceci permet à l'administrateur système de supprimer n'importe quel processus.

À chaque signal est associé un numéro. Les correspondances entre numéro et nom des signaux se trouvent généralement dans le fichier `/usr/include/signal.h` ou dans le fichier `/usr/include/sys/signal.h` suivant le système.

14.2 Envoi d'un signal

Depuis un interprète de commandes

La commande `kill` permet d'envoyer un signal à un processus dont on connaît le numéro (il est facile de le déterminer grâce à la commande **`ps`**) :

```
kill -HUP 1664
```

Ici, on envoie le signal `SIGHUP` au processus numéro 1664 (notez qu'en utilisant la commande **`kill`**, on écrit le nom du signal sous forme abrégée, sans le `SIG` initial).

On aurait également pu utiliser le numéro du signal plutôt que son nom :

```
kill -1 1664
```

On envoie le signal numéro 1 (SIGHUP) au processus numéro 1664.

Depuis un programme en C

La fonction C kill() permet d'envoyer un signal à un processus :

```
#include <sys/types.h>
#include <signal.h>

/* ... */

pid_t pid = 1664 ;

/* ... */

if ( kill(pid,SIGHUP) == -1 ) {
   /* erreur : le signal n'a pas pu etre envoye */
}
```

Ici, on envoie le signal SIGHUP au processus numéro 1664.

On aurait pu directement mettre 1 à la place de SIGHUP, mais l'utilisation des noms des signaux rend le programme plus lisible et plus portable (le signal SIGHUP a toujours le numéro 1 sur tous les systèmes, mais ce n'est pas le cas de tous les signaux).

14.3 Interface de programmation

L'interface actuelle de programmation des signaux (qui respecte la norme POSIX.1) repose sur la fonction sigaction(). L'ancienne interface, qui utilisait la fonction signal(), est à proscrire pour des raisons de portabilité. Nous en parlerons néanmoins rapidement pour indiquer ses défauts.

La fonction sigaction()

La fonction sigaction() indique au système comment réagir sur réception d'un signal. Elle prend comme paramètres :

1. Le numéro du signal auquel on souhaite réagir.
2. Un pointeur sur une structure de type sigaction. Dans cette structure, deux membres nous intéressent :
   - sa_handler, qui peut être :
     - un pointeur vers la fonction de traitement du signal ;
     - SIG_IGN pour ignorer le signal ;
     - SIG_DFL pour restaurer la réaction par défaut.
   - sa_flags, qui indique des options liées à la gestion du signal. Étant donné l'architecture du noyau Unix, un appel système interrompu par un signal est toujours avorté et renvoie EINTR au processus appelant. Il faut alors relancer

cet appel système. Néanmoins, sur les Unix modernes, il est possible de demander au système de redémarrer automatiquement certains appels système interrompus par un signal. La constante `SA_RESTART` est utilisée à cet effet.

Le membre `sa_mask` de la structure `sigaction` indique la liste des signaux devant être bloqués pendant l'exécution de la fonction de traitement du signal. On ne veut généralement bloquer aucun signal, c'est pourquoi on initialise `sa_mask` à zéro au moyen de la fonction `sigemptyset()`.

3. Un pointeur sur une structure de type `sigaction`, structure qui sera remplie par la fonction selon l'ancienne configuration de traitement du signal. Ceci ne nous intéresse pas ici, d'où l'utilisation d'un pointeur nul.

La valeur renvoyée par `sigaction()` est :
- 0 si tout s'est bien passé.
- -1 si une erreur est survenue. Dans ce cas l'appel à `sigaction()` est ignoré par le système.

Ainsi, dans l'exemple ci-dessous, à chaque réception du signal `SIGUSR1`, le programme sera dérouté vers la fonction `TraiteSignal()`, puis reprendra son exécution comme si de rien n'était. En particulier, les appels système qui auraient été interrompus par le signal seront relancés automatiquement par le système.

Listing 14.1 – *Traitement d'un signal*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* prototype de la fonction gestionnaire de signal */
/* le parametre est de type int (impose), et il n'y */
/* a pas de valeur de retour */
void TraiteSignal(int sig);

int main(int argc, char *argv[]) {
   struct sigaction act;

   /* une affectation de pointeur de fonction */
   /* c'est equivalent d'ecrire act.sa_handler = &TraiteSignal */
   act.sa_handler = TraiteSignal;
   /* le masque (ensemble) des signaux non pris en compte est mis */
   /* a l'ensemble vide (aucun signal n'est ignore) */
   sigemptyset(&act.sa_mask);
   /* Les appels systemes interrompus par un signal */
   /* seront repris au retour du gestionnaire de signal */
   act.sa_flags = SA_RESTART;
   /* enregistrement de la reaction au SIGUSR1 */
   if ( sigaction(SIGUSR1,&act,NULL) == -1 ) {
      /* perror permet d'afficher la chaine avec */
      /* le message d'erreur de la derniere commande */
      perror("sigaction");
      exit(EXIT_FAILURE);
   }

   printf("Je suis le processus numero %d.\n" ,getpid());
```

```
   for(;;) { /* boucle infinie equivalente a while (1) */
      sigset_t sigmask; /* variable locale a cette boucle */
      sigemptyset(&sigmask); /* mask = ensemble vide */
      /* on interromp le processus jusqu'a l'arrivee d'un signal */
      /* (mask s'il n'etait pas vide correspondrait aux signaux ignores) */
      sigsuspend(&sigmask);
      printf("Je viens de recevoir un signal et de le traiter\n");
   }

   exit(EXIT_SUCCESS);
}

void TraiteSignal(int sig) {
   printf("Reception du signal numero %d.\n", sig);
}
```

La fonction `sigsuspend()` utilisée dans la boucle infinie permet de suspendre l'exécution du programme jusqu'à réception d'un signal. Son argument est un pointeur sur une liste de signaux devant être bloqués. Comme nous ne désirons bloquer aucun signal, cette liste est mise à zéro au moyen de la fonction `sigemptyset()`. Les programmes réels se contentent rarement d'attendre l'arrivée d'un signal sans rien faire, c'est pourquoi la fonction `sigsuspend()` est assez peu utilisée dans la réalité.

La fonction `signal()`

La vieille interface de traitement des signaux utilise la fonction `signal()` au lieu de `sigaction()`.

Listing 14.2 – *Ancienne interface de traitement des signaux*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void TraiteSignal (int sig);

int main(int argc, char *argv[]) {
   if ( signal(SIGUSR1,TraiteSignal ) == SIG_ERR ) {
      perror("signal");
      exit(EXIT_FAILURE);
   }
   printf("Je suis le processus numero %d.\n", getpid());
   for (;;) {
      pause();
   }
   exit(EXIT_SUCCESS);
}

void TraiteSignal (int sig) {
   printf("Reception du signal numero %d.\n", sig);
}
```

La fonction `signal()` indique au système comment réagir sur réception d'un signal. Elle prend comme paramètres :

1. Le numéro du signal auquel on s'intéresse.

2. Une valeur qui peut être :
   – un pointeur vers la fonction de traitement du signal ;
   – `SIG_IGN` pour ignorer le signal ;
   – `SIG_DFL` pour restaurer la réaction par défaut.

De manière évidente, la fonction `signal()` est plus limitée que la nouvelle interface offerte par la fonction `sigaction()` parce qu'elle ne permet pas d'indiquer les options de traitement du signal, en particulier le comportement que l'on souhaite pour les appels système interrompus par la réception du signal.

Elle a aussi un effet de bord très vicieux par rapport à la fonction `sigaction`, c'est pourquoi il ne faut plus utiliser la fonction `signal()`.

La norme POSIX.1 (à laquelle la fonction `sigaction()` est conforme) spécifie que, si une fonction de traitement a été indiquée pour un signal, elle doit être appelée à chaque réception de ce signal et c'est bien ce qui se passe lorsqu'on utilise `sigaction()`.

En revanche, lorsqu'on utilise la fonction `signal()`, le comportement du système peut être différent :

– Sur les Unix BSD, tout se passe comme si l'on avait utilisé `sigaction()` et la fonction de traitement est bien appelée chaque fois qu'on reçoit le signal.
– Sur les Unix System V, en revanche, la fonction de traitement est bien appelée la première fois qu'on reçoit le signal, mais elle ne l'est pas si on le reçoit une seconde fois. Il faut alors refaire un appel à la fonction `signal()` dans la fonction de traitement pour rafraîchir la mémoire du système.

Cette différence de comportement oblige à gérer les signaux de deux manières différentes suivant la famille d'Unix.

14.4 Conclusion

Les signaux sont la forme la plus simple de communication entre processus. Cependant, ils ne permettent pas d'échanger des données.

En revanche, de par leur traitement asynchrone, ils peuvent être très utiles pour informer les processus de conditions exceptionnelles (relecture de fichier de configuration après modification manuelle, par exemple) ou pour synchroniser des processus.

14.5 Exercices

Question 1

Écrire un programme **`sigusr1`** qui, sur réception du signal `SIGUSR1`, affiche le texte « `Reception du signal SIGUSR1` ». Le programme principal sera de la forme :

```
for (;;) { /* boucle infinie equivalente a while (1) {} */
   sigset_t sigmask ;
   sigemptyset(&sigmask);
   sigsuspend(&sigmask);
}
```

et il serait judicieux de faire afficher au programme son numéro de processus à son lancement.
Fonction à utiliser :

```
#include <signal.h>
int sigaction(int sig, struct sigaction *act, struct sigaction *oact)
```

Question 2

La fonction `alarm()` permet de demander au système d'exploitation d'envoyer au processus appelant un signal `SIGALRM` après un nombre de secondes donné.

Écrire un programme `alarm` qui demande à l'utilisateur de taper un nombre. Afin de ne pas attendre indéfiniment, un appel à `alarm()` sera fait juste avant l'entrée du nombre pour que le signal `SIGALRM` soit reçu au bout de 5 secondes.

Dans ce cas, on veut que l'appel système de lecture du clavier soit interrompu lors de la réception du signal, il faut donc initialiser le membre `sa_flags` de la structure `sigaction` à 0.
Fonctions à utiliser :

```
#include <signal.h>
#include <unistd.h>
int sigaction(int sig, struct sigaction *act, struct sigaction *oact)
int alarm(int secondes)
```

Question 3

Écrire un programme `sigchld` qui appelle `fork()`. Le père mettra en place une fonction pour traiter le signal `SIGCHLD` (qui affichera, par exemple, « `Reception du signal SIGCHLD` »), attendra 10 secondes, puis affichera « `Fin du pere` ». Le fils affichera « `Fin du fils dans 2 sec` », puis se terminera deux secondes après.

Quel est l'intérêt du signal `SIGCHLD` ? Qu'est-ce qu'un zombie ?

14.6 Corrigés

Listing 14.3 – *Réception d'un signal utilisateur*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void TraiteSignal(int sig) {
   printf("Reception du signal SIGUSR1\n");
}

int main(int argc, char *argv[]) {
   struct sigaction act;

   /* remplissage de la structure sigaction */
   act.sa_handler = TraiteSignal;
   sigemptyset(&act.sa_mask);
   act.sa_flags = SA_RESTART;
   /* enregistrement de la reaction au SIGUSR1 */
   /* on ne s'interesse pas au precedent gestionnaire de SIGUSR1 */
   /* d'ou le troisieme argument a NULL */
   if ( sigaction(SIGUSR1,&act,NULL) == -1 ) {
      perror("sigaction");
      exit(EXIT_FAILURE);
   }
   printf("On peut m'envoyer un signal SIGUSR1 au moyen de la commande\n");
   printf("kill -USR1 %d\n",getpid());

   for (;;) { /* boucle infinie en attente d'un signal */
      sigset_t sigmask;
      sigemptyset(&sigmask);
      /* le processus est bloque sans consommer de temps processeur */
      /* jusqu'a l'arrivee d'un signal. Une simple boucle while (1) {}; */
      /* fonctionnerait aussi mais consommerait du temps processeur */
      sigsuspend(&sigmask);
      printf("Je viens de recevoir un signal et de le traiter\n");
   }
   exit(EXIT_SUCCESS);
}
```

Listing 14.4 – *Temporisation d'une saisie*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void TraiteSignal(int sig) {
   printf("\nReception du signal SIGALRM\n");
}

int main(int argc, char *argv[]) {
   struct sigaction act;
   int nombre = 1664;

   /* remplissage de la structure sigaction */
   act.sa_handler = TraiteSignal;
```

```
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0; /* un appel systeme interrompu n'est pas repris */
   if ( sigaction(SIGALRM,&act,NULL) == -1 ) {
      perror("sigaction");
      exit(EXIT_FAILURE);
   }

   printf("Entrez un nombre [ce sera %d par defaut dans 5 sec] : ",nombre);
   /* fflush permet de forcer l'affichage a l'ecran */
   /* alors que normalement la bibliotheque standard attend le \n */
   fflush(stdout);

   /* demande le reveil dans 5 sec */
   alarm(5);
   /* scanf() se base sur un appel read(), qui sera interrompu */
   /* par le SIGALARM s'il arrive pendant son execution */
   scanf("%d",&nombre);

   /* on eteint le reveil */
   alarm(0);
   printf("Vous avez choisi la valeur %d\n",nombre);

   exit(EXIT_SUCCESS);
}
```

Listing 14.5 – *Dialogue père / fils par signaux*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void TraiteSignal(int sig) {
   printf("Reception du signal SIGCHLD\n");
}

int main(int argc, char *argv[]) {
   struct sigaction act ;
   sigset_t sigmask ;

   act.sa_handler = TraiteSignal;
   sigemptyset(&act.sa_mask);
   act.sa_flags = SA_RESTART;
   if ( sigaction(SIGCHLD,&act,NULL) == -1 ) {
      perror("sigaction");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils */
      /* je ne fais rien, je meurs vite */
      printf("Fin du fils dans 2 sec\n");
      sleep(2);
      exit(EXIT_SUCCESS);
    default :   /* processus pere */
      sigemptyset(&sigmask);
      sigsuspend(&sigmask);
      printf("Un signal a ete recu, fin dans 10sec\n");
      sleep(10);
```

```
      printf("Fin du pere\n");
      exit(EXIT_SUCCESS);
   } /* switch */
}
```

15

Les tuyaux sous Unix

Les tuyaux [1] permettent à un groupe de processus d'envoyer des données à un autre groupe de processus. Ces données sont envoyées directement en mémoire sans être stockées temporairement sur disque, ce qui est donc très rapide.

Tout comme un tuyau de plomberie, un tuyau de données a deux côtés : un côté permettant d'écrire des données dedans et un côté permettant de les lire. Chaque côté du tuyau est un descripteur de fichier ouvert soit en lecture soit en écriture, ce qui permet de s'en servir très facilement, au moyen des fonctions d'entrée / sortie classiques.

La lecture d'un tuyau est bloquante, c'est-à-dire que si aucune donnée n'est disponible en lecture, le processus essayant de lire le tuyau sera suspendu (il ne sera pas pris en compte par l'ordonnanceur et n'occupera donc pas inutilement le processeur) jusqu'à ce que des données soient disponibles. L'utilisation de cette caractéristique comme effet de bord peut servir à synchroniser des processus entre eux (les processus lecteurs étant synchronisés sur les processus écrivains).

La lecture d'un tuyau est destructrice, c'est-à-dire que si plusieurs processus lisent le même tuyau, toute donnée lue par l'un disparaît pour les autres. Par exemple, si un processus écrit les deux caractères `ab` dans un tuyau lu par les processus A et B et que A lit un caractère dans le tuyau, il lira le caractère `a` qui disparaîtra immédiatement du tuyau sans que B puisse le lire. Si B lit alors un caractère dans le tuyau, il lira donc le caractère `b` que A, à son tour, ne pourra plus y lire. Si l'on veut donc envoyer des informations identiques à plusieurs processus, il est nécessaire de créer un tuyau vers chacun d'eux.

De même qu'un tuyau en cuivre a une longueur finie, un tuyau de données à une capacité finie. Un processus essayant d'écrire dans un tuyau plein se verra suspendu en attendant qu'un espace suffisant se libère.

Vous avez sans doute déjà utilisé des tuyaux. Par exemple, lorsque vous tapez

```
menthe22> ls | wc -l
```

l'interprète de commandes relie la sortie standard de la commande **`ls`** à l'entrée standard de la commande **`wc`** au moyen d'un tuyau.

1. Le terme anglais est *pipe*, que l'on traduit généralement par *tuyau* ou *tube*.

Les tuyaux sont très utilisés sous UNIX pour faire communiquer des processus entre eux. Ils ont cependant deux contraintes :

– les tuyaux ne permettent qu'une communication unidirectionnelle ;
– les processus pouvant communiquer au moyen d'un tuyau doivent être issus d'un ancêtre commun qui devra avoir créé le tuyau.

15.1 Manipulation des tuyaux

L'appel système `pipe()`

Un tuyau se crée très simplement au moyen de l'appel système `pipe()` :

```
#include <unistd.h>

int tuyau[2], retour;
retour = pipe(tuyau);
if ( retour == -1 ) {
   /* erreur : le tuyau n'a pas pu etre cree */
}
```

L'argument de `pipe()` est un tableau de deux descripteurs de fichier (un descripteur de fichier est du type `int` en C) similaires à ceux renvoyés par l'appel système `open()` et qui s'utilisent de la même manière. Lorsque le tuyau a été créé, le premier descripteur, `tuyau[0]`, représente le côté lecture du tuyau et le second, `tuyau[1]`, représente le côté écriture.

Un moyen mnémotechnique pour se rappeler quelle valeur représente quel côté est de rapprocher ceci de l'entrée et de la sortie standard. L'entrée standard, dont le numéro du descripteur de fichier est toujours 0, est utilisée pour lire au clavier : $0 \rightarrow$ lecture. La sortie standard, dont le numéro du descripteur de fichier est toujours 1, est utilisée pour écrire à l'écran : $1 \rightarrow$ écriture.

Néanmoins, pour faciliter la lecture des programmes et éviter des erreurs, il est préférable de définir deux constantes dans les programmes qui utilisent les tuyaux :

```
#define LECTURE  0
#define ECRITURE 1
```

Mise en place d'un tuyau

La mise en place d'un tuyau permettant à deux processus de communiquer est relativement simple. Prenons l'exemple d'un processus qui crée un fils auquel il va envoyer des données :

1. Le processus père crée le tuyau au moyen de `pipe()`.
2. Puis il crée un processus fils grâce à `fork()`. Les deux processus partagent donc le tuyau.
3. Puisque le père va écrire dans le tuyau, il n'a pas besoin du côté lecture, donc il le ferme.

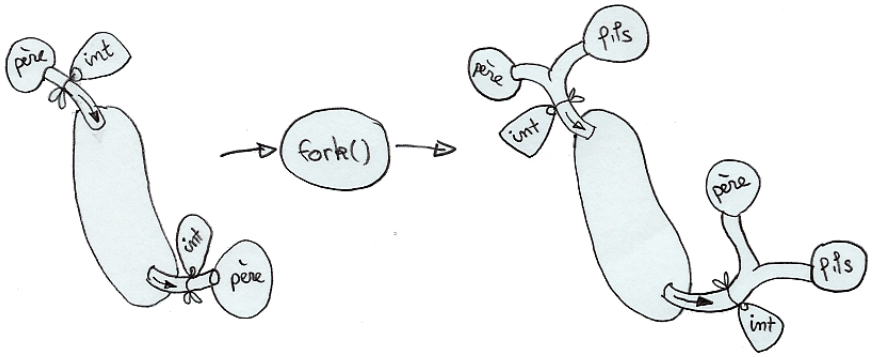


FIGURE 15.1 – *Une fois le tuyau créé, la duplication va offrir deux points d'entrée et deux points de sortie sur le tuyau puisque les descripteurs sont partagés entre le processus père et le processus fils. Nous avons donc un tuyau possédant deux lecteurs et deux écrivains, il faut impérativement faire quelque chose !*

4. De même, le fils ferme le côté écriture.
5. Le processus père peut dès lors envoyer des données au fils.

Le tuyau doit être créé avant l'appel à la fonction `fork()` pour qu'il puisse être partagé entre le processus père et le processus fils (les descripteurs de fichiers ouverts dans le père sont hérités par le fils après l'appel à `fork()`).

Comme indiqué dans l'introduction, un tuyau ayant plusieurs lecteurs peut poser des problèmes, c'est pourquoi le processus père doit fermer le côté lecture après l'appel à `fork()` (il n'en a de toute façon pas besoin). Il en va de même pour un tuyau ayant plusieurs écrivains donc le processus fils doit aussi fermer le côté écriture. Omettre de fermer le côté inutile peut entraîner l'attente infinie d'un des processus si l'autre se termine. Imaginons que le processus fils n'ait pas fermé le côté écriture du tuyau. Si le processus père se termine, le fils va rester bloqué en lecture du tuyau sans recevoir d'erreur puisque son descripteur en écriture est toujours valide. En revanche, s'il avait fermé le côté écriture, il aurait reçu un code d'erreur en essayant de lire le tuyau, ce qui l'aurait informé de la fin du processus père.

Le programme suivant illustre cet exemple, en utilisant les appels système `read()` et `write()` pour la lecture et l'écriture dans le tuyau :

Listing 15.1 – *Utilisation des tuyaux*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2], nb, i;
   char donnees[10];

   if (pipe(tuyau) == -1) { /* creation du pipe */
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) { /* les deux processus partagent le pipe */
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]); /* on ferme le cote ecriture */
      /* on peut alors lire dans le pipe */
      nb = read(tuyau[LECTURE], donnees, sizeof(donnees));
      for (i = 0; i < nb; i++) {
         putchar(donnees[i]);
      }
      putchar('\n');
      close(tuyau[LECTURE]);
      exit(EXIT_SUCCESS);
    default :   /* processus pere, ecrivain */
      close(tuyau[LECTURE]); /* on ferme le cote lecture */
      strncpy(donnees, "bonjour", sizeof(donnees));
      /* on peut ecrire dans le pipe */
      write(tuyau[ECRITURE], donnees, strlen(donnees));
      close(tuyau[ECRITURE]);
      exit(EXIT_SUCCESS);
   }
}
```

Fonctions d'entrées / sorties standard avec les tuyaux

Puisqu'un tuyau s'utilise comme un fichier, il serait agréable de pouvoir utiliser les fonctions d'entrées / sorties standard (`fprintf()`, `fscanf()`...) au lieu de `read()` et `write()`, qui sont beaucoup moins pratiques. Pour cela, il faut transformer les descripteurs de fichiers en pointeurs de type `FILE *`, comme ceux renvoyés par `fopen()`.

Nous pouvons le faire en utilisant la fonction `fdopen()` vue dans le chapitre consacré aux entrées / sorties (cf 11.3). Rappelons qu'elle prend en argument le descripteur de fichier à transformer et le mode d'accès au fichier (`"r"` pour la lecture et `"w"` pour l'écriture) et renvoie le pointeur de type `FILE *` permettant d'utiliser les fonctions d'entrée/sortie standard.

L'exemple suivant illustre l'utilisation de la fonction `fdopen()` :

Listing 15.2 – *Dialogue père / fils*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define LECTURE  0
#define ECRITURE 1

int main (int argc, char *argv[]) {
   int tuyau[2];
   char str[100];
   FILE *mon_tuyau ;

   if ( pipe(tuyau) == -1 ) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]);
      /* ouvre un descripteur de flot FILE * a partir */
      /* du descripteur de fichier UNIX */
      mon_tuyau = fdopen(tuyau[LECTURE], "r");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      /* mon_tuyau est un FILE * accessible en lecture */
      fgets(str, sizeof(str), mon_tuyau);
      printf("Mon pere a ecrit : %s\n", str);
      /* il faut faire fclose(mon_tuyau) ou a la rigueur */
      /* close(tuyau[LECTURE]) mais surtout pas les deux */
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
    default :   /* processus pere, ecrivain */
      close(tuyau[LECTURE]);
      mon_tuyau = fdopen(tuyau[ECRITURE], "w");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      /* mon_tuyau est un FILE * accessible en ecriture */
      fprintf(mon_tuyau, "petit message\n");
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
   }
}
```

Il faut cependant garder à l'esprit que les fonctions d'entrées / sorties standard utilisent une zone de mémoire tampon lors de leurs opérations de lecture ou d'écriture. L'utilisation de cette zone tampon permet d'optimiser, en les regroupant, les accès au disque avec des fichiers classiques mais elle peut se révéler particulièrement gênante avec un tuyau. Dans ce cas, la fonction `fflush()` peut se révéler très utile puisqu'elle permet d'écrire la zone tampon dans le tuyau sans attendre qu'elle soit remplie.

Il est à noter que l'appel à `fflush()` était inutile dans l'exemple précédent en raison de son appel implicite lors de la fermeture du tuyau par `fclose()`.

L'exemple suivant montre l'utilisation de la fonction `fflush()` :

Listing 15.3 – *Forcer l'écriture dans le tuyau*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2];
   char str[100];
   FILE *mon_tuyau;

   if ( pipe(tuyau) == -1 ) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]);
      mon_tuyau = fdopen(tuyau[LECTURE], "r");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      fgets(str, sizeof(str), mon_tuyau);
      printf("[fils] Mon pere a ecrit : %s\n", str);
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
   default :   /* processus pere, ecrivain */
      close(tuyau[LECTURE]);
      mon_tuyau = fdopen(tuyau[ECRITURE], "w");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      fprintf(mon_tuyau, "un message de test\n");
      printf("[pere] Je viens d'ecrire dans le tuyau,\n");
      printf("       mais les donnees sont encore\n");
      printf("       dans la zone de memoire tampon.\n");
      sleep(5);
      fflush(mon_tuyau);
      printf("[pere] Je viens de forcer l'ecriture\n");
      printf("       des donnees de la memoire tampon\n");
      printf("       vers le tuyau.\n");
      printf("       J'attends 5 secondes avant de fermer\n");
      printf("       le tuyau et de me terminer.\n");
      sleep(5);
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
   }
}
```

Redirection des entrée et sorties standard

Une technique couramment employée avec les tuyaux est de relier la sortie standard d'une commande à l'entrée standard d'une autre, comme quand on tape

```
ls | wc -l
```

Le problème, c'est que la commande `ls` est conçue pour afficher à l'écran et pas dans un tuyau. De même pour `wc` qui est conçue pour lire au clavier.

Pour résoudre ce problème, Unix fournit une méthode élégante. Les fonctions `dup()` et `dup2()` permettent de dupliquer le descripteur de fichier passé en argument :

```
#include <unistd.h>
int dup(int descripteur);
int dup2(int descripteur, int copie);
```

Dans le cas de `dup2()`, on passe en argument le descripteur de fichier à dupliquer ainsi que le numéro du descripteur souhaité pour la copie. Le descripteur `copie` est éventuellement fermé avant d'être réalloué.

Pour `dup()` [2], le plus petit descripteur de fichier non encore utilisé permet alors d'accéder au même fichier que `descripteur`.

Mais comment déterminer le numéro de ce plus petit descripteur ? Sachant que l'entrée standard a toujours 0 comme numéro de descripteur et que la sortie standard a toujours 1 comme numéro de descripteur, c'est très simple lorsqu'on veut rediriger l'un de ces descripteurs (ce qui est quasiment toujours le cas). Prenons comme exemple la redirection de l'entrée standard vers le côté lecture du tuyau :

1. On ferme l'entrée standard (descripteur de fichier numéro 0 ou, de manière plus lisible, `STDIN_FILENO`, défini dans `<unistd.h>`). Le plus petit numéro de descripteur non utilisé est alors 0.
2. On appelle `dup()` avec le numéro de descripteur du côté lecture du tuyau comme argument. L'entrée standard est alors connectée au côté lecture du tuyau.
3. On peut alors fermer le descripteur du côté lecture du tuyau qui est maintenant inutile puisqu'on peut y accéder par l'entrée standard.

La façon de faire pour connecter la sortie standard avec le côté écriture du tuyau est exactement la même.

Le programme suivant montre comment lancer **`ls | wc -l`** en utilisant la fonction `dup2()` :

Listing 15.4 – *Duplication des descripteurs*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2];
```

2. Il est recommandé d'utiliser plutôt `dup2()` que `dup()` pour des raisons de simplicité.

```
   if (pipe(tuyau) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
   case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
   case 0 :   /* processus fils, ls , ecrivain */
      close(tuyau[LECTURE]);
      /* dup2 va brancher le cote ecriture du tuyau */
      /* comme sortie standard du processus courant */
      if (dup2(tuyau[ECRITURE], STDOUT_FILENO) == -1) {
         perror("Erreur dans dup2()");
      }
      /* on ferme le descripteur qui reste pour */
      /* << eviter les fuites >> ! */
      close(tuyau[ECRITURE]);
      /* ls en ecrivant sur stdout envoie en fait dans le */
      /* tuyau sans le savoir */
      if (execlp("ls", "ls", NULL) == -1) {
         perror("Erreur dans execlp()");
         exit(EXIT_FAILURE);
      }
   default :   /* processus pere, wc , lecteur */
      close(tuyau[ECRITURE]);
      /* dup2 va brancher le cote lecture du tuyau */
      /* comme entree standard du processus courant */
      if (dup2(tuyau[LECTURE], STDIN_FILENO) == -1) {
         perror("Erreur dans dup2()");
      }
      /* on ferme le descripteur qui reste */
      close(tuyau[LECTURE]);
      /* wc lit l'entree standard, et les donnees */
      /* qu'il recoit proviennent du tuyau */
      if (execlp("wc", "wc", "-l", NULL) == -1) {
         perror("Erreur dans execlp()");
         exit(EXIT_FAILURE);
      }
   }
   exit(EXIT_SUCCESS);
}
```

La séquence utilisant `dup2()` :

```
if ( dup2(tuyau[ECRITURE], STDOUT_FILENO) == -1 ) {
   perror("Erreur dans dup2()");
}
close(tuyau[ECRITURE]);
```

est équivalente à celle-ci en utilisant `dup()` :

```
close(STDOUT_FILENO);
if ( dup(tuyau[ECRITURE]) == -1 ) {
   perror("Erreur dans dup()");
}
close(tuyau[ECRITURE]);
```

Synchronisation de deux processus au moyen d'un tuyau

Un effet de bord intéressant des tuyaux est la possibilité de synchroniser deux processus. En effet, un processus tentant de lire un tuyau dans lequel il n'y a rien est suspendu jusqu'à ce que des données soient disponibles [3]. Donc, si le processus qui écrit dans le tuyau le fait plus lentement que celui qui lit, il est possible de synchroniser le processus lecteur sur le processus écrivain.

L'exemple suivant met en œuvre une utilisation possible de cette synchronisation :

Listing 15.5 – *Synchronisation des lectures / écritures*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2], i;
   char car;

   if (pipe(tuyau) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]);
      /* on lit les caracteres un a un */
      while (read(tuyau[LECTURE], &car, 1) != 0 ) {
         putchar(car);
         /* affichage immediat du caracter lu */
         fflush(stdout);
      }
      close(tuyau[LECTURE]);
      putchar('\n');
      exit(EXIT_SUCCESS);
    default :   /* processus pere, ecrivain */
      close(tuyau[LECTURE]);
      for (i = 0; i < 10; i++) {
         /* on obtient le caractere qui represente le chiffre i */
         /* en prenant le i-eme caractere a partir de '0' */
         car = '0'+i;
         /* on ecrit ce seul caractere */
         write(tuyau[ECRITURE], &car, 1);
         sleep(1); /* et on attend 1 sec */
      }
      close(tuyau[ECRITURE]);
      exit(EXIT_SUCCESS);
   }
}
```

3. À moins qu'il n'ait indiqué au système, grâce à l'option `O_NONBLOCK` de la fonction `fcntl()`, de lui renvoyer une erreur au lieu de le suspendre.

Le signal `SIGPIPE`

Lorsqu'un processus écrit dans un tuyau qui n'a plus de lecteurs (parce que les processus qui lisaient le tuyau sont terminés ou ont fermé le côté lecture du tuyau), ce processus reçoit le signal `SIGPIPE`. Comme le comportement par défaut de ce signal est de terminer le processus, il peut être intéressant de le gérer afin, par exemple, d'avertir l'utilisateur puis de quitter proprement le programme. Les lecteurs curieux pourront approfondir l'utilisation des signaux en consultant les pages de manuel des fonctions suivantes :

- `sigaction()`;
- `sigemptyset()`;
- `sigsuspend()`;
- ...

L'équipe enseignante se tient à leur disposition pour des compléments d'information.

Autres moyens de communication entre processus

Les tuyaux souffrent de deux limitations :

- ils ne permettent qu'une communication unidirectionnelle ;
- les processus pouvant communiquer au moyen d'un tuyau doivent être issus d'un ancêtre commun.

Ainsi, d'autres moyens de communication entre processus ont été développés pour pallier ces inconvénients :

Les tuyaux nommés (*FIFOs*) sont des fichiers spéciaux qui, une fois ouverts, se comportent comme des tuyaux (les données sont envoyées directement sans être stockées sur disque). Comme ce sont des fichiers, ils peuvent permettre à des processus quelconques (pas nécessairement issus d'un même ancêtre) de communiquer.

Les sockets permettent une communication bidirectionnelle entre divers processus, fonctionnant sur la même machine ou sur des machines reliées par un réseau. Ce sujet est abordé dans le TP 16.

Les tuyaux nommés s'utilisent facilement et sont abordés dans le paragraphe suivant.

Les tuyaux nommés

Comme on vient de le voir, l'inconvénient principal des tuyaux est de ne fonctionner qu'avec des processus issus d'un ancêtre commun. Pourtant, les mécanismes de communication mis en jeu dans le noyau sont généraux et, en fait, seul l'héritage des descripteurs de fichiers après un `fork()` impose cette restriction.

Pour s'en affranchir, il faut que les processus désirant communiquer puissent désigner le tuyau qu'ils souhaitent utiliser. Ceci se fait grâce au système de fichiers.

Un tuyau nommé est donc un fichier :

```
menthe22> mkfifo fifo
menthe22> ls -l fifo
prw-r--r--  1 in201    in201    0 Jan 10 17:22 fifo
```

Il s'agit cependant d'un fichier d'un type particulier, comme le montre le `p` dans l'affichage de **`ls`**.

Une fois créé, un tuyau nommé s'utilise très facilement :

```
menthe22> echo coucou > fifo &
[1] 25312
menthe22> cat fifo
[1]  + done       echo coucou > fifo
coucou
```

Si l'on fait abstraction des affichages parasites du *shell*, le tuyau nommé se comporte tout à fait comme on s'y attend. Bien que la façon de l'utiliser puisse se révéler trompeuse, un tuyau nommé transfère bien ses données d'un processus à l'autre en mémoire, sans les stocker sur disque. La seule intervention du système de fichiers consiste à permettre l'accès au tuyau par l'intermédiaire de son nom.

Il faut noter que :

– Un processus tentant d'écrire dans un tuyau nommé ne possédant pas de lecteurs sera suspendu jusqu'à ce qu'un processus ouvre le tuyau nommé en lecture. C'est pourquoi, dans l'exemple, **`echo`** a été lancé en tâche de fond, afin de pouvoir récupérer la main dans le *shell* et d'utiliser **`cat`**. On peut étudier ce comportement en travaillant dans deux fenêtres différentes.
– De même, un processus tentant de lire dans un tuyau nommé ne possédant pas d'écrivains se verra suspendu jusqu'à ce qu'un processus ouvre le tuyau nommé en écriture. On peut étudier ce comportement en reprenant l'exemple mais en lançant d'abord **`cat`** puis **`echo`**.

En particulier, ceci signifie qu'un processus ne peut pas utiliser un tuyau nommé pour stocker des données afin de les mettre à la disposition d'un autre processus une fois le premier processus terminé. On retrouve le même phénomène de synchronisation qu'avec les tuyaux classiques.

En `C`, un tuyau nommé se crée au moyen de la fonction `mkfifo()` :

```
#include <sys/stat.h>

int retour;
retour = mkfifo("fifo", 0644);
if ( retour == -1 ) {
   /* erreur : le tuyau nomme n'a pas pu etre cree */
}
```

Il doit ensuite être ouvert (grâce à `open()` ou à `fopen()`) et s'utilise au moyen des fonctions d'entrées / sorties classiques.

15.2 Exercices

Question 1

Écrire un programme `tuyau1` qui met en place un tuyau et crée un processus fils. Le processus fils lira des lignes de caractères du clavier, les enverra au processus père par l'intermédiaire du tuyau et ce dernier les affichera à l'écran.

Le processus fils se terminera lorsqu'on tapera `Control-D` (qui fera renvoyer `NULL` à `fgets()`), après avoir fermé le tuyau. Le processus fils se terminera de la même façon.

Cet exercice permet de mettre en évidence la synchronisation de deux processus au moyen d'un tuyau.

Fonctions à utiliser :

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int pipe(int tuyau[2])
pid_t fork()
FILE *fdopen(int fichier, char *mode)
char *fgets(char *chaine, size_t taille, FILE *fichier)
```

Question 2

On se propose de réaliser un programme mettant en communication deux commandes par l'intermédiaire d'un tuyau, comme si l'on tapait dans un shell

```
ls | wc -l
```

La sortie standard de la première commande est reliée à l'entrée standard de la seconde.

Écrire un programme `tuyau2` qui met en place un tuyau et crée un processus fils. Le processus père exécutera la commande avec option `sort +4 -n` grâce à la fonction `execlp()` et le processus fils exécutera la commande avec option **`ls -l`**. Pour cela, il faut auparavant rediriger l'entrée standard du processus père vers le côté lecture du tuyau et la sortie standard du processus fils vers le côté écriture du tuyau au moyen de la fonction `dup2()`.

Fonctions à utiliser :

```
#include <sys/types.h>
#include <unistd.h>

int pipe(int tuyau[2])
pid_t fork()
int dup2(int descripteur, int copie)
int execlp(const char *fichier, const char *arg, ...)
```

Question 3

Reprenez le programme **`myshell`** du TP précédent et modifiez-le afin qu'il reconnaisse aussi les commandes de la forme

```
ls -l | sort +4 -n
```

Fonctions à utiliser :

```
#include <sys/types.h>
#include <string.h>
#include <unistd.h>

char *strtok(char *chaine, char *separateurs)
int pipe(int tuyau[2])
pid_t fork()
int dup2(int descripteur, int copie)
int execlp(const char *fichier, const char *arg, ...)
```

La fonction `strtok()` vous sera utile pour analyser la ligne tapée au clavier.

Si vous avez le temps, essayez de modifier le programme pour qu'il reconnaisse un enchaînement quelconque de commandes :

```
cat toto | grep tata | wc -l
```

Dans quel ordre vaut-il mieux lancer les processus fils ?

15.3 Corrigés

Listing 15.6 – *Corrigé du premier exercice*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2];
   FILE *mon_tuyau;
   char ligne[80];

   if (pipe(tuyau) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, ecrivain */
      close(tuyau[LECTURE]); /* on ferme le cote lecture */
      /* ouvre un descripteur de flot FILE * a partir */
      /* du descripteur de fichier UNIX */
      mon_tuyau = fdopen(tuyau[ECRITURE], "w");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      printf("Je suis le fils, tapez des phrases svp\n");
      /* on lit chaque ligne au clavier */
      /* mon_tuyau est un FILE * accessible en ecriture */
      while (fgets(ligne, sizeof(ligne), stdin) != NULL) {
         /* on ecrit la ligne dans le tuyau */
         fprintf(mon_tuyau, "%s", ligne);
         fflush(mon_tuyau);
      }
      /* il faut faire fclose(mon_tuyau) ou a la rigueur */
      /* close(tuyau[LECTURE]) mais surtout pas les deux */
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
    default :   /* processus pere, lecteur */
      close(tuyau[ECRITURE]); /* on ferme le cote ecriture */
      mon_tuyau = fdopen(tuyau[LECTURE], "r");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      /* on lit chaque ligne depuis le tuyau */
      /* mon_tuyau est un FILE * accessible en lecture */
      while (fgets(ligne, sizeof(ligne), mon_tuyau)) {
         /* et on affiche la ligne a l'ecran */
         /* la ligne contient deja un \n a la fin */
         printf(">>> %s", ligne);
      }
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
```

```
   }
}
```

Listing 15.7 – *Corrigé du deuxième exercice*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2];

   if (pipe(tuyau) == -1) {
      perror("Erreur dans pipe().");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork().");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils */
      close(tuyau[LECTURE]);
      /* dup2 va brancher le cote ecriture du tuyau */
      /* comme sortie standard du processus courant */
      dup2(tuyau[ECRITURE],STDOUT_FILENO);
       /* on ferme le descripteur qui reste */
     close(tuyau[ECRITURE]);
      if (execlp("ls", "ls", "-l", NULL) == -1) {
         perror("Erreur dans execlp()");
         exit(EXIT_FAILURE);
      }
    default :   /* processus pere */
      close(tuyau[ECRITURE]);
      /* dup2 va brancher le cote lecture du tuyau */
      /* comme entree standard du processus courant */
      dup2(tuyau[LECTURE], STDIN_FILENO);
      /* on ferme le descripteur qui reste */
      close(tuyau[LECTURE]);
      if (execlp("sort", "sort", "+4", "-n", NULL) == -1) {
         perror("Erreur dans execlp()");
         exit(EXIT_FAILURE);
      }
   }
   exit(EXIT_SUCCESS);
}
```

Listing 15.8 – *Corrigé du troisième exercice*

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define LECTURE  0
```

```
#define ECRITURE 1

int main(int argc, char *argv[]) {
  char ligne[80], *arg1[10], *arg2[10], *tmp;
  int i, tuyau[2];

  printf("myshell> ");
  /* lecture de l'entree standard ligne par ligne */
  while (fgets(ligne, sizeof(ligne), stdin) != NULL) {
    /* fgets lit egalement le caractere de fin de ligne */
    if (strcmp(ligne, "exit\n") == 0) {
      exit(EXIT_SUCCESS);
    }

    /* on decoupe la ligne en s'arretant des que */
    /* l'on trouve un | : cela donne la premiere commande */
    for (tmp = strtok(ligne, " \t\n"), i = 0;
         tmp != NULL && strcmp(tmp , "|") != 0;
         tmp = strtok(NULL, " \t\n"), i++) {
      arg1[i] = tmp;
    }
    arg1[i] = NULL;

    /* on decoupe la suite si necessaire pour obtenir la */
    /* deuxieme commande */
    arg2[0] = NULL;
    if (tmp != NULL && strcmp(tmp, "|") == 0) {
      for (tmp = strtok(NULL, " \t\n"), i = 0;
           tmp != NULL;
           tmp = strtok(NULL, " \t\n"), i++) {
        arg2[i] = tmp;
      }
      arg2[i] = NULL;
    }

    if (arg2[0] != NULL) { /* il y a une deuxieme commande */
      pipe(tuyau);
    }

    switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils */
      if (arg2[0] != NULL) {  /* tuyau */
        close(tuyau[LECTURE]);
        dup2(tuyau[ECRITURE], STDOUT_FILENO);
        close(tuyau[ECRITURE]);
      }
      execvp(arg1[0], arg1);
      /* on n'arrivera jamais ici, sauf en cas d'erreur */
      perror("Erreur dans execvp()");
      exit(EXIT_FAILURE);
    default :   /* processus pere */
      if (arg2[0] != NULL) {
        /* on recree un autre fils pour la deuxieme commande */
        switch (fork()) {
        case -1 :   /* erreur */
          perror("Erreur dans fork()");
          exit(EXIT_FAILURE);
        case 0 :   /* processus fils */
          close(tuyau[ECRITURE]);
          dup2(tuyau[LECTURE], STDIN_FILENO);
          close(tuyau[LECTURE]);
```

```
          execvp(arg2[0], arg2);
          perror("Erreur dans execvp()");
          exit(EXIT_FAILURE);
        default :   /* processus pere */
          /* on ferme les deux cote du tuyau */
          close(tuyau[LECTURE]);
          close(tuyau[ECRITURE]);
          /* et on attend la fin d'un fils */
          wait(NULL);
        }
      }
      /* on attend la fin du fils */
      wait(NULL);
    }
    printf("myshell> ");
  }
  exit(EXIT_SUCCESS);
}
```

Listing 15.9 – *Corrigé du quatrième exercice*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define LECTURE  0
#define ECRITURE 1

/* le type liste chainee d'arguments */
typedef struct argument {
  char *arg;
  struct argument *suivant;
} *t_argument;

/* le type commande, liste doublement chainee */
typedef struct commande {
  int nombre;
  t_argument argument;
  struct commande *suivant;
  struct commande *precedent;
} *t_commande;

int main(int argc ,char *argv[]) {
  char ligne[80];
  t_commande commande;

  commande = (t_commande)malloc(sizeof(struct commande));
  printf("myshell> ");
  while (fgets(ligne, sizeof(ligne), stdin) != NULL) {
    char *tmp;
    t_commande com;
    t_argument arg, precedent;
    int tuyau[2], ecriture;
    pid_t pid, dernier;

    if (strcmp(ligne ,"exit\n") == 0) {
      exit(EXIT_SUCCESS);
    }
```

```
/* premier maillon */
com = commande;
com->argument  = NULL ;
com->nombre    = 0 ;
com->suivant   = NULL ;
com->precedent = NULL ;
precedent = NULL ;

do {
  /* remplissage des arguments d'une commande */
  /* avec allocation des maillons de la liste d'arguments */
  for (tmp = strtok((commande->argument == NULL)?ligne:NULL,
                    " \t\n");
       tmp != NULL && strcmp(tmp, "|") != 0;
       tmp = strtok(NULL, " \t\n")) {
    arg = (t_argument)malloc(sizeof(struct argument));
    if (arg == NULL) {
      perror("Erreur dans malloc()");
      exit(EXIT_FAILURE);
    }
    /* chainage de la liste d'arguments */
    if (precedent != NULL) {
      precedent->suivant = arg;
      precedent = arg;
    } else {
      com->argument = arg;
      precedent = arg;
    }
    arg->arg = tmp;
    arg->suivant = NULL;
    com->nombre++;
  }

  if (tmp != NULL) {  /* vrai uniquement si on a un | */
    /* allocation d'une nouvelle commande */
    com->suivant = (t_commande)malloc(sizeof(struct commande));
    if (com->suivant == NULL) {
      perror("Erreur dans malloc()");
      exit(EXIT_FAILURE);
    }
    /* chainage double du nouveau maillon */
    com->suivant->precedent = com;
    com = com->suivant;
    com->argument = NULL;
    com->nombre = 0;
    com->suivant = NULL;
    precedent = NULL;
  }
} while (tmp != NULL); /* do */

/* on cree les processus en remontant la ligne de commande */
for (; com != NULL; com = com->precedent) {
  int i;
  char **args;

  /* creation du tuyau */
  if (com->precedent != NULL) {
    if (pipe(tuyau) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
    }
  }

  /* creation du processus devant executer la commande */
```

```
  switch (pid = fork()) {
  case -1 :   /* erreur */
    perror("Erreur dans fork()");
    exit(EXIT_FAILURE);
  case 0 :   /* processus fils : commande */
    if (com->precedent != NULL) { /* redirection de stdin */
      close(tuyau[ECRITURE]);
      dup2(tuyau[LECTURE], STDIN_FILENO);
      close(tuyau[LECTURE]);
    }
    if (com->suivant != NULL) {  /* redirection de stdout */
      /* ecriture est le cote ecriture du tuyau */
      /* allant vers la commande plus a droite sur la ligne */
      /* qui a ete lancee au tour de boucle precedent */
      dup2(ecriture, STDOUT_FILENO);
      close(ecriture);
    }
    /* allocation du tableau de pointeurs d'arguments */
    args = (char **)malloc((com->nombre+1)*sizeof(char *));
    for (i = 0, arg = com->argument;
         arg != NULL;
         i++, arg = arg->suivant) {
      args[i] = arg->arg;
    }
    args[i] = NULL;
    /* recouvrement par la commande */
    execvp(args[0], args);
    perror("erreur execvp");
    exit(EXIT_FAILURE);
  default :   /* processus pere : shell */
    if (com->suivant == NULL) {
      dernier = pid;
    }
    /* on ferme les extremites inutiles des tuyaux */
    if (com->suivant != NULL) {
      close(ecriture);
    }
    if (com->precedent != NULL) {
      close(tuyau[LECTURE]);
      /* attention, c'est l'entree du bloc de droite */
      /* il va servir lors du tour de boucle suivant */
      ecriture = tuyau[ECRITURE];
    } else {
      /* on attend la fin du dernier processus directement */
      waitpid(dernier, NULL, 0);
    }
  }
}

/* liberation de la memoire allouee */
/* on reavance jusqu'a la derniere commande */
for (com = commande; com->suivant != NULL; com = com->suivant) ;

while (com != NULL) { /* pour toutes les commandes */
  arg = com->argument;
  while (arg != NULL) { /* on libere les arguments */
    t_argument suivant;
    suivant = arg->suivant;
    free(arg);
    arg = suivant;
  }

  com = com->precedent;
  if (com != NULL) {
```

```
        free(com->suivant); /* on libere la commande */
      }
    }
    printf("myshell> ");
  }
  free(commande);
  exit(EXIT_SUCCESS);
}
```

Enfin voici une possibilité de code n'utilisant ni `strtok()` ni allocation dynamique. La gestion complète de l'entrée standard pour le premier processus de la ligne est laissée aux soins du lecteur.

Listing 15.10 – *Une deuxième version du corrigé du quatrième exercice*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>

#define LECTURE  0
#define ECRITURE 1

/* nombre maximal d'arguments d'une commande */
#define MAXARG 10

/* compte le nombre d'arguments de la commande */
/* la plus a droite de la ligne, s'arrete au | */
char *recherche_pipe(char *str, int *narg) {
    int l,i;
    int n = 1;
    l = strlen(str);
    for (i = l-1; str[i] != '|' && i > 0; i--) {
        if (isspace(str[i-1]) && !isspace(str[i])) n++;
    }
    *narg = n;
    if (str[i] == '|') {
        str[i] = '\0';
        return(str+i+1);
    }
    return(str);
}

/* Decoupage de la ligne en place (sans recopie) */
void decoupe_commande(char *ligne, char *arg[]) {
    int i=0;
    /* on avance jusqu'au premier argument */
    while (isspace(*ligne)) ligne++;
    /* on traite les arguments tant que ce n'est */
    /* pas la fin de la ligne */
    while (*ligne != '\0' && *ligne != '\n') {
        arg[i++]=ligne;
        /* on avance jusqu'au prochain espace */
        while (!isspace(*ligne) && *ligne!='\0')
            ligne++;
        /* on remplace les espaces par '\0' ce qui marque */
        /* la fin du parametre precedent */
        while (isspace(*ligne) && *ligne!='\0')
```

```
            *ligne++='\0';
    }
    arg[i]=NULL;
}


/* execute un processus prenant ecriture en entree */
int execute_commande(char *arg[], int *ecriture) {
    int tuyau[2];
    int pid;

    pipe(tuyau);
    switch (pid = fork()) {
       case -1 :   /* erreur */
         perror("Erreur dans fork()");
         exit(EXIT_FAILURE);
       case 0 :   /* processus fils */
         close(tuyau[ECRITURE]);
         dup2(tuyau[LECTURE], STDIN_FILENO);
         close(tuyau[LECTURE]);
         if (*ecriture >= 0) {
             dup2(*ecriture, STDOUT_FILENO);
             close(*ecriture);
         }
         execvp(arg[0], arg);
         /* on n'arrivera jamais ici, sauf en cas d'erreur */
         perror("Erreur dans execvp()");
         exit(EXIT_FAILURE);
       default :   /* processus pere */
         close(tuyau[LECTURE]);
         close(*ecriture);
         *ecriture = tuyau[ECRITURE];
         return pid;
    }
}


int main(int argc ,char *argv[]) {
   char ligne[80];
   char *arg[MAXARG];
   char *basecommande;
   int narg, letuyau;
   int i, pid;

   printf("myshell> ");
   while (fgets(ligne, sizeof(ligne), stdin) != NULL) {
      if (strcmp(ligne ,"exit\n") == 0) {
         exit(EXIT_SUCCESS);
      }
     /* on supprime le caractere de fin de ligne */
     /* s'il existe (fgets le lit) */
     i=strlen(ligne)-1;
     if (ligne[i] == '\n') ligne[i]='\0';

     basecommande = NULL;
     letuyau = -1;
     while (basecommande != ligne) {
      /* recherche la base de la commande la plus a droite */
      basecommande = recherche_pipe(ligne, &narg);
      if (narg > MAXARG) {
        fprintf(stderr, "Trop de parametres\n");
        exit(EXIT_FAILURE);
      }
      /* decoupe cette commande en tableau arg[] */
```

```
      decoupe_commande(basecommande, arg);
      /* lance la commande, en notant le pid de la derniere */
      if (letuyau == -1) {
           pid = execute_commande(arg, &letuyau);
      } else {
           execute_commande(arg, &letuyau);
      }
     }
     /* le premier processus de la ligne n'a pas de stdin */
     close(letuyau);
     waitpid(pid, NULL, 0);
     printf("myshell> ");
    }
    exit(EXIT_SUCCESS);
}
```

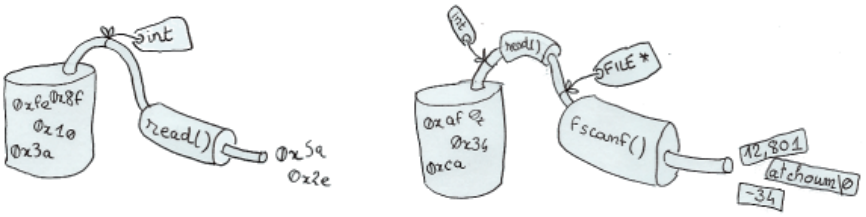


FIGURE 15.2 – *Le descripteur de fichier entier ne permet qu'une manipulation octet par octet. Le descripteur de fichier de type* `FILE` *permet quant à lui une manipulation plus structurée.*

15.4 Corrections détaillées

Nous allons dans un premier temps revenir très rapidement sur les fichiers afin que les différentes fonctions que nous utiliserons pour manipuler les tuyaux soient bien présentes dans les esprits des lecteurs.

Les fichiers sous Unix

Les fichiers représentent en fait des ensembles de données manipulées par l'utilisateur. Un fichier n'est donc rien d'autre qu'un réceptacle à données. Le système nous fournit différents moyens de manipuler ces réceptacles, car avant de pouvoir obtenir les données il faut avant tout pouvoir se saisir du bon réceptacle. Entre l'utilisateur et le fichier, l'interface la plus simple est le descripteur de fichier, c'est-à-dire un entier : il s'agit du numéro de la ligne de la table des fichiers ouverts faisant référence au fichier souhaité. Ce descripteur est obtenu très simplement par l'appel système `open()`.

Un descripteur de fichier (point d'entrée dans la table des fichiers ouverts) ne permet que des manipulations simples sur les fichiers : la lecture et l'écriture se feront par entité atomique, donc des *octets*, via les deux fonctions système `read()` et `write()`.

Afin de pouvoir manipuler des entités plus structurées (des entiers, des mots, des `double`, etc.) il nous faut utiliser une autre interface que le descripteur entier comme le montre la figure 15.2.

L'interface mise en place par le descripteur complexe `FILE` permet, outre une manipulation plus simple des entités à lire ou à écrire, une temporisation des accès au fichier. Ainsi au lieu d'écrire octet par octet dans le fichier, le système placera dans un premier temps les différentes données dans une zone tampon. Quand la nécessité s'en fera sentir, la zone tampon sera vidée. Ceci vous explique pourquoi certaines écritures ne se font jamais lorsqu'un programme commet une erreur et s'arrête, même si l'erreur est intervenu après l'appel à `fprintf()`.

Nous pouvons aisément passer du descripteur entier au descripteur complexe en utilisant la fonction `fdopen()`.

```
int descrip_trivial;
FILE *descript_complexe;

descript_trivial = open("Mon_beau_sapin.tex",O_RDONLY,NULL);
descript_complexe = fdopen(descript_trivial,"r");

fscanf(descript_complexe,"%d %f %d",
       &entier,&reel,&autre_entier);
```

Lorsque nous utilisons la fonction `fdopen()`, nous n'ouvrons pas le fichier une seconde fois, nous nous contentons simplement d'obtenir une nouvelle façon de manipuler les données qui y sont stockées. Donc, lorsque nous fermons le fichier, nous devons utiliser **soit** la fonction système `close()`, **soit** la fonction de la bibliothèque standard `fclose()`, mais certainement pas les deux !

Quel lien avec les tuyaux ? Et bien les tuyaux sont des fichiers un peu particuliers donc mieux vaut savoir comment manipuler les fichiers ! Il est important de garder à l'esprit qu'un descripteur de fichier est obtenu en demandant une ouverture de fichier avec un mode : lecture ou écriture, il en sera de même avec les tuyaux.

Les tuyaux

Un tuyau est une zone d'échange de données entre deux ou plusieurs processus. Cette zone est cependant assez restrictive quant aux échanges autorisés, ceux-ci sont unidirectionnels et destructifs :

- la communication ne peut se fait que d'un groupe de processus (les écrivains) vers d'autres (les lecteurs). Une fois qu'un processus a choisit sa nature (écrivain ou lecteur), impossible de revenir en arrière.
- toute donnée lue par un lecteur n'est plus disponible pour les autres lecteurs.

Comme le nom le suggère assez bien, un tuyau doit posséder deux descripteurs de fichier, l'un permettant d'écrire, et l'autre permettant de lire. La figure 15.3 décrit de manière schématique un tuyau et les deux descripteurs attachés.

On subodore donc que la fonction permettant de créer un tel objet doit soit renvoyer un tableau de deux entiers, soit prendre en paramètre un tableau de deux entiers. Mais comme nous savons que les fonctions systèmes retournent en général un code d'erreur, la bonne solution doit être la seconde :

```
int pipe(int filedes[2]);
```

Il existe un moyen simple pour se souvenir *quid* des deux descripteurs est celui utilisé en lecture (du coup l'autre sera utilisé en écriture). Le périphérique servant à lire n'est rien d'autre que le clavier, soit de manière un peu plus générale `stdin`, dont le numéro est `0`. Le descripteur utilisé en lecture est donc `fildes[0]`.

Nous allons commencer par utiliser un tuyau au sein d'un seul processus, ce qui ne servira pas à grand chose sinon à fixer les esprits sur les différents moyens mis à notre disposition pour lire et écrire dans un tuyau.

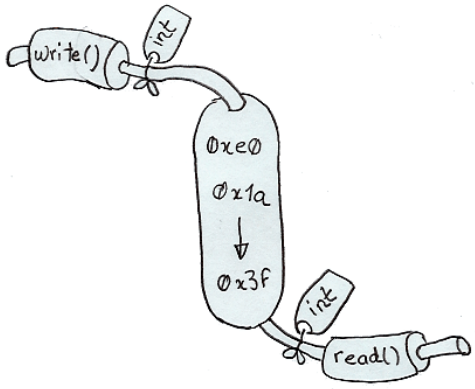


FIGURE 15.3 – *Un tuyau permet de ranger (dans l'ordre d'arrivée selon les écritures) des données en mémoire pour les récupérer. Il est donc fourni avec deux descripteurs de fichier, l'un servant à écrire et l'autre à lire.*

Listing 15.11 – *Utilisation basique*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  FILE *fpin,*fpout;
  int tuyau[2];
  int retour,k;
  char phrase_in[256];
  char phrase_out[256];
  double pi;

  retour = pipe(tuyau);
  if (retour == -1) {
    perror("Impossible de creer le tuyau");
    exit(EXIT_FAILURE);
  }
  for(k=0;k<10;k++) phrase_in[k] = 'a'+k;

  write(tuyau[1],phrase_in,10);
  printf("J'ecris dans mon tuyau\n");
  sleep(1);
  read(tuyau[0],phrase_out,10);
  printf("J'ai lu dans mon tuyau\n");

  for(k=0;k<10;k++) printf("%c ",phrase_out[k]);
  printf("\n");

  fpout = fdopen(tuyau[0],"r");
  fpin = fdopen(tuyau[1],"w");
```

```
  fprintf(fpin,"Bonjour voici une phrase\n");fflush(fpin);
  sleep(1);
  fgets(phrase_out,255,fpout);
  printf("J'ai lu: \"%s\" dans le tuyau\n",phrase_out);

  fprintf(fpin,"3.141592\n");fflush(fpin);
  sleep(1);
  fscanf(fpout,"%lf",&pi);
  printf("J'ai lu pi dans le tuyau: %f\n",pi);
  close(tuyau[0]);
  close(tuyau[1]);
  exit(EXIT_SUCCESS);
}
```

La création du tuyau est très simple, l'appel système `pipe()` nous renvoie un code d'erreur si le tuyau n'a pas pu être créé. Si tout se passe bien nous obtenons nos deux descripteurs entiers dans le tableau `tuyau[]` passé en paramètre.

Dans un premier temps nous allons simplement écrire 10 octets dans le tuyau par l'intermédiaire du descripteur d'écriture. Nous utilisons la fonction `write()`, intermédiaire d'écriture incontournable pour les descripteurs de fichier entiers. Puis nous lisons, mais sur le côté lecture du tuyau, à l'aide de la fonction `read()`. On lit exactement le bon nombre de caractères (ou moins à la rigueur, mais surtout pas plus).

Afin de faire appel à des fonctions manipulant des données un peu plus évoluées que les octets, nous demandons une représentation plus complexe des deux descripteurs, et pour ce faire nous utilisons la fonction `fdopen()` qui nous renvoie un descripteur complexe (`FILE *`) à partir d'un descripteur entier.

Nous pouvons alors écrire et lire des données structurées, comme le montre la suite du programme, avec toutefois un bémol important, l'utilisation de la fonction `fflush()`, car nous le savons les descripteurs complexes utilisent des tampons et ne placent les données dans leur fichier destination immédiatement.

Les tuyaux nommés

L'utilisation des tuyaux doit permettre à différents processus de communiquer. Avant d'utiliser les fonctions vues dans le précédent chapitre (duplication et recouvrement), nous allons nous intéresser à une forme particulière de tuyaux, les tuyaux nommés. Puisque nous avons bien compris que les tuyaux n'étaient ni plus ni moins que des sortes de fichiers, il doit être possible de créer véritablement des fichiers qui se comportent comme des tuyaux. Pour cela, nous avons à notre disposition la fonction `mkfifo()`. Elle permet de créer un fichier qui se comporte comme un tuyau, et puisque ce fichier possède un nom, nous pourrons écrire et lire dedans en l'ouvrant au moyen de son nom ! Les trois petits programmes qui suivent vous permettent de voir l'utilisation et surtout le séquencement des opérations d'écriture et de lecture.

Tout d'abord le programme de création, d'écriture et puis de destruction du tuyau nommé :

Listing 15.12 – *Création simple, écriture et fermeture*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
  int retour;
  FILE *fp;
  int k;

  retour = mkfifo("ma_fifo",0644);
  if (retour == -1) {
    perror("Impossible de creer le tuyaux nomme");
    exit(EXIT_FAILURE);
  }
  fp = fopen("ma_fifo","w");
  for(k=0;k<6;k++) {
    fprintf(fp,"Ecriture numero %d dans la fifo\n",k);
    fflush(fp);
    sleep(2);
  }
  fclose(fp);
  unlink("ma_fifo");
  exit(EXIT_SUCCESS);
}
```

Ensuite le programme de lecture qui prend un argument afin de pouvoir être identifié car nous allons le lancer dans 2 fenêtres différentes :

Listing 15.13 – *Le programme de lecture complémentaire au précédent*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  FILE *fp;
  char phrase[256];

  if (argc < 2) {
    fprintf(stderr,"usage: %s <un nom>\n",argv[0]);
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen("ma_fifo","r")) == NULL) {
    perror("P2: Impossible d'ouvrir ma_fifo");
    exit(EXIT_FAILURE);
  }
  while(fgets(phrase,255,fp)!=NULL) {
    printf("%s: \"%s\"\n",argv[1],phrase);
  }
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```

Nous commençons par lancer le programme de création / écriture. Comme l'écriture dans un tuyau sans lecteur est bloquante, cela nous laisse le temps de lancer les deux

programmes lecteurs. L'affichage donne ceci (chaque fenêtre est identifiée par un numéro différent dans l'invite de l'interprète de commandes) :

```
menthe31>p1
menthe22>p2 premier
premier: "Ecriture numero 0 dans la fifo"
premier: "Ecriture numero 2 dans la fifo"
premier: "Ecriture numero 4 dans la fifo"
menthe22>
menthe12>p2 deuxieme
deuxieme: "Ecriture numero 1 dans la fifo"
deuxieme: "Ecriture numero 3 dans la fifo"
deuxieme: "Ecriture numero 5 dans la fifo"
menthe22>
```

Nous voyons tout d'abord que le programme **p1** reste bloqué tant qu'il reste des lecteurs. Ensuite nous remarquons que le programme **p2** lancé dans deux fenêtres différentes, lit alternativement les différentes écritures ce qui reflète bien la notion de lecture destructive.

Il est important de garder à l'esprit que les données échangées ne sont jamais écrites sur le disque. Le fichier `ma_fifo` ne sert qu'à donner un nom, mais tous les échanges restent confinés à la mémoire.

Nous allons maintenant passer aux tuyaux permettant à différents processus issus d'un même père de partager des données.

Tuyaux et processus

Nous savons que l'appel à `fork()` permet de conserver les descripteurs de fichier ouverts lors de la duplication. Nous allons tirer partie de cela pour créer un processus fils qui va pouvoir dialoguer avec son père.

Il est fondamental de bien comprendre que l'appel à `fork()` vient dupliquer les différents descripteurs et qu'une fois la duplication réalisée, le tuyau possède deux points d'entrée et deux points de sortie comme le montre la figure 15.1.

Nous allons mettre en œuvre cela en répondant à la question 15.2. Notre programme va s'articuler autour d'une fonction qui lira des caractères sur le clavier (l'entrée standard), une fonction qui les écrira dans le tuyau et une fonction qui lira dans le tuyau.

Dans un premier temps nous créons le tuyau, car il est impératif qu'il existe avant la duplication pour être partagé. À la suite de la duplication, nous scinderons le programme en deux, la partie « fils » réalisera l'envoi des données dans le tuyau, la partie « père » les lira et les affichera à l'écran.

Listing 15.14 – *Utilisation de la duplication*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define LECTURE 0
```

```
#define ECRITURE 1


int lire_clavier(char *ligne, int length)
{
  int ret;
  ret = (int)fgets(ligne,length,stdin);
  return ret;
}

void proc_fils(int tuyau[])
{
  char ligne[256];
  FILE *fp;

  /* Fermeture du tuyau en lecture */
  close(tuyau[LECTURE]);

  if ((fp = fdopen(tuyau[ECRITURE],"w"))==NULL) {
    perror("Impossible d'obtenir un descripteur decent");
    close(tuyau[ECRITURE]);
    exit(EXIT_FAILURE);
  }
  while(lire_clavier(ligne,256)) {
    fprintf(fp,"%s",ligne);fflush(fp);
  }
  close(tuyau[ECRITURE]);
  fprintf(stdout,"Fin d'ecriture tuyau\n");
  exit(EXIT_SUCCESS);
}

void proc_pere(int tuyau[])
{
  char ligne[256];
  FILE *fp;

  /* Fermeture du tuyau en ecriture */
  close(tuyau[ECRITURE]);

  if ((fp = fdopen(tuyau[LECTURE],"r"))==NULL) {
    perror("Impossible d'obtenir un descripteur decent");
    close(tuyau[LECTURE]);
    exit(EXIT_FAILURE);
  }
  while(fgets(ligne,256,fp)!=NULL) {
    ligne[strlen(ligne)-1] = '\0';
    fprintf(stdout,"Pere:%s\n",ligne);
  }
  close(tuyau[LECTURE]);
  fprintf(stdout,"Fin de lecture tuyau\n");
  exit(EXIT_SUCCESS);

}

int main(int argc, char **argv)
{
  int tuyau[2];

  if (pipe(tuyau) == -1) {
    perror("Impossible de tuyauter");
    exit(EXIT_FAILURE);
  }

  switch(fork()) {
```

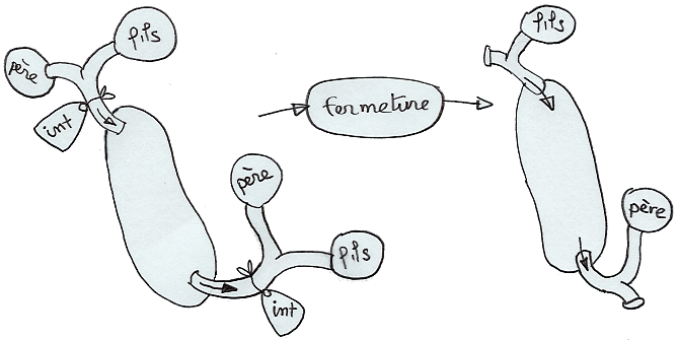


FIGURE 15.4 – *Une fois le tuyau créé et la duplication effective, on ferme l'un des lecteurs et l'un des écrivains. Le tuyau ne possède plus qu'une entrée (le fils) et une sortie (le père).*

```
  case -1:
    perror("Impossible de forker");
    exit(EXIT_FAILURE);
  case 0: /* processus fils */
    proc_fils(tuyau);
  default: /* processus pere */
    proc_pere(tuyau);
  }
  exit(EXIT_FAILURE);
}
```

Il est essentiel de bien comprendre le fonctionnement décrit par la figure 15.1. À l'issue de la fonction `fork()`, il existe un descripteur en écriture pour le fils, un autre pour le père, ainsi qu'un descripteur en lecture pour le fils et un autre pour le père. Notre tuyau possède donc deux points d'entrée et deux points de sortie. Il est impératif de mettre fin à cette situation un peu dangereuse. En effet un tuyau ouvert en écriture (avec un ou plusieurs points d'entrée) ne peut pas signifier à son (ou ses lecteurs) qu'il n'est nécessaire d'attendre des données.

Le processus fils, qui est l'écrivain dans le tuyau, va tout d'abord fermer **son** côté lecteur. De la même façon, le processus père, qui est le lecteur du tuyau, va fermer **son** côté écrivain. Ainsi nous obtenons, comme le montre le diagramme chronologique de la figure 15.4, le tuyau ne possède plus à un instant donné qu'un écrivain, le processus fils, et un lecteur, le processus père.

Tuyaux et recouvrement

Une des utilisations possibles est de pouvoir faire communiquer des programmes entre eux, nous l'avons vu avec les tuyaux nommés. Peut-on aboutir à la même chose

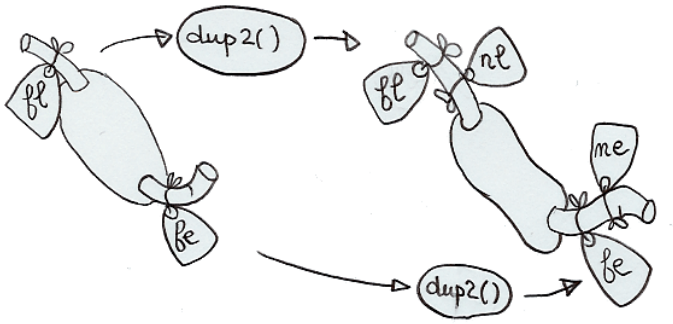


FIGURE 15.5 – *La duplication de chaque descripteur de fichier permet de faire aboutir un nouveau descripteur (*`ne` *ou* `nl`*) sur l'entrée ou la sortie d'un tuyau. Si l'on choisit par exemple de dupliquer* `stdin` *sur* `tuyau[0]` *alors un processus qui partage le tuyau et qui lit dans* `stdin` *lira en fait dans le tuyau.*

sans ces tuyaux nommés ? Avec des programmes que nous concevons nous-mêmes et que nous pouvons intégrer dans un développement, la chose est strictement identique à ce que nous venons de faire dans le programme précédent. Par contre, comment procéder à l'exécution de cette commande assez simple : **`ls -l | wc -l`** ? Il faut « brancher » la sortie standard de la commande **`ls`** sur l'entrée standard de la commande **`wc`**. Pour ce faire nous allons utiliser une nouvelle fonction : `dup2()`.

Cette fonction crée une copie d'un descripteur de fichier, et en paraphrasant le manuel, après un appel réussi à cette fonction, le descripteur et sa copie peuvent être utilisés de manière interchangeable. La solution est donc là : on crée un tuyau et on duplique son entrée (le côté écrivain) sur la sortie standard de la première commande (**`ls -l`**). Puis on duplique sa sortie (le côté lecteur) sur l'entrée standard de la deuxième commande (**`wc -l`**). Ainsi, la première commande écrit dans le tuyau et la deuxième commande lit dans le tuyau. La figure 15.5 décrit ce cheminement.

Nous allons donc pouvoir programmer à l'aide de `fork()` et `dup2()` l'exercice de recouvrement. Nous avons deux fonctions et nous allons utiliser `fork()` pour obtenir deux processus. Lequel du père ou du fils doit se charger de la première fonction ? Il faudrait que le père se termine après le fils, donc le père doit attendre des informations du fils, ce qui implique nécessairement que le père soit en charge de la dernière fonction et le fils de la première. La figure 15.6 décrit la chronologie des événements.

Listing 15.15 – *Tuyau et recouvrement*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
```

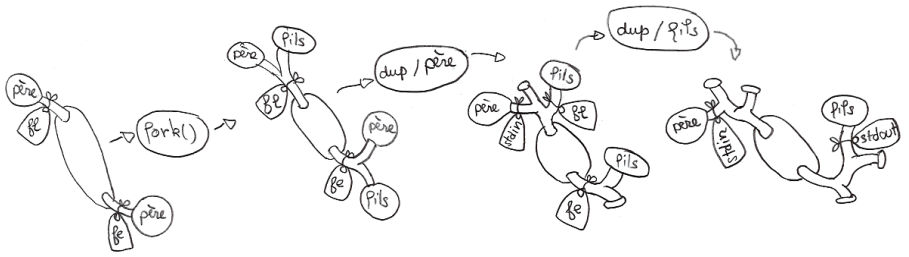


FIGURE 15.6 – *Après la duplication de processus, nous avons quatre descripteurs disponibles sur le tuyau, deux pour le père et deux pour le fils. Dans le processus père, l'utilisation de la duplication de descripteur et deux fermetures adéquates permettent de couper l'entrée en lecture du père et de brancher l'entrée standard sur le tuyau en lecture. Le père ferme aussi son entrée en écriture. Du côté du fils, la duplication de descripteur permet de brancher la sortie standard sur le côté écriture du tuyau et de fermer l'entrée écriture du fils. Le fils n'étant pas un lecteur, il ferme l'entrée lecture du tuyau. On obtient donc bien un tuyau dans lequel il n'y a qu'une entrée en lecture (l'entrée standard qui servira au père lors de son recouvrement) et une entrée en écriture (la sortie standard qui servira au fils lors de son recouvrement).*

```
#include <sys/types.h>
#include <sys/wait.h>

#define LECTURE 0
#define ECRITURE 1


int main(int argc, char **argv)
{
  int tuyau[2];

  if (pipe(tuyau) == -1) {
    perror("Impossible de tuyauter");
    exit(EXIT_FAILURE);
  }

  switch(fork()) {
  case -1:
    perror("Impossible de forker");
    exit(EXIT_FAILURE);
  case 0: /* processus fils */
      /* Il est juste ecrivain, donc fermeture du cote */
      /* lecteur.*/
    close(tuyau[LECTURE]);
    /* La premiere commande ls -l ecrit ses resultats */
    /* dans le tuyau et non sur la sortie standard */
    /* donc on duplique */
    if (dup2(tuyau[ECRITURE],STDOUT_FILENO)==-1) {
      perror("Impossible du dupliquer un descripteur");
      close(tuyau[ECRITURE]);
      exit(EXIT_FAILURE);
```

```
    }
    /* Comme le processus sera recouvert, le */
    /* descripteur tuyau[ECRITURE] est inutile */
    /* donc dangereux, on le ferme!*/
    close(tuyau[ECRITURE]);
    /* On recouvre le processus avec "ls -l" */
    execlp("ls","ls","-l",NULL);
    /* si on arrive ici c'est que execlp va mal!*/
    exit(EXIT_FAILURE);
  default: /* processus pere */
    /* Il est juste lecteur, donc fermeture du cote */
    /* ecrivain.*/
    close(tuyau[ECRITURE]);
    /* la derniere commande wc -l doit lire ses */
    /* arguments dans le tuyau et non */
    /* sur l'entree standard donc on duplique */
    if (dup2(tuyau[LECTURE],STDIN_FILENO)==-1) {
      perror("Impossible du dupliquer un descripteur");
      close(tuyau[LECTURE]);
      exit(EXIT_FAILURE);
    }
    /* Comme le processus sera recouvert, le */
    /* descripteur tuyau[LECTURE] est inutile */
    /* donc dangereux, on le ferme!*/
    close(tuyau[LECTURE]);
    /* On recouvre le processus avec "wc -l" */
    execlp("wc","wc","-l",NULL);
    /* si on arrive ici c'est que execlp va mal!*/
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

L'interprète de commandes

Le programme qui suit est relativement commenté. Il est surtout important de comprendre que dans ce programme il y a plusieurs tuyaux et que la sortie (côté lecture) de l'un est branchée sur le côté écriture d'un autre afin d'élaborer la suite des commandes.

Listing 15.16 – *Ecriture d'un shell*

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

#define LECTURE 0
#define ECRITURE 1

#define MAXCHAR_LIGNE 80
#define MAXARG 10

/* Structure de liste chainee pour les arguments */
/* d'une commande */
```

```
typedef struct argument {
  char *arg;              /* l'argument (non recopie)     */
  struct argument *suivant; /* l'adresse du maillon suivant */
} Argument;

/* Structure de liste chainee pour les commandes */
typedef struct commande {
  int numero;            /* le numero de la commande (4fun)     */
  int argc;              /* le nombre d'arguments de la commande*/
  Argument *argv;         /* la liste chainee des arguments     */
  struct commande *suivante; /* l'adresse du maillon suivant    */
} Commande;

/* Cette fonction ajoute un argument (char *) a la liste
 * chainee "l" des arguments.
 * Si la liste est vide (l==NULL) la fonction retourne
 * l'adresse du nouveau maillon, sinon la fonction rajoute
 * l'argument a la fin de la liste en se rappelant elle meme
 * La liste construite a partir de :
 * l=NULL;
 * l = add_argument(l,arg1);
 * l = add_argument(l,arg2);
 * l = add_argument(l,arg3);
 *
 * sera donc :  l(arg1) --> l(arg2) --> l(arg3) --> NULL
 */
Argument *add_argument(Argument *l, char *argvalue)
{
  Argument *newarg;

  if (l==NULL) {
    if ((newarg = malloc(sizeof(Argument))) == NULL) {
      perror("Erreur d'allocation d'argument.");
      exit(EXIT_FAILURE);
    }
    newarg->arg = argvalue;
    newarg->suivant = NULL;
    return newarg;
  }
  if (l->suivant == NULL) {
    if ((newarg = malloc(sizeof(Argument))) == NULL) {
      perror("Erreur d'allocation d'argument.");
      exit(EXIT_FAILURE);
    }
    newarg->arg = argvalue;
    newarg->suivant = NULL;
    l->suivant = newarg;
    return l;
  }
  add_argument(l->suivant,argvalue);
  return l;
}

/* Cette fonction ajoute une commande d'arguments "arg"
 * a la liste chainee de commandes "l".
 * La nouvelle commande est rajoutee au debut de
 * la liste (a l'inverse de la commande add_argument)
 */
Commande *add_commande(Commande *l, Argument *arg, int argc)
{
  Commande *newcmd;
  if (argc <= 0) return l;

  if ((newcmd = malloc(sizeof(Commande))) == NULL) {
```

```
    perror("Erreur d'allocation de commande.");
    exit(EXIT_FAILURE);
  }
  newcmd->argv = arg;
  newcmd->argc = argc;
  newcmd->suivante = l;
  newcmd->numero = (l!=NULL?l->numero+1:0);
  return newcmd;
}

/* Liberation des maillons (pas du champ (char *)arg
 * car ce dernier n'est pas recopie) de la liste
 * d'arguments "l"
 */
Argument *free_argliste(Argument *l)
{
  Argument *toBfree;
  while(l != NULL) {
    toBfree = l;
    l=l->suivant;
    free(toBfree);
  }
  return NULL;
}

/* Liberation des maillons (champ Argument *argv compris)
 * de la liste des commandes "l"
 */
Commande *free_cmdliste(Commande *l)
{
  Commande *toBfree;
  while(l != NULL) {
    toBfree = l;
    l=l->suivante;
    free_argliste(toBfree->argv);
    free(toBfree);
  }
  return NULL;
}

void show_argliste(Argument *l)
{
  Argument *cur;
  int i=0;

  cur = l;
  while(cur != NULL) {
    fprintf(stdout,"\tArgument %3d: \"%s\" (%10p)\n",
            i++,cur->arg,cur->arg);
    cur = cur->suivant;
  }
}

void show_commande(Commande *cur)
{
  fprintf(stdout,"Commande %3d avec %2d argument(s): \n",
          cur->numero,cur->argc);
  show_argliste(cur->argv);
}

void show_allcommande(Commande *c)
{
  Commande *cur;
```

```
  cur = c;
  while(cur != NULL) {
    show_commande(cur);
    cur = cur->suivante;
  }
}

/* Fonction permettant de construire la liste chainee
 * d'arguments a partir d'une chaine de caracteres.
 * Chaque argument est separe du suivant par " " ou par
 * un caractere de tabulation ou par un retour chariot (ce
 * qui ne devrait pas arriver
 */
Argument *parse_argument(char *ligne, int *argc)
{
  char *curarg,*ptrtmp;
  Argument *listearg=NULL;
  *argc=0;
  while((curarg =
     strtok_r(listearg==NULL?ligne:NULL," \t\n",&ptrtmp)) != NULL) {
    listearg = add_argument(listearg,curarg);
    *argc += 1;
  }
  return listearg;
}

/* Fonction d'analyse d'une ligne de commandes. Chaque commande
 * est separee de la suivante par le caractere '|' (tuyau). Une
 * fois reperee la fin de la commande, les caracteres compris entre
 * le debut (start) et la fin (on remplace '|' par '\0') sont
 * envoyes a parse_argument puis la nouvelle commande est placee
 * dans la liste chainee des commandes.
 */
Commande *parse_ligne(char *ligne)
{
  Commande *com=NULL;
  Argument *args=NULL;
  int argc;
  char *start;
  int i,len;

 /* on supprime le dernier caractere: \n */
  ligne[strlen(ligne)-1] = '\0';
  len = strlen(ligne);
  for (i = 0,start=ligne;i<len;i++) {
    if (ligne[i] == '|') {
      ligne[i] = '\0';
      args = parse_argument(start,&argc);
      com = add_commande(com,args,argc);
      start = ligne+i+1;;
    }
  }
  args = parse_argument(start,&argc);
  com = add_commande(com,args,argc);
  return com;
}

/* Cette fonction prend une liste chainee d'arguments et construit
 * un tableau d'adresses telles que chacune pointe sur un argument
 * de la liste chainee. Ceci permet d'utiliser execvp() pour recouvrir
 * le processusen cours par la commande souhaitee avec ses arguments.
 */
char **construire_argv(Commande *curcom)
{
```

```
  Argument *curarg;
  char **argv;
  int i;

  argv = malloc((curcom->argc+1)*sizeof(char *));
  for(i=0,curarg=curcom->argv;i<curcom->argc;
      i++,curarg=curarg->suivant) {
    argv[i] = curarg->arg;
  }
  argv[i] = NULL;
  return argv;
}

/*
 * Cette fonction met en place l'execution d'une commande. Elle
 * commence par creer un tuyau puis procede a un appel a fork().
 *
 * Le processus fils ferme le cote ecriture, branche son entree
 * standard sur le cote lecture puis ferme le cote lecture.
 * Si le fils ne correspond pas au premier appel (donc a la
 * derniere commande de la ligne analysee), il branche sa sortie
 * standard sur le cote ecriture du tuyau precedent puis il ferme
 * le cote ecriture du tuyau precedent. Il se recouvre ensuite
 * avec la commande desiree.
 *
 * Le pere ferme le cote lecture du tuyau en cours et si il ne
 * s'agit pas du premier appel (donc de la derniere commande de
 * la ligne analysee) il ferme le cote ecriture du tuyau precedent.
 * Il met a jour la variable prec_ecriture en disant que pour
 * les appels a venir, le tuyau precedent en ecriture est le tuyau
 * courant en ecriture. Il retourne enfin le pid de son fils.
 */
int execute_commande(char **argv, int *prec_ecriture)
{
  int tuyau[2];
  int pid;

  if (pipe(tuyau) == -1) {
    perror("Impossible de creer le tuyau");
    exit(EXIT_FAILURE);
  }
  pid = fork();
  switch (pid) {
  case -1:
    perror("Impossible de creer un processus");
    exit(EXIT_FAILURE);
  case 0:
    /* pas besoin d'etre ecrivain dans ce tuyau */
    close(tuyau[ECRITURE]);
    /* branchement de stdin sur la lecture du tuyau */
    dup2(tuyau[LECTURE],STDIN_FILENO);
    close(tuyau[LECTURE]);

    /* branchement de stdout sur l'ecriture du tuyau precedent */
    /* s'il y a un tuyau precedent */
    if (*prec_ecriture >= 0) {
      dup2(*prec_ecriture,STDOUT_FILENO);
      close(*prec_ecriture);
    }
    execvp(argv[0],argv);
    perror("Impossible de proceder au recouvrement");
    fprintf(stderr,"commande: \"%s\"\n",argv[0]);
    exit(EXIT_FAILURE);
  default :
```

```
    /* pas besoin de lire sur le tuyau car on est le pere,
     * et on se contente de regarder passer les anges
     */
    close(tuyau[LECTURE]);
    if (*prec_ecriture >= 0) {
      if(close(*prec_ecriture)) {
        fprintf(stderr,"Pere:Impossible de fermer");
        fprintf(stderr,"le tuyau precedent en ecriture\n");
      }
    }
    *prec_ecriture = tuyau[ECRITURE];
    return pid;
  }
  exit(EXIT_FAILURE);
}

int main(int margc, char **margv)
{
  char ligne[MAXCHAR_LIGNE];
  Commande *com=NULL;
  Commande *curcom;
  char **argv;
  int prec_ecriture;
  int pid=-1;

  /* On affiche une invitation! */
  printf("myshell>");

  /* Tant que l'on peut lire des trucs, on lit */
  while (fgets(ligne, sizeof(ligne),stdin) != NULL) {

    /* Si l'utilisateur veut sortir de ce magnifique interprete
     * on sort en tuant le processus en court
     */
    if (!strncmp(ligne,"exit",4)) exit(EXIT_SUCCESS);

    /* On analyse la ligne de commandes afin de construire
     * les differentes listes chainees.
     */
    com = parse_ligne(ligne);

    /* Les commandes sont maintenant rangees dans une liste   */
    /* la derniere commande de la ligne est en tete de liste  */
    /* puis vient la suivante, etc. jusqu'a la premiere de la */
    /* ligne qui se trouve en fin de liste */

    /* Le tuyau precedent n'existe pas encore, donc -1 */
    prec_ecriture=-1;
    curcom = com;

    /* On analyse chaque commande de la ligne en commencant par
     * celle qui est la plus a droite
     */
    while (curcom != NULL) {

      /* on construit le fameux tableau char *argv[] */
      argv = construire_argv(curcom);

      /* S'il s'agit de la premiere commande (celle de droite!)
       * on recupere son pid() histoire de pouvoir attendre qu'elle
       * soit finie avant de rendre la main a l'utilisateur
       */
      if (prec_ecriture < 0)
        pid = execute_commande(argv,&prec_ecriture);
```

```
      else
        execute_commande(argv,&prec_ecriture);
      free(argv);
      argv=NULL;
      curcom = curcom->suivante;
    }
    /* Fin d'analyse de la ligne */

    if (prec_ecriture >= 0) {
      if (close(prec_ecriture)) {
        fprintf(stderr,"Pere:Impossible de fermer ");
        fprintf(stderr,"le tuyau precedent en ecriture ");
        fprintf(stderr,"en fin de boucle\n");
      }
    }

    if (pid >=0) {
      waitpid(pid,NULL,0);
    }
    com=free_cmdliste(com);
    printf("myshell>");
  }
  exit(EXIT_SUCCESS);
}
```

16

Les *sockets* sous Unix

16.1 Introduction

Les *sockets* représentent la forme la plus complète de communication entre processus. En effet, elles permettent à plusieurs processus quelconques d'échanger des données, sur la même machine ou sur des machines différentes. Dans ce cas, la communication s'effectue grâce à un protocole réseau défini lors de l'ouverture de la socket (`IP`, `ISO`, `XNS`...).

Ce document ne couvre que les sockets réseau utilisant le protocole `IP` (*Internet Protocol*), celles-ci étant de loin les plus utilisées.

16.2 Les RFC *request for comments*

Il sera souvent fait référence dans la suite de ce chapitre à des documents appelés *request for comments* (RFC). Les RFC sont les documents décrivant les standards de l'Internet. En particulier, les protocoles de communication de l'Internet sont décrits dans des RFC. Les RFC sont numérotés dans l'ordre croissant au fur et à mesure de leur parution. Ils sont consultables sur de nombreux sites dont :

– `<URL:ftp://ftp.isi.edu/in-notes/>` (site de référence)

16.3 Technologies de communication réseau

Il existe rarement une connexion directe (un câble) entre deux machines. Pour pouvoir voyager d'une machine à l'autre, les informations doivent généralement traverser un certain nombre d'équipements intermédiaires. La façon de gérer le comportement de ces équipements a mené à la mise au point de deux technologies principales de communication réseau :

La commutation de circuits nécessite, au début de chaque connexion, l'établissement d'un circuit depuis la machine source vers la machine destination, à travers un nombre d'équipements intermédiaires fixé à l'ouverture de la connexion.

Cette technologie, utilisée pour les communications téléphoniques et par le système ATM (*Asynchronounous Transfer Mode*) présente un inconvénient majeur. En effet, une fois le circuit établi, il n'est pas possible de le modifier pour la communication en cours, ce qui pose deux problèmes :
– si l'un des équipements intermédiaires tombe en panne ou si le câble entre deux équipements successifs est endommagé, la communication est irrémédiablement coupée, il n'y a pas de moyen de la rétablir et il faut en initier une nouvelle ;
– si le circuit utilisé est surchargé, il n'est pas possible d'en changer pour en utiliser un autre.

En revanche, la commutation de circuits présente l'avantage de pouvoir faire très facilement de la réservation de bande passante puisqu'il suffit pour cela de réserver les circuits adéquats.

La commutation de paquets repose sur le découpage des informations en morceaux, dits *paquets*, dont chacun est expédié sur le réseau indépendamment des autres. Chaque équipement intermédiaire, appelé *routeur*, est chargé d'aiguiller les paquets dans la bonne direction, vers le routeur suivant. Ce principe est appelé *routage*. Il existe deux types de routage :

Le routage statique, qui stipule que les paquets envoyés vers tel réseau doivent passer par tel routeur. Le routage statique est enregistré dans la configuration du routeur et ne peut pas changer suivant l'état du réseau.

Le routage dynamique, qui repose sur un dialogue entre routeurs et qui leur permet de modifier le routage suivant l'état du réseau. Le dialogue entre routeurs repose sur un *protocole de routage* (tel que OSPF (RFC 2328), RIP (RFC 1058) ou BGP (RFC 1771) dans le cas des réseaux `IP`).

Le routage dynamique permet à la commutation de paquets de pallier les inconvénients de la commutation de circuits :
– si un routeur tombe en panne ou si un câble est endommagé, le routage est modifié dynamiquement pour que les paquets empruntent un autre chemin qui aboutira à la bonne destination (s'il existe un tel chemin, bien entendu) ;
– si un chemin est surchargé, le routage peut être modifié pour acheminer les paquets par un chemin moins embouteillé, s'il existe.

En revanche, contrairement à la commutation de circuits, il est assez difficile de faire de la réservation de bande passante avec la commutation de paquets. En contrepartie, celle-ci optimise l'utilisation des lignes en évitant la réservation de circuits qui ne seront que peu utilisés.

16.4 Le protocole `IP`

`IP` (*Internet Protocol*, RFC 791 pour IPv4, RFC 2460 pour IPv6) est un protocole reposant sur la technologie de commutation de paquets.

Chaque machine reliée à un réseau IP se voit attribuer une adresse, dite *adresse IP*, unique sur l'Internet, qui n'est rien d'autre qu'un entier sur 32 bits [1]. Afin de faciliter leur lecture et leur mémorisation, il est de coutume d'écrire les adresses IP sous la forme de quatre octets séparés par des points. Ainsi, la machine ensta.ensta.fr a pour adresse IP 147.250.1.1.

Les noms de machines sont cependant plus faciles à retenir et à utiliser mais, pour son fonctionnement, IP ne reconnaît que les adresses numériques. Un système a donc été mis au point pour convertir les noms en adresses IP et vice versa : le DNS (*Domain Name System*, RFC 1034 et RFC 1035).

Chaque paquet IP contient deux parties :

L'en-tête, qui contient diverses informations nécessaires à l'acheminement du paquet. Parmi celles-ci, on peut noter :
- l'adresse IP de la machine source ;
- l'adresse IP de la machine destination ;
- la taille du paquet.

Le contenu du paquet proprement dit, qui contient les informations à transmettre.

IP ne garantit pas que les paquets émis soient reçus par leur destinataire et n'effectue aucun contrôle d'intégrité sur les paquets reçus. C'est pourquoi deux protocoles plus évolués ont été bâtis au-dessus d'IP : UDP et TCP. Les protocoles applicatifs couramment utilisés, comme SMTP ou HTTP, sont bâtis au-dessus de l'un de ces protocoles (bien souvent TCP). Ce système en couches, où un nouveau protocole offrant des fonctionnalités plus évoluées est bâti au-dessus d'un protocole plus simple, est certainement l'une des clés du succès d'IP.

Le protocole UDP

UDP (*User Datagram Protocol*, RFC 768) est un protocole très simple, qui ajoute deux fonctionnalités importantes au-dessus d'IP :
- l'utilisation de numéros de ports par l'émetteur et le destinataire ;
- un contrôle d'intégrité sur les paquets reçus.

Les numéros de ports permettent à la couche IP des systèmes d'exploitation des machines source et destination de faire la distinction entre les processus utilisant les communication réseau.

En effet, imaginons une machine sur laquelle plusieurs processus sont susceptibles de recevoir des paquets UDP. Cette machine est identifiée par son adresse IP, elle ne recevra donc que les paquets qui lui sont destinés, puisque cette adresse IP figure dans le champ destination de l'en-tête des paquets. En revanche, comment le système d'exploitation peut-il faire pour déterminer à quel processus chaque paquet est destiné ? Si l'on se limite à l'en-tête IP, c'est impossible. Il faut donc ajouter, dans un en-tête spécifique au paquet UDP, des informations permettant d'aiguiller les informations vers le bon processus. C'est le rôle du numéro de port destination.

1. Ceci est vrai pour IPv4. Les adresses IPv6 sont sur 128 bits.

Le numéro de port source permet au processus destinataire des paquets d'identifier l'expéditeur ou au système d'exploitation de la machine source de prévenir le bon processus en cas d'erreur lors de l'acheminement du paquet.

Un paquet `UDP` est donc un paquet `IP` dont le contenu est composé de deux parties :

l'en-tête `UDP`, qui contient les numéros de ports source et destination, ainsi que le code de contrôle d'intégrité ;

le contenu du paquet `UDP` à proprement parler.

Au total, un paquet `UDP` est donc composé de trois parties :

1. l'en-tête `IP` ;
2. l'en-tête `UDP` ;
3. le contenu du paquet.

`UDP` ne permet cependant pas de gérer la retransmission des paquets en cas de perte ni d'adapter son débit à la bande passante disponible. Enfin, le découpage des informations à transmettre en paquets est toujours à la charge du processus source qui doit être capable d'adapter la taille des paquets à la capacité du support physique (par exemple, la taille maximale (MTU, *Maximum Transmission Unit*) d'un paquet transmissible sur un segment Ethernet est de 1500 octets, en-têtes `IP` et `UDP` compris, voir RFC 894).

Le protocole `TCP`

`TCP` (*Transmission Control Protocol*, RFC 793) est un protocole établissant un canal bidirectionnel entre deux processus.

De même qu'`UDP`, `TCP` ajoute un en-tête supplémentaire entre l'en-tête `IP` et les données à transmettre, en-tête contenant les numéros de port source et destination, un code de contrôle d'intégrité, exactement comme `UDP`, ainsi que diverses informations que nous ne détaillerons pas ici.

Un paquet `TCP` est donc un paquet `IP` dont le contenu est composé de deux parties :

l'en-tête `TCP`, qui contient, entre autres, les numéros de ports source et destination, ainsi que le code de contrôle d'intégrité ;

le contenu du paquet `TCP` à proprement parler.

Au total, un paquet `TCP` est donc composé de trois parties :

1. l'en-tête `IP` ;
2. l'en-tête `TCP` ;
3. le contenu du paquet.

Contrairement à `UDP`, qui impose aux processus voulant communiquer de gérer eux-même le découpage des données en paquets et leur retransmission en cas de perte, `TCP` apparaît à chaque processus comme un fichier ouvert en lecture et en écriture, ce qui lui confère une grande souplesse. À cet effet, `TCP` gère lui-même le découpage des

données en paquets, le recollage des paquets dans le bon ordre et la retransmission des paquets perdus si besoin est. TCP s'adapte également à la bande passante disponible en ralentissant l'envoi des paquets si les pertes sont trop importantes (RFC 1323) puis en ré-augmentant le rythme au fur et à mesure, tant que les pertes sont limitées.

16.5 Interface de programmation

L'interface de programmation des sockets peut paraître compliquée au premier abord, mais elle est en fait relativement simple parce que faite d'une succession d'étapes simples.

La fonction `socket()`

La fonction `socket()` permet de créer une socket de type quelconque :

```
#include <sys/types.h>
#include <sys/socket.h>

int s , domaine , type , protocole ;

/* initialisation des variables domaine , type et protocole */

s = socket ( domaine , type , protocole ) ;
if ( s == -1 )
{
   /* erreur : la socket n'a pas pu etre creee */
}
```

Le paramètre `domaine` précise la famille de protocoles à utiliser pour les communications. Il y a deux familles principales :

`PF_UNIX` pour les communications locales n'utilisant pas le réseau ;

`PF_INET` pour les communications réseaux utilisant le protocole `IP`.

Nous n'étudierons par la suite que la famille `PF_INET`.

Le paramètre `type` précise le type de la socket à créer :

`SOCK_STREAM` pour une communication bidirectionnelle sûre. Dans ce cas, la socket utilisera le protocole `TCP` pour communiquer.

`SOCK_DGRAM` pour une communication sous forme de datagrammes. Dans ce cas, la socket utilise le protocole `UDP` pour communiquer.

`SOCK_RAW` pour pouvoir créer soi-même ses propres paquets `IP` ou les recevoir sans traitement de la part de la couche réseau du système d'exploitation, ce qui est indispensable dans certains cas très précis.

Le paramètre `protocole` doit être égal à zéro.

La valeur renvoyée par la fonction `socket()` est -1 en cas d'erreur, sinon c'est un descripteur de fichier qu'on peut par la suite utiliser avec les fonctions `read()` et `write()`, transformer en pointeur de type `FILE *` avec `fdopen()`, etc.

Nous n'étudierons que les sockets de type `SOCK_STREAM` par la suite, celles-ci étant les plus utilisées.

Exemple de client `TCP`

La programmation d'un client `TCP` est fort simple et se déroule en cinq étapes :

1. création de la socket ;
2. récupération de l'adresse `IP` du serveur grâce au DNS ;
3. connexion au serveur ;
4. dialogue avec le serveur ;
5. fermeture de la connexion.

Création de la socket

Elle se fait simplement à l'aide de la fonction `socket()` :

```
#include <sys/types.h>
#include <sys/socket.h>

int client_socket ;

client_socket = socket ( PF_INET , SOCK_STREAM , 0 ) ;
if ( client_socket == -1 )
{
   /* erreur */
}
```

Interrogation du DNS

Le protocole `IP` ne connaît que les adresses `IP`. Il est donc nécessaire d'interroger le DNS pour récupérer l'adresse `IP` du serveur à partir de son nom. Il peut aussi être utile de récupérer le nom du serveur si le client est lancé avec une adresse `IP` comme argument. Pour cela, on utilise deux fonctions : `gethostbyname()` et `gethostbyaddr()`.

La fonction `gethostbyname()`

La fonction `gethostbyname()` permet d'interroger le DNS de façon à obtenir l'adresse `IP` d'une machine à partir de son nom :

```
#include <netdb.h>

struct hostent   *hostent ;
char             *nom = "ensta.ensta.fr" ;

hostent = gethostbyname ( nom ) ;
if ( hostent == NULL )
{
   /* erreur : le nom de la machine n'est pas declare dans le DNS */
}
```

Cette fonction prend en argument une chaîne de caractères (pointeur de type `char *`) contenant le nom de la machine et renvoie un pointeur vers une structure de type `hostent` (défini dans le fichier d'en-tête `<netdb.h>`). Cette structure contient trois membres qui nous intéressent :

`h_name` est le nom canonique [2] de la machine (pointeur de type `char *`) ;

`h_addr_list` est la liste des adresse `IP` [3] de la machine (pointeur de type `char **`).

`h_length` est la taille de chacune des adresses stockées dans `h_addr_list` (ceci sert à en permettre la recopie sans faire de suppositions sur la taille des adresses).

La fonction `gethostbyaddr()`

La fonction `gethostbyaddr()` permet d'interroger le DNS de façon à obtenir le nom canonique d'une machine à partir de son adresse `IP` :

```
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct hostent   *hostent ;
char             *ip = "147.250.1.1" ;
unsigned long    adresse ;

adresse = inet_addr ( ip ) ;

hostent = gethostbyaddr ( ( char * ) &adresse , sizeof ( adresse ) ,
                          AF_INET ) ;
if ( hostent == NULL )
{
   /* erreur : le nom de la machine n'est pas declare dans le DNS */
}
```

La fonction `inet_addr()` permet de transformer une chaîne de caractères représentant une adresse `IP` (comme 147.250.1.1) en entier long.

La fonction `gethostbyaddr()` prend en argument :

1. un pointeur de type `char *` vers l'adresse `IP` sous forme numérique (d'où la nécessité d'utiliser `inet_addr()`) ;
2. la taille de cette adresse ;
3. le type d'adresse utilisée, `AF_INET` pour les adresses `IP`.

Elle renvoie un pointeur vers une structure de type `hostent`, comme le fait la fonction examinée au paragraphe précédent `gethostbyname()`.

Connexion au serveur `TCP`

La connexion au serveur `TCP` se fait au moyen de la fonction `connect()` :

2. Une machine peut avoir plusieurs noms, comme `ensta.ensta.fr` et `h0.ensta.fr`. Le *nom canonique* d'une machine est son nom principal, ici `ensta.ensta.fr`. Les autres noms sont des *alias*.
3. Car à un nom peuvent être associées plusieurs adresses `IP` bien que ce soit assez rare en pratique.

```
if ( connect ( client_socket , ( struct sockaddr * ) &serveur_sockaddr_in ,
                               sizeof ( serveur_sockaddr_in ) ) == -1 )
{
   /* erreur : connexion impossible */
}
```

Cette fonction prend en argument :

1. le descripteur de fichier de la socket préalablement créée ;
2. un pointeur vers une structure de type `sockaddr` ;
3. la taille de cette structure.

Dans le cas des communications `IP`, le pointeur vers la structure de type `sockaddr` est un fait un pointeur vers une structure de type `sockaddr_in`, définie dans le fichier d'en-tête `<netinet/in.h>` (d'où la conversion de pointeur).

Cette structure contient trois membres utiles :

`sin_family` est la famille de protocoles utilisée, `AF_INET` pour `IP` ;

`sin_port` est le port `TCP` du serveur sur lequel se connecter ;

`sin_addr.s_addr` est l'adresse `IP` du serveur.

Le port `TCP` doit être fourni sous un format spécial. En effet les microprocesseurs stockent les octets de poids croissant des entiers courts (16 bits) et des entiers longs (32 bits) de gauche à droite (microprocesseurs dits *big endian* tels que le Motorola 68000) ou de droite à gauche (microprocesseurs dits *little endian* tels que les Intel). Pour les communications `IP`, le format *big endian* a été retenu. Il faut donc éventuellement convertir le numéro de port du format natif de la machine vers le format réseau au moyen de la fonction `htons()` (*host to network short*) pour les machines à processeur *little endian*. Dans le doute, on utilise cette fonction sur tous les types de machines (elle est également définie sur les machines à base de processeur *big endian*, mais elle ne fait rien).

L'adresse `IP` n'a pas besoin d'être convertie, puisqu'elle est directement renvoyée par le DNS au format réseau.

```
#include <netdb.h>
#include <string.h>
#include <netinet/in.h>

struct hostent       hostent ;
struct sockaddr_in   serveur_sockaddr_in ;
unsigned short       port ;

/* creation de la socket */

/* appel a gethostbyname() ou gethostbyaddr() */

memset ( &serveur_sockaddr_in , 0 ,
         sizeof ( serveur_sockaddr_in ) ) ;
serveur_sockaddr_in.sin_family      = AF_INET ;
serveur_sockaddr_in.sin_port        = htons ( port ) ;
memcpy ( &serveur_sockaddr_in.sin_addr.s_addr ,
         hostent->h_addr_list[0] ,
         hostent->h_length ) ;
```

La fonction `memset()` permet d'initialiser globalement la structure à zéro afin de supprimer toute valeur parasite.

Dialogue avec le serveur

Le dialogue entre client et serveur s'effectue simplement au moyen des fonctions `read()` et `write()`.

Cependant, notre client devra lire au clavier et sur la socket sans savoir à l'avance sur quel descripteur les informations seront disponibles. Il faut donc écouter deux descripteurs à la fois et lire les données sur un descripteur quand elles sont disponibles.

Pour cela, on utilise la fonction `select()` (cf chapitre 18). On indique donc les descripteurs qui nous intéressent (client_socket et l'entrée standard) puis on appelle la fonction `select()` :

```
fd_set rset ;

FD_ZERO ( &rset ) ;
FD_SET  ( client_socket , &rset ) ;
FD_SET  ( STDIN_FILENO  , &rset ) ;
if ( select ( client_socket + 1 , &rset , NULL , NULL , NULL ) == -1 )
{
   /* erreur */
}
```

Les arguments passés à la fonction `select()` sont `client_socket + 1`, correspondant au plus grand numéro de descripteur à surveiller plus un puisque l'entrée standard a 0 comme numéro de descripteur, et `&rset`, correspondant à l'ensemble des descripteurs de fichier à surveiller.

On utilise ensuite la fonction `FD_ISSET` pour déterminer quel descripteur est prêt :

```
if ( FD_ISSET ( client_socket , &rset ) )
{
   /* lire sur la socket */
}

if ( FD_ISSET ( STDIN_FILENO , &rset ) )
{
   /* lire sur l'entree standard */
}
```

Fermeture de la connexion

La socket se ferme simplement au moyen de la fonction `close()`. On peut aussi utiliser la fonction `shutdown()`, qui permet une fermeture sélective de la socket (soit en lecture, soit en écriture).

Client `TCP` complet

Listing 16.1 – *Exemple de client*

```
#include <sys/types.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

/* ------------------------------------------------- */

int socket_client (char *serveur , unsigned short port)
{
  int                 client_socket ;
  struct hostent      *hostent ;
  struct sockaddr_in  serveur_sockaddr_in ;

  client_socket = socket ( PF_INET , SOCK_STREAM , 0 ) ;
  if ( client_socket == -1 ) {
    perror ( "socket" ) ;
    exit ( EXIT_FAILURE ) ;
  }

  if ( inet_addr ( serveur ) == INADDR_NONE ) { /* nom */
    hostent = gethostbyname ( serveur ) ;
    if ( hostent == NULL ) {
      perror ( "gethostbyname" ) ;
      exit ( EXIT_FAILURE ) ;
    }
  } else   /* adresse IP */ {
    unsigned long addr = inet_addr ( serveur ) ;
    hostent = gethostbyaddr (( char * ) &addr , sizeof(addr),
                             AF_INET) ;
    if ( hostent == NULL ) {
      perror ( "gethostbyaddr" ) ;
      exit ( EXIT_FAILURE ) ;
    }
  }

  memset ( &serveur_sockaddr_in , 0 , sizeof ( serveur_sockaddr_in ) ) ;
  serveur_sockaddr_in.sin_family      = AF_INET ;
  serveur_sockaddr_in.sin_port        = htons ( port ) ;
  memcpy ( &serveur_sockaddr_in.sin_addr.s_addr ,
           hostent->h_addr_list[0] ,
           hostent->h_length) ;
  printf ( ">>> Connexion vers le port %d de %s [%s]\n" ,
           port , hostent->h_name ,
           inet_ntoa ( serveur_sockaddr_in.sin_addr ) ) ;

  if ( connect(client_socket, (struct sockaddr *) &serveur_sockaddr_in,
               sizeof ( serveur_sockaddr_in ) ) == -1 ) {
    perror ( "connect" ) ;
    exit ( EXIT_FAILURE ) ;
  }

   return client_socket ;
}

/* --------------------------------*/

int main ( int argc , char **argv )
{
```

```
  char             *serveur ;
  unsigned short   port ;
  int              client_socket ;

  if ( argc != 3 ) {
    fprintf ( stderr , "usage: %s serveur port\n" , argv[0] ) ;
    exit ( EXIT_FAILURE ) ;
  }

  serveur = argv[1] ;
  port = atoi ( argv[2] ) ;

  client_socket = socket_client ( serveur , port ) ;

  for ( ; ; ) {
    fd_set rset ;

    FD_ZERO ( &rset ) ;
    FD_SET  ( client_socket , &rset ) ;
    FD_SET  ( STDIN_FILENO  , &rset ) ;

    if (select( client_socket+1, &rset, NULL, NULL, NULL) == -1 ) {
      perror ( "select" ) ;
      exit ( EXIT_FAILURE ) ;
    }

      if (FD_ISSET(client_socket, &rset) ) {
        int             octets ;
        unsigned char   tampon[BUFSIZ] ;

        octets = read(client_socket, tampon, sizeof(tampon));
        if ( octets == 0 ) {
          close ( client_socket ) ;
          exit ( EXIT_SUCCESS ) ;
        }
        write(STDOUT_FILENO, tampon, octets);
      }

      if (FD_ISSET(STDIN_FILENO, &rset)) {
        int             octets ;
        unsigned char   tampon[BUFSIZ] ;

        octets = read(STDIN_FILENO, tampon, sizeof(tampon));
        if ( octets == 0 ) {
          close ( client_socket ) ;
          exit ( EXIT_SUCCESS ) ;
        }
        write(client_socket, tampon, octets);
      }
  }
  exit ( EXIT_SUCCESS ) ;
}
```

Exemple de serveur TCP

La programmation d'un serveur TCP est somme toute fort semblable à celle d'un client TCP :

1. création de la socket ;
2. choix du port à écouter ;

3. attente d'une connexion ;
4. dialogue avec le client ;
5. fermeture de la connexion.

Création de la socket

La création d'une socket serveur se déroule strictement de la même façon que la création d'une socket client :

```
#include <sys/types.h>
#include <sys/socket.h>

int serveur_socket ;

serveur_socket = socket ( PF_INET , SOCK_STREAM , 0 ) ;
if ( serveur_socket == -1 )
{
   /* erreur */
}
```

On appelle cependant une fonction supplémentaire, `setsockopt()`. En effet, après l'arrêt d'un serveur `TCP`, le système d'exploitation continue pendant un certain temps à écouter sur le port `TCP` que ce dernier utilisait afin de prévenir d'éventuels client dont les paquets se seraient égarés de l'arrêt du serveur, ce qui rend la création d'un nouveau serveur sur le même port impossible pendant ce temps-là. Pour forcer le système d'exploitation à réutiliser le port, on utilise l'option `SO_REUSEADDR` de la fonction `setsockopt()` :

```
int option = 1 ;

if ( setsockopt ( serveur_socket , SOL_SOCKET , SO_REUSEADDR ,
                  &option , sizeof ( option ) ) == -1 )
{
   /* erreur */
}
```

Choix du port à écouter

Le choix du port sur lequel écouter se fait au moyen de la fonction `bind()` :

```
#include <netinet/in.h>

struct sockaddr_in   serveur_sockaddr_in ;
unsigned short       port ;

memset ( &serveur_sockaddr_in , 0 , sizeof ( serveur_sockaddr_in ) ) ;
serveur_sockaddr_in.sin_family      = AF_INET ;
serveur_sockaddr_in.sin_port        = htons ( port ) ;
serveur_sockaddr_in.sin_addr.s_addr = INADDR_ANY ;

if ( bind ( serveur_socket , ( struct sockaddr * ) &serveur_sockaddr_in ,
                             sizeof ( serveur_sockaddr_in ) ) == -1 )
{
   /* erreur */
}
```

Cette fonction prend en argument :

1. le descripteur de fichier de la socket préalablement créée ;
2. un pointeur vers une structure de type `sockaddr` ;
3. la taille de cette structure.

La structure de type `sockaddr_in` s'initialise de la même façon que pour le client `TCP` (n'oubliez pas de convertir le numéro de port au format réseau au moyen de la fonction `htons()`). La seule différence est l'utilisation du symbole `INADDR_ANY` à la place de l'adresse `IP`. En effet, un serveur `TCP` est sensé écouter sur toutes les adresses `IP` dont dispose la machine, ce que permet `INADDR_ANY`.

La fonction `listen()` permet d'informer le système d'exploitation de la présence du nouveau serveur :

```
if ( listen ( serveur_socket , SOMAXCONN ) == -1 )
{
   /* erreur */
}
```

Cette fonction prend en argument :

1. le descripteur de fichier de la socket préalablement créée ;
2. le nombre maximum de connexions pouvant être en attente (on utilise généralement le symbole `SOMAXCONN` qui représente le nombre maximum de connexions en attente autorisé par le système d'exploitation).

Attente d'une connexion

La fonction `accept()` permet d'attendre qu'un client se connecte au serveur :

```
int                 client_socket ;
struct sockaddr_in  client_sockaddr_in ;
int                 taille = sizeof ( client_sockaddr_in ) ;

client_socket = accept ( serveur_socket ,
                         ( struct sockaddr * ) &client_sockaddr_in ,
                         &taille ) ;
if ( client_socket == -1 )
{
   /* erreur */
}
```

Cette fonction prend trois arguments :

1. le numéro de descripteur de la socket ;
2. un pointeur vers une structure de type `sockaddr`, structure qui sera remplie avec les paramètres du client qui s'est connecté ;
3. un pointeur vers un entier, qui sera rempli avec la taille de la structure ci-dessus.

En cas de succès de la connexion, la fonction `accept()` renvoie un nouveau descripteur de fichier représentant la connexion avec le client. Le descripteur initial peut encore

servir à de nouveaux clients pour se connecter au serveur, c'est pourquoi les serveurs appellent ensuite généralement la fonction `fork()` afin de créer un nouveau processus chargé du dialogue avec le client tandis que le processus initial continuera à attendre des connexions.

Après le `fork()`, chaque processus ferme le descripteur qui lui est inutile (exactement comme avec les tuyaux).

Dialogue avec le client

Le dialogue avec le client s'effectue de la même façon que dans le cas du client (voir paragraphe 16.5).

Fermeture de la connexion

La fermeture de la connexion s'effectue de la même façon que dans le cas du client (voir paragraphe 16.5).

Serveur `TCP` complet

Listing 16.2 – *Exemple de serveur*

```
#include <sys/types.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
/* ------------------------------------ */

int socket_serveur ( unsigned short port )
{
  int                 serveur_socket , option = 1 ;
  struct sockaddr_in  serveur_sockaddr_in ;

  serveur_socket = socket ( PF_INET , SOCK_STREAM , 0 ) ;
   if ( serveur_socket == -1 ) {
     perror ( "socket" ) ;
     exit ( EXIT_FAILURE ) ;
   }

   if (setsockopt(serveur_socket, SOL_SOCKET, SO_REUSEADDR,
                  &option, sizeof(option)) == -1 ) {
     perror ( "setsockopt" ) ;
     exit ( EXIT_FAILURE ) ;
   }

   memset ( &serveur_sockaddr_in , 0 , sizeof ( serveur_sockaddr_in ) ) ;
   serveur_sockaddr_in.sin_family      = AF_INET ;
   serveur_sockaddr_in.sin_port        = htons ( port ) ;
   serveur_sockaddr_in.sin_addr.s_addr = INADDR_ANY ;
```

```
   if (bind(serveur_socket, (struct sockaddr *)&serveur_sockaddr_in,
            sizeof(serveur_sockaddr_in)) == -1 ) {
     perror ( "bind" ) ;
     exit ( EXIT_FAILURE ) ;
   }

   if (listen(serveur_socket, SOMAXCONN) == -1 ) {
     perror ( "listen" ) ;
     exit ( EXIT_FAILURE ) ;
   }
   return serveur_socket ;
}

/* ------------------------------ */

void serveur ( int client_socket )
{
  for ( ; ; ) {
    fd_set rset ;

    FD_ZERO ( &rset ) ;
    FD_SET  ( client_socket , &rset ) ;
    FD_SET  ( STDIN_FILENO  , &rset ) ;

    if (select(client_socket+1, &rset, NULL, NULL, NULL) == -1) {
      perror ( "select" ) ;
      exit ( EXIT_FAILURE ) ;
    }

    if (FD_ISSET(client_socket, &rset) ) {
      int             octets ;
      unsigned char   tampon[BUFSIZ] ;

      octets = read(client_socket, tampon, sizeof(tampon));
      if ( octets == 0 ) {
        close ( client_socket ) ;
        printf ( ">>> Deconnexion\n" ) ;
        exit ( EXIT_SUCCESS ) ;
      }
      write (STDOUT_FILENO, tampon, octets);
    }

    if (FD_ISSET(STDIN_FILENO, &rset)) {
      int             octets ;
      unsigned char   tampon[BUFSIZ] ;

      octets = read(STDIN_FILENO, tampon, sizeof(tampon));
      if ( octets == 0 ) {
        close ( client_socket ) ;
        exit ( EXIT_SUCCESS ) ;
      }
      write(client_socket, tampon, octets);
    }
  }
}

/* ------------------------------ */

int main ( int argc , char **argv )
{
  unsigned short   port ;
  int              serveur_socket ;

  if ( argc != 2 ) {
```

```
    fprintf ( stderr , "usage: %s port\n" , argv[0] ) ;
    exit ( EXIT_FAILURE ) ;
  }

  port = atoi (argv[1]);
  serveur_socket = socket_serveur (port);

  for ( ; ; ) {
    fd_set rset ;

    FD_ZERO ( &rset ) ;
    FD_SET  ( serveur_socket , &rset ) ;
    FD_SET  ( STDIN_FILENO   , &rset ) ;

    if (select(serveur_socket+1, &rset, NULL, NULL, NULL) == -1) {
      perror ( "select" ) ;
      exit ( EXIT_FAILURE ) ;
    }

    if (FD_ISSET(serveur_socket, &rset)) {
      int                 client_socket ;
      struct sockaddr_in  client_sockaddr_in ;
      socklen_t           taille = sizeof ( client_sockaddr_in ) ;
      struct hostent      *hostent ;

      client_socket = accept (serveur_socket,
                              (struct sockaddr *) &client_sockaddr_in,
                              &taille);
      if ( client_socket == -1 ) {
        perror ( "accept" ) ;
        exit ( EXIT_FAILURE ) ;
      }

      switch (fork( )) {
      case -1 :   /* erreur */
        perror ( "fork" ) ;
        exit ( EXIT_FAILURE ) ;
      case 0 :   /* processus fils */
        close ( serveur_socket ) ;

        hostent = gethostbyaddr(
                       (char *) &client_sockaddr_in.sin_addr.s_addr,
                        sizeof(client_sockaddr_in.sin_addr.s_addr),
                       AF_INET);
           if ( hostent == NULL ) {
             perror ( "gethostbyaddr" ) ;
             exit ( EXIT_FAILURE ) ;
           }
           printf ( ">>> Connexion depuis %s [%s]\n" , hostent->h_name ,
                    inet_ntoa ( client_sockaddr_in.sin_addr ) ) ;

           serveur (client_socket);

           exit(EXIT_SUCCESS);
      default :   /* processus pere */
        close (client_socket);
      }
    }
  }
  exit ( EXIT_SUCCESS ) ;
}
```

17

Les *threads* POSIX sous Unix

17.1 Présentation

Rappels sur les *threads*

Dans les architectures à multi-processeurs à mémoire partagée (SMP pour *Symetric Multi Processing*), les *threads* peuvent être utilisés pour implémenter un parallélisme d'exécution. Historiquement, les fabricants ont implémenté leur propre version de la notion de *thread* et de ce point de vue la portabilité du code était un réel souci pour la plupart des développeurs.

Les systèmes Unix ont vu assez rapidement la naissance d'une interface de programmation en langage C des *threads* et cette normalisation est maintenant connue comme les *threads* POSIX, ou encore les pthreads.

Qu'est-ce qu'un *thread* ? Techniquement parlant, on peut voir un *thread* comme un flot d'instructions indépendant dont l'exécution sera conduite sous la houlette du système d'exploitation. Du point de vue du développeur, ce concept peut être vu comme une procédure de son programme dont l'exécution est indépendante du reste du déroulement du programme (ou presque !). Pour aller un peu plus loin dans l'esprit du développeur, imaginons qu'un programme contiennent plusieurs procédures. La notion de *thread* permet d'exécuter, **au sein de ce processus** de manière **simultannée / indépendante**, toutes ces procédures sous la haute gouvernance du système d'exploitation. Nous pourrions, si nous ne connaissions pas ce qu'est un processus sous UNIX, dire que ces procédures sont autant de processus indépendants. Sous certains UNIX, les *threads* sont gérés à l'intérieur du processus (leur « portée » est limitée à celle du processus).

Ces « processus indépendants » partagent nombre de choses avec le processus dont ils sont issus. Parmi ces choses nous retrouvons

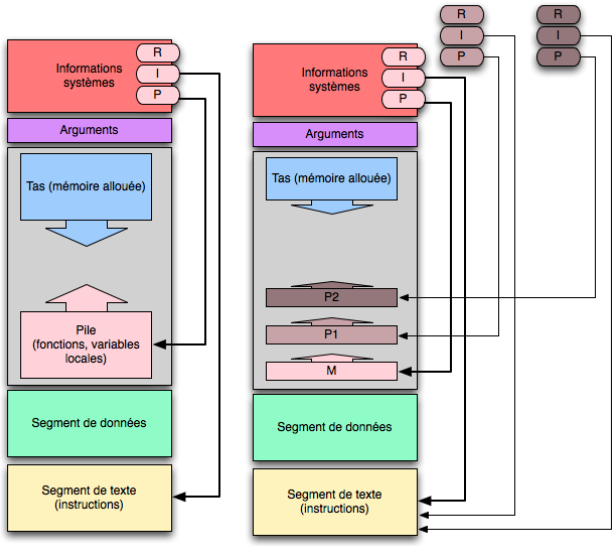
- le pid ;
- l'environnement ;
- le répertoire de travail ;
- les instructions du programme (du moins la partie commune), *i.e.* la section texte ;

– le tas (les allocations mémoires) ;
– les descripteurs de fichiers ouverts ;
– les bibliothèques partagées (branchement vers d'autres segments de texte partagés par d'autres processus).

Par contre, sont spécifiques à chaque *thread* les données suivantes :

– les registres ;
– la pile ;
– le pointeur d'instruction.

Un *thread* n'est donc pas un processus au sens où nous l'avons étudié, il n'a pas d'entrée dans la table des processus et il partage un espace mémoire avec le processus qui l'a créé.



Un *thread* permet d'exécuter un ensemble d'instructions de manière simultanée par plusieurs processeurs, quand l'architecture hôte le permet. Cette simultanéité impose quelques contraintes, notamment lors de l'accès à la mémoire. Certains accès ne peuvent pas être réalisés de manière concurrente (deux *threads* ne doivent pas essayer d'écrire au même moment au même endroit). Lorsque la **simultanéité d'exécution est impossible** (ou tout simplement néfaste au bon déroulement du programme) on entre dans ce que l'on appelle une **section critique** [1].

1. En effet, comme les différents *threads* partagent le même tas, les mêmes descripteurs de fichiers, des accès mal maîtrisés à ces données partagées peuvent avoir des effets de bord très néfastes. C'est donc dans cette procédure que les choses deviennent critiques. De plus, l'utilisation de mécanisme de synchronisation peut rendre cette section critique bloquante, à l'image de processus écrivain/lecteur dans un tuyau dont le lecteur attend l'écrivain qui lui même attend le lecteur…

L'API système de manipulation des *threads*

Pour utiliser les *threads* POSIX il convient d'inclure un fichier d'en-tête : `pthread.h`. Il est nécessaire de lier les programmes avec la bibliothèque qui contient le code des *threads* POSIX, à savoir `libpthread.*` par la directive `-lpthread` passée au compilateur.

`pthread_create()` : cette fonction permet de créer un *thread*. Elle prend en paramètre un pointeur vers un identifiant de *thread*, un pointeur vers un ensemble d'attributs, un pointeur vers la fonction à exécuter et enfin un pointeur vers des données lequel sera transmis comme paramètre à la fonction à exécuter :

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg)
```

- `pthread_t *restrict thread` : pointeur vers un type pthread qui sera rempli par la fonction et servira à identifier le *thread* ;
- `const pthread_attr_t *restrict attr` : pointeur vers une structure d'attributs, peut être le pointeur NULL dans ce cas les attributs par défaut sont utilisés ;
- `void *(*start_routine)(void *)` : pointeur vers la fonction qui sera exécutée. Cette fonction accepte en paramètre une adresse et retourne une adresse ;
- `void *restrict arg` : adresse des données qui seront éventuellement passées à la fonction exécutée. Si l'on ne souhaite pas passer de données, on peut passer NULL.

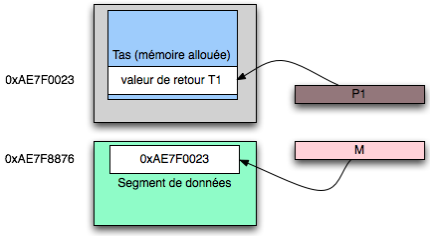
L'utilisation du mot clé `restrict` n'est pas une chose utile au compilateur. Ce mot clé sert simplement à attirer votre attention sur le fait que la variable à laquelle est accolé ce mot clé ne doit être accédée, du point de vue mémoire, que par cette fonction à cet instant. Il est donc judicieux que l'identifieur de *thread*, les attributs et les arguments soient uniques à chaque appel de la fonction `pthread_create()`. Comme la plupart des appels systèmes, cette fonction retourne 0 quand tout se passe bien et un nombre différent de zéro en cas d'erreur.

`pthread_join()` : cette fonction permet d'attendre la fin d'un *thread*, un peu comme le ferait `waitpid()`. Cette fonction bloque jusqu'à ce que le *thread* retourne. :

```
int pthread_join(pthread_t thread,
                 void **value_ptr);
```

- `pthread_t thread` : identifiant du thread dont on attend la terminaison ;
- `void **value_ptr` : pointeur permettant de récupérer l'adresse utilisée par le *thread* pour consigner sa valeur de retour. Ce pointeur peut être NULL si l'on ne désire pas connaître cette valeur de retour.

Cette fonction ne doit être appelée qu'une seule fois par identifiant de *thread*, sinon son comportement est indéfini. La fonction appelée lors de la création du *thread* renvoie une adresse mémoire **allouée sur le tas** dans laquelle elle place sa valeur de retour. C'est cette adresse qui sera placée dans la variable `value_ptr` dont on donne l'adresse.



L'exemple qui suit donne la marche à suivre pour créer un *thread* et attendre sa terminaison avec la récupération de sa valeur de terminaison. Il faut en effet bien avoir à l'esprit que lors de sa terminaison, tout ce qui se trouve dans la pile du *thread* (variables locales) est perdu. Retourner l'adresse d'une variable locale serait donc aberrant :

> *Quand un* thread *se termine, le résultat d'un accès aux variables locales (auto) du* thread *est indéfini. Ainsi, une référence à des variables locales du* thread *se terminant ne doit pas être utilisée comme paramètre de* `pthread_exit()`.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void *f(void *param)
{
  int *num = (int *)param;
  int *retval=NULL;
  /*Allocation sur le tas*/
  if ((retval = malloc(2*sizeof(int))) == NULL) {
    perror("malloc in thread failed");
    return NULL;
  }
  if (*num == 0) {
        retval[0] = 1; retval[1] = 2;
  } else {
    retval[0] = 10; retval[1] = 20;
  }
  return (void *)retval;
}

int main(int argc, char **argv)
{
  pthread_t t[2];
  int n[2] = {0,1};
  int ret;
  void *value_ptr;
  int *return_value_a;
  int *return_value_b;
  if ((ret = pthread_create(&(t[0]),
        NULL,f,(void *)(&(n[0])))) !=0)
    exit(EXIT_FAILURE);

  if ((ret = pthread_create(&(t[1]),
        NULL,f,(void *)(&(n[1])))) !=0)
```

```
    exit(EXIT_FAILURE);

  if (pthread_join(t[0],&value_ptr) != 0)
        exit(EXIT_FAILURE);
  return_value_a = (int *)value_ptr;
  printf("Thread 0 returns with: %d %d (stored at %p)\n",
          return_value_a[0],return_value_a[1],value_ptr);

  if (pthread_join(t[1],&value_ptr) != 0)
        exit(EXIT_FAILURE);
  return_value_b = (int *)value_ptr;
  printf("Thread 1 ends with: %d %d (stored at %p)\n",
          return_value_b[0],return_value_b[1],value_ptr);
  exit(EXIT_SUCCESS);
}
```

Synchronisation par verrou

Comme nous pouvons définir des points de synchronisation entre processus en utilisant un dialogue par tuyau et des mécanismes bloquants de lecture / écriture, il est possible de réaliser la même chose sur les *threads* mais par l'utilisation de verrou selon un principe d'exclusion mutuelle (d'où le nom de `mutex`). Le principe est assez simple, il s'agit d'essayer de prendre un verrou, lequel ne peut être pris qu'une seule fois. La prise du verrou, qui est une opération bloquante, par le premier *thread* empêche un autre *thread* d'avancer si ce dernier essaye lui aussi de prendre le même verrou.

On se sert généralement de ces verrous afin de réaliser deux types d'opérations :

- la synchronisation de plusieurs *threads* par un autre (que nous appellerons le *thread* de synchronisation). Chaque *thread* essaye à l'issue de l'exécution d'un certain nombre d'instructions et avant d'entamer le reste de leur travail de prendre un verrou. Ce verrou est pris dès le départ par le *thread* de synchronisation, puis est relâché à un instant précis (dans le temps ou par analyse de la valeur de certaines variables) ;
- la protection des accès en écriture sur le tas (qui équivaut à une synchronisation en mémoire). Si plusieurs *threads* écrivent au même endroit en même temps (cela peut arriver avec une architecture `SMP`), le résultat peut devenir imprévisible pour celui qui lit. En effet les lectures font appel à des variables dont la valeur peut très bien se situer dans le cache d'un des processeurs (c'est parfois même demandé par le compilateur). Il est donc impératif de synchroniser les caches mémoires avant de réaliser cette lecture car il serait dangereux de croire que le contenu de la mémoire reflète exactement le contenu du cache. On demande donc une synchronisation entre la mémoire et les caches des processeurs. Les différents *threads* devant accéder à la même portion de la mémoire essaieront dans un premier temps de prendre un verrou, réaliseront leur écriture, puis relâcheront le verrou. Cela aura pour cause de synchroniser l'état de la mémoire. Mais c'est véritablement la combinaison `unlock` / `lock` qui réalisera ce que l'on appelle une cohérence de cache. C'est aussi quelque part assez tranquillisant, imaginons

en effet un seul instant que le verrou (situé quelque part dans le tas, donc en mémoire partagée) se trouve dans un registre du processeur 1…

Les verrous se gèrent à l'aide de plusieurs appels systèmes dont nous exploiterons les deux qui suivent.

`pthread_mutex_lock()` permet de prendre un verrou et de le rendre ainsi indisponible pour les autres fonctions. Cette fonction est bloquante, donc si le verrou est déjà pris, l'appel à `pthread_mutex_lock()` va suspendre l'exécution du *thread* jusqu'à ce que le verrou puisse être pris.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

– `pthread_mutex_t *mutex` : adresse du verrou que l'on désire prendre. Cette adresse doit naturellement correspondre à une adresse mémoire du tas ou de la zone de données mais sûrement pas à une variable locale qui ne pourrait pas être partagée entre les *threads*.

`pthread_mutex_unlock()` permet de relâcher le verrou et de le rendre ainsi disponible pour d'autres fonctions. Lorsque celles-ci prendront le verrou, il y aura (il devrait y avoir) une cohérence de cache afin de garantir que la mémoire partagée accédée après la prise du verrou sera correcte.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

– `pthread_mutex_t *mutex` : adresse du verrou que l'on désire relâcher. Même remarque que pour la fonction précédente.

Réveil de *threads*

Il est possible de suspendre l'exécution d'un *thread* et d'attendre qu'une condition devienne valide pour entamer l'exécution d'une partie du code. Ce genre d'attente est assez agréable car elle permet de réveiller plusieurs *threads* de manière simultanée alors que la prise de verrou était mutuellement exclusive. Cependant, les différentes fonctions que nous allons exposer doivent être manipulées avec rigueur, sinon le risque d'impasse (`deadlock`) est garanti ! En général les conditions sont initialisées, puis chaque *thread* attend que la condition devienne correcte, un autre *thread* signale que la condition devient correct ce qui réveille les autres *threads*.

Une condition est toujours associée à un verrou et ce pour éviter les courses à la mort (*race conditions*). Imaginons en effet que pendant que le *thread* 1 se prépare à attendre la condition, un autre *thread* la signale comme valide. L'opération de mise en attente étant un enchaînement d'opérations, ces dernières, et notamment celle qui vérifie la condition, sont placées sous l'effet de synchronisation d'un verrou.

Plusieurs fonctions systèmes permettent de gérer les conditions.

`pthread_cond_init()` : cette fonction permet d'initialiser une condition. Elle s'utilise très souvent avec les attributs par défaut.

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

– `pthread_cond_t *cond` : l'adresse d'une variable de condition ;
– `pthrad_condattr_t *cond_attr` : les attributs liées à la condition. Si l'on veut choisir les attributs par défaut on peut passer une adresse NULL.

`pthread_cond_wait()` : cette fonction permet d'attendre une condition. Cette fonction a besoin d'un verrou pour se protéger contre d'éventuelles courses à la mort. Cette fonction va tout d'abord relâcher le verrou passé en paramètre et attendre le signal de condition sans consommer de CPU (suspension d'activité). À la réception de ce dernier, elle reprendra le verrou et retournera à la fonction appelante. Il est donc **primordial** d'avoir pris le verrou passé en paramètre avant d'appeler cette fonction. Il est tout aussi primordial de relâcher le verrou lorsque que la fonction d'attente se termine. Voici un extrait du manuel :

> ...`pthread_cond_wait()` déverrouille de manière atomique le mutex (comme le ferait `pthread_mutex_unlock()`) et attend que la variable de condition cond soit signalée. L'exécution du *thread* est suspendue et ne consomme aucun temps `CPU` jusqu'à ce que le signal associé à la variable de condition arrive. Le *mutex* doit être verrouillé par la *thread* appelant avant l'appel de la fonction `pthread_cond_wait`. Avant de revenir à la fonction appelante, `pthread_cond_wait` verrouille de nouveau le *mutex* (comme le ferait `pthread_mutex_lock()`).
>
> Déverrouiller le mutex et suspendre l'activité en attendant la variable de condition est une opération atomique. Ainsi, si tous les *threads* essayent d'entrée de jeu d'acquérir le *mutex* avant de signaler la condition, cela garantit que la condition ne peut être signalée (et donc aussi ignorée) entre le temps où le *thread* verrouille le *mutex* et le temps où il commence à attendre le signal de condition. ...

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

– `pthread_cond_t *cond` : l'adresse d'une variable de condition ;
– `pthread_mutex_t *mutex` : l'adresse d'un verrou.

`pthread_cond_broadcast()` permet de signaler à **tous les *threads*** que la condition est levée et permet ainsi de tous les réveiller.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

– `pthread_cond_t *cond` : l'adresse de la variable de condition.

Il existe aussi une fonction permettant de ne réveiller qu'un unique *thread* parmi ceux qui sont suspendus dans l'attente du signal associé à une même variable de condition.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

– `pthread_cond_t *cond` : l'adresse de la variable de condition.

17.2 Exercices

Question 1

Ecrire un programme multi-threadé permettant de calculer moyenne et écart type d'une suite de nombres de manière parallèle. La fonction `main()` initialisera un tableau de nombres entiers aléatoires puis elle créera deux *threads*. Le premier calculera et retournera la somme des nombres, le second calculera et retournera la somme des carrés des nombres. La fonction `main()` attendra la fin des deux *threads* et utilisera les valeurs de retour pour donner moyenne et écart type. On rappelle que

$$\mu = \frac{1}{N}\sum_{i}^{N}\alpha_i$$

et

$$\sigma^2 = \frac{1}{N}\sum_{i}^{N}\alpha_i^2 - \left(\frac{1}{N}\sum_{i}^{N}\alpha_i\right)^2$$

Ecrire une version simple et sans utiliser de *thread* de ce programme. Mesurer les temps d'exécution (`man time`).

Question 2

Reprendre la même problématique de calcul de moyenne et d'écart type mais en divisant le tableau en N parties (N sera passé en ligne de commande). Créer $2 \times N$ *threads* qui travailleront chacun sur une partie du tableau. Analyser les performances en fonction de N.

Question 3

On désire peaufiner le programme précédent. Un premier *thread* réalise l'initialisation du tableau aléatoire. Deux autres *threads* attendent une levée de condition pour démarrer le calcul de la somme et de la somme des carrés. Lorsque le premier *thread* termine le remplissage du tableau, il émet un signal de réveil à destination des deux *threads* de calcul.

17.3 Corrigés

Premier exercice

Version séquentielle sans utilisation de *thread*. On utilise un chronomètre dans la fonction `main()` pour mesure le temps consommé par les calculs séquentielles. Ceci permet de ne pas prendre en compte la génération du tableau de nombres.

Listing 17.1 – *Calcul sans utilisation des* threads

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#define MAX_NUM 10000000

struct my_chrono {
  struct timeval tp_start;
  struct timeval tp_current;
};

void start_chrono(struct my_chrono *c)
{
  gettimeofday(&(c->tp_start),NULL);
}

int stop_chrono(struct my_chrono *c)
{
  int res;
  gettimeofday(&(c->tp_current),NULL);
  res = (int)(((c->tp_current.tv_sec) -
               (c->tp_start.tv_sec)) * 1000 +
              ((c->tp_current.tv_usec) -
               (c->tp_start.tv_usec)) / 1000 );
  return res;
}

void sequentiel(int *tt, float *m, float *e)
{
  long k;
  float sum, sumc;
  for(k=0,sum = 0.0;k<MAX_NUM;k++) {
    sum += tt[k];
  }
  for(k=0,sumc = 0.0;k<MAX_NUM;k++) {
    sumc += tt[k] * tt[k];
  }
  *m = sum/MAX_NUM;
  *e = (float)sqrt(sumc/MAX_NUM - *m * *m);
  return;
}

int main(int argc, char **argv)
{
  int *num = NULL;
  long k;
  int r;
  float m,e;
  struct my_chrono c;

  if ((num =malloc(MAX_NUM*sizeof(int))) == NULL) {
```

```
    perror("allocation impossible");
    exit(EXIT_FAILURE);
  }
  for(k=0;k<MAX_NUM;k++) {
    num[k] = (int)random() % 1000;
  }
  start_chrono(&c);
  sequentiel(num,&m,&e);
  r = stop_chrono(&c);
  print("Moyenne (S): %f Ecart type: %f Temps ecoule:%d ms\n",m,e,r);
  exit(EXIT_SUCCESS);
}
```

L'exécution de ce programme donne les résultats suivants (valeur non reproductible et totalement dépendante de la charge, et du CPU, voir à ce propos le complément sur l'architecture donné en fin de document) :

```
moi@ici:> time ./stat_noth
Moyenne (S): 499.2311 Ecart type: 282.3061 Temps ecoule:810 ms
moi@ici:>
```

Version multi-threadé du programme (toujours avec un chronomètre) :

Listing 17.2 – *Version parallèle du calcul*

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>
#define MAX_NUM 10000000

struct my_chrono {
  struct timeval tp_start;
  struct timeval tp_current;
};

void start_chrono(struct my_chrono *c)
{
  gettimeofday(&(c->tp_start),NULL);
}

int stop_chrono(struct my_chrono *c)
{
  int res;
  gettimeofday(&(c->tp_current),NULL);
  res = (int)(((c->tp_current.tv_sec) -
               (c->tp_start.tv_sec)) * 1000 +
              ((c->tp_current.tv_usec) -
               (c->tp_start.tv_usec)) / 1000 );
  return res;
}

void *calcul_sum(void * param)
{
  int *tt=(int *)param;
  float *retval;
  long k;
  if ((retval = malloc(sizeof(float))) == NULL) return NULL;

  for(k=0,*retval = 0.0;k<MAX_NUM;k++) {
```

```
    *retval += tt[k];
  }
  return ((void *)retval);
}

void *calcul_sumc(void * param)
{
  int *tt=(int *)param;
  float *retval;
  long k;
  if ((retval = malloc(sizeof(float))) == NULL) return NULL;

  for(*retval = 0.0,k=0;k<MAX_NUM;k++) {
    *retval += tt[k] * tt[k];
  }
  return ((void *)retval);
}

int main(int argc, char **argv)
{
  pthread_t t[2];
  int ret;
  int *num = NULL;
  long k;
  int r;
  float *sum, *sumc;
  float m,e;

  struct my_chrono c;

  if ((num =malloc(MAX_NUM*sizeof(int))) == NULL) {
    perror("allocation impossible");
    exit(EXIT_FAILURE);
  }
  for(k=0;k<MAX_NUM;k++) {
    num[k] = (int)random() % 1000;
  }

  start_chrono(&c);
  if ((ret = pthread_create(t+0,NULL,calcul_sum,(void *)num)) != 0) {
    perror("calcul_sum impossible");
    exit(EXIT_FAILURE);
  }

  if ((ret = pthread_create(t+1,NULL,calcul_sumc,(void *)num)) != 0) {
    perror("calcul_sumc impossible");
    exit(EXIT_FAILURE);
  }

  if ((ret = pthread_join(t[0],(void *)&sum)) != 0) {
    perror("calcul_sum non joignable");
    exit(EXIT_FAILURE);
  }

  if ((ret = pthread_join(t[1],(void *)&sumc)) != 0) {
    perror("calcul_sumc non joignable");
    exit(EXIT_FAILURE);
  }
  r = stop_chrono(&c);
  if (sum == NULL || sumc == NULL) {
    fprintf(stderr,"Terminaison manquee\n");
    exit(EXIT_FAILURE);
  }
  m = *sum/MAX_NUM;
```

```
  e = (float)sqrt(*sumc/MAX_NUM - m*m);
  free(sum);free(sumc);
  printf("Moyenne (T): %f Ecart type: %f Temps ecoule:%d ms\n",m,e,r);

  exit(EXIT_SUCCESS);
}
```

Temps d'exécution :

```
moi@ici:> time ./stat_th
Moyenne (T): 499.2311 Ecart type: 282.3061 Temps ecoule:511 ms
moi@ici:>
```

Deuxième exercice

On ne se sert plus de la valeur de retour des *threads* pour obtenir les sommes. Puisqu'il faut passer une structure plus complexe dans les paramètres d'un *thread*, on glisse à l'intérieur de cette structure :

- l'accès à l'adresse du tableau, c'est un paramètre que tous les *threads* utilisent, mais il est accédé en lecture ;
- l'accès à la valeur de la somme et de la somme des carrés, mais ce seront des paramètres locaux à chaque *thread* ;
- l'accès aux index de début et de fin de calcul dans le tableau général, qui sont, certes, accédés en lecture, mais différents d'un *thread* à l'autre.

Le programme se déroule donc en plusieurs étapes. La première étape concerne l'allocation du tableau général, l'allocation du tableau d'identifiant de *thread* et l'allocation des structures de paramètres (avec mise en place des valeurs qui vont bien). La deuxième étape lance le remplissage du tableau, le démarrage des *threads* puis leur attente et enfin l'affichage du résultat et du temps écoulé.

Listing 17.3 – *Passage de paramètre*

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>
#define MAX_NUM 10000000

struct t_param {
  int *tt;
  long start_idx,end_idx;
  float sum,sumc;
};

struct my_chrono {
  struct timeval tp_start;
  struct timeval tp_current;
};

void start_chrono(struct my_chrono *c)
{
  gettimeofday(&(c->tp_start),NULL);
```

```
}

int stop_chrono(struct my_chrono *c)
{
  int res;
  gettimeofday(&(c->tp_current),NULL);
  res = (int)(((c->tp_current.tv_sec) -
               (c->tp_start.tv_sec)) * 1000 +
              ((c->tp_current.tv_usec) -
               (c->tp_start.tv_usec)) / 1000 );
  return res;
}

void *calcul_sum(void * param)
{
  struct t_param *prm = (struct t_param *)param;
  long k;
  for(k=prm->start_idx;k<prm->end_idx;k++) {
    prm->sum += prm->tt[k];
  }
  return (NULL);
}

void *calcul_sumc(void * param)
{
  struct t_param *prm = (struct t_param *)param;
  long k;
  for(k=prm->start_idx;k<prm->end_idx;k++) {
    prm->sumc += (prm->tt[k]*prm->tt[k]);
  }
  return (NULL);
}


int main(int argc, char **argv)
{
  pthread_t *t=NULL;
  struct t_param *p=NULL;
  int *tt=NULL;
  int num_thread;

  struct my_chrono c;

  long k;
  int ret,r;
  float sum,sumc,m,e;

  num_thread = (argc >= 2 ? atoi(argv[1]):4);

  if ((p =malloc(num_thread*sizeof(struct t_param))) == NULL) {
    perror("allocation p impossible");
    exit(EXIT_FAILURE);
  }
  if ((t =malloc(2*num_thread*sizeof(pthread_t))) == NULL) {
    perror("allocation t impossible");
    exit(EXIT_FAILURE);
  }
  if ((tt =malloc(MAX_NUM*sizeof(int))) == NULL) {
    perror("allocation tt impossible");
    exit(EXIT_FAILURE);
  }

  for(k=0;k<num_thread;k++) {
    p[k].tt = tt;
```

```
    p[k].start_idx = k*MAX_NUM/num_thread;
    p[k].end_idx = (k+1)*MAX_NUM/num_thread;
    p[k].sum = p[k].sumc = 0.0;
  }
  for(k=0;k<MAX_NUM;k++) {
    tt[k] = (int)random() % 1000;
  }

  start_chrono(&c);

  for(k=0;k<num_thread;k++) {
    if ((ret = pthread_create(t+2*k,NULL,
                              calcul_sum,(void *)(p+k))) != 0) {
      perror("calcul_sum impossible");
      exit(EXIT_FAILURE);
    }
    if ((ret = pthread_create(t+2*k+1,NULL,
                              calcul_sumc,(void *)(p+k))) != 0) {
      perror("calcul_sum impossible");
      exit(EXIT_FAILURE);
    }
  }
  for(k=0;k<num_thread;k++) {
    if ((ret = pthread_join(t[2*k],NULL)) != 0) {
      perror("calcul_sum impossible");
      exit(EXIT_FAILURE);
    }
    if ((ret = pthread_join(t[2*k+1],NULL)) != 0) {
      perror("calcul_sum impossible");
      exit(EXIT_FAILURE);
    }
  }
  r = stop_chrono(&c);
  for(k=0,sum=0.0,sumc=0.0;k<num_thread;k++) {
    sum += p[k].sum;
    sumc += p[k].sumc;
  }
  m = sum/MAX_NUM;
  e = (float)sqrt(sumc/MAX_NUM - m*m);
  printf("Moyenne (T): %f Ecart type: %f Temps ecoule: %dms\n",m,e,r);
  exit(EXIT_SUCCESS);
}
```

C'est naturellement le dernier *thread* qui fixe l'arrêt du programme. Le fait d'attendre dans l'ordre les *threads* peut parfois entraîner un certain retard dans la mesure où le *thread* 1 peut très bien se terminer en dernier en raison de l'ordonnanceur. Il est impossible de prédire lequel des *threads* se terminera en premier.

Troisième exercice

La structure précédente se voit complétée par deux adresses, celle d'un mutex et celle d'une condition. Le *thread* d'initialisation va envoyer le signal de lever de condition à tous les *threads* qui sont en attente, à savoir les deux *threads* de calcul qui se sont arrêtés sur l'attente de condition. On suit scrupuleusement le manuel qui signale qu'il faut impérativement avoir verrouillé le *mutex* avant de se placer sur l'attente de condition. On continue ce respect du manuel en débloquant le verrou lorsque la

fonction d'attente conditionnelle retourne. La sortie du programme dont le code source suit est par exemple :

```
moi@ici:> ./waitsomeone
init_tableau starts
sum trie to start
sumc trie to start
init_tableau stops
sum starts
sumc starts
sum stops
sumc stops
Moyenne (T): 171.7986 Ecart type: 382.6317 Temps ecoule: 3213ms
```

Listing 17.4 – *Utilisation des conditions*

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>
#define MAX_NUM 100000000

struct t_param {
  int *tt;
  int numero;
  long num_nombre;
  float sum,sumc;
  pthread_mutex_t *mutex;
  pthread_cond_t *condition;
};

struct my_chrono {
  struct timeval tp_start;
  struct timeval tp_current;
};

void start_chrono(struct my_chrono *c)
{
  gettimeofday(&(c->tp_start),NULL);
}

int stop_chrono(struct my_chrono *c)
{
  int res;
  gettimeofday(&(c->tp_current),NULL);
  res = (int)(((c->tp_current.tv_sec) -
               (c->tp_start.tv_sec)) * 1000 +
              ((c->tp_current.tv_usec) -
               (c->tp_start.tv_usec)) / 1000 );
  return res;
}

void *init_tableau(void * param)
{
  struct t_param *prm = (struct t_param *)param;
  long k;
  fprintf(stderr,"init_tableau starts\n");
  for(k=0;k<prm->num_nombre;k++) {
    prm->tt[k] = (int)random() % 1000;
  }
  fprintf(stderr,"init_tableau stops\n");
```

```
  pthread_cond_broadcast(prm->condition);
  return (NULL);
}

void *calcul_sum(void * param)
{
  struct t_param *prm = (struct t_param *)param;
  long k;
  fprintf(stderr,"sum trie to start\n");
  pthread_mutex_lock(prm->mutex);
  pthread_cond_wait(prm->condition,prm->mutex);
  pthread_mutex_unlock(prm->mutex);
  fprintf(stderr,"sum starts\n");
  for(k=0;k<prm->num_nombre;k++) {
    prm->sum += prm->tt[k];
  }
  fprintf(stderr,"sum stops\n");
  return (NULL);
}

void *calcul_sumc(void * param)
{
  struct t_param *prm = (struct t_param *)param;
  long k;
  fprintf(stderr,"sumc trie to start\n");
  pthread_mutex_lock(prm->mutex);
  pthread_cond_wait(prm->condition,prm->mutex);
  pthread_mutex_unlock(prm->mutex);
  fprintf(stderr,"sumc starts\n");
  for(k=0;k<prm->num_nombre;k++) {
    prm->sumc += (prm->tt[k]*prm->tt[k]);
  }
  fprintf(stderr,"sumc stops\n");
  return (NULL);
}


int main(int argc, char **argv)
{
  pthread_t *t=NULL;
  struct t_param *p=NULL;
  int *tt=NULL;
  int num_thread;
  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
  pthread_cond_t condition = PTHREAD_COND_INITIALIZER;

  struct my_chrono c;

  long k;
  int ret,r;
  float sum,sumc,m,e;

  num_thread = 3;

  if ((p =malloc(num_thread*sizeof(struct t_param))) == NULL) {
    perror("allocation p impossible");
    exit(EXIT_FAILURE);
  }
  if ((t =malloc(2*num_thread*sizeof(pthread_t))) == NULL) {
    perror("allocation t impossible");
    exit(EXIT_FAILURE);
  }
  if ((tt =malloc(MAX_NUM*sizeof(int))) == NULL) {
    perror("allocation tt impossible");
```

```
    exit(EXIT_FAILURE);
  }

  for(k=0;k<num_thread;k++) {
    p[k].tt = tt;
    p[k].numero = k;
    p[k].num_nombre = MAX_NUM;
    p[k].sum = p[k].sumc = 0.0;
    p[k].mutex = &mutex;
    p[k].condition = &condition;
  }

  start_chrono(&c);
  ret = pthread_create(t+0,NULL,init_tableau,(void *)(p+0));
  if (ret != 0) {
    perror("init_tableau impossible");
    exit(EXIT_FAILURE);
  }
  ret = pthread_create(t+1,NULL,calcul_sum,(void *)(p+1));
  if (ret != 0) {
    perror("calcul_sum impossible");
    exit(EXIT_FAILURE);
  }
  ret = pthread_create(t+2,NULL,calcul_sumc,(void *)(p+2));
  if (ret != 0) {
    perror("calcul_sumc impossible");
    exit(EXIT_FAILURE);
  }

  for(k=0;k<num_thread;k++) {
    if ((ret = pthread_join(t[k],NULL)) != 0) {
      perror("jonction impossible");
      exit(EXIT_FAILURE);
    }
  }
  r = stop_chrono(&c);
  for(k=0,sum=0.0,sumc=0.0;k<num_thread;k++) {
    sum += p[k].sum;
    sumc += p[k].sumc;
  }
  m = sum/MAX_NUM;
  e = (float)sqrt(sumc/MAX_NUM - m*m);
  printf("Moyenne (T): %f Ecart type: %f Temps ecoule: %dms\n",m,e,r);
  exit(EXIT_SUCCESS);
}
```

17.4 Architectures et programmation parallèle

Deux types d'architectures matérielles apportant des gains en programmation parallèle existent :

- les architectures de type « multi-processeurs » ;
- les architectures de type « multi-cœurs ».

Les différences, même si elles peuvent paraître minces vues de l'extérieur, sont pourtant fondamentales. Dans le premier cas, chaque processeur est doté de sa batterie de caches (L1, L2), d'un *slot* sur la carte mère, alors que dans le deuxième cas, la puce intègre deux processeurs qui peuvent partager leur cache de niveau 2 et ne nécessitent pas

de passage par la carte mère pour la communication. Notons toutefois que si la série Athlon 64X2 d'AMD permettait un dialogue interne des deux cœurs, le Pentium D 820, lui, passait par la carte mère pour la communication entre les deux cœurs (l'ouverture du boitier montrait clairement la présence de 2 processeurs, donc pratiquement une architecture de type bi-processeur et non bi-cœur).

Cette différence est fondamentale dans la mesure où, nous le savons bien maintenant, le goulet d'étranglement dans une architecture reste toujours la communication entre les différents constituants. Lorsque ce goulet porte sur le cœur même de l'architecture, *i.e.* le processeur, cela peut avoir des conséquences assez désastreuses sur les temps de calcul.

Notons la fameuse série des AMD « tri-cœurs », conception permettant de recycler les puces « quadri-cœurs » dont un des cœurs était défectueux et désactivé avant sa mise sur le marché !

Le fait de devoir sortir d'un processeur pour dialoguer avec l'autre est extrêmement coûteux en latence. L'intégration des deux niveaux de cache dans les puces permet de gagner du temps. Le fait de partager le même cache de niveau 2 apporte un réel gain de performance mais nous signale clairement que la synchronisation des accès en mémoire lors de l'utilisation de *threads* est une chose fondamentale. C'est un gain car le fait de devoir faire circuler les synchronisations de cache à l'intérieur du circuit est beaucoup plus rapide que de devoir passer par une architecture extérieure aux processeurs comme c'est le cas dans les architectures « bi-processeurs ». Rappelons que la synchronisation du cache mémoire (qui est une unité très proche du cœur) est fondamentale pour que les lectures dans la mémoire (laquelle a été rapatriée dans le cache de niveau 2) soient correctes.

Le partage des caches est une chose très intéressante en matière de performance. Les architectures parallèles intègrent des mécanismes de *bus snooping* qui vérifient si une donnée présente dans le cache a fait l'objet de modifications de la part d'un processeur. Cela impose aux mécanismes d'écriture de signaler (et donc d'envoyer un message supplémentaire sur le bus système) tous accès à la mémoire. Partager le même cache permet de s'affranchir de cela. C'est une chose que ne pourront jamais réaliser des architectures de type « multi-processeurs ».

Pour analyser dans le détail les performances de vos programmes, vous devriez presque rechercher dans l'ENSTA ParisTech des machines d'architectures différentes afin de trouver la mieux adaptée à, votre programme, votre système d'exploitation...

18

Surveillances des entrées / sorties sous Unix

18.1 Introduction

Quelquefois, il est nécessaire qu'une application soit capable de gérer plusieurs entrées / sorties synchrones simultanément. Par exemple, un serveur sur un réseau doit généralement être capable de gérer des connections simultanément avec plusieurs clients. La communication avec un client réseau s'effectue comme la plupart des entrées / sorties sous Unix, c'est-à-dire via un descripteur de fichier (cf. chapitre 16). Pour le cas du serveur, gérer plusieurs connections revient donc à lire / écrire sur plusieurs descripteurs de fichier à la fois.

Avec les fonctions vues dans les chapitres précédents, deux méthodes sont possibles pour gérer plusieurs entrées / sorties synchrones :

- traiter les descripteurs de fichier dans l'ordre : dans une boucle `for`, chaque descripteur est lu / écrit dans un ordre prédéfini. Le problème de cette méthode est que, comme les entrées / sorties sont synchrones (c'est-à-dire que les méthodes `read` et `write` ne rendent la main que lorsque les données ont été lues / écrites), les données d'un client 2 ne seront traités qu'une fois que le client 1 aura lui même envoyé des données à traiter ;
- créer un processus par descripteur de fichier : bien que la création d'un nouveau processus ne soit pas difficile sous Unix (cf. chapitre 10), gérer plusieurs clients avec cette méthode peut poser des problèmes difficiles de communication, de synchronisation (notamment pour l'accès aux ressources) et s'avérer coûteuse pour le système lorsque de nombreux clients doivent être gérés.

Certains systèmes d'exploitations offrent la possibilité de configurer les entrées / sorties pour qu'elles soient asynchrones (c'est-à-dire pour que les fonctions `read` et `write` soient non bloquantes). Une telle possibilité résout partiellement le problème puisque dans le cas de l'utilisation d'une boucle `for`, il devient alors possible de ne pas rester bloqué sur un client. Cependant, lorsqu'aucun client ne communique, il est quand même nécessaire de vérifier en permanence l'état de la communication avec chaque client en utilisant de façon inutile le processeur alors que le processus devrait être endormi dans l'attente d'une communication.

Enfin, il est possible de gérer plusieurs communications avec plusieurs clients avec des signaux (cf. chapitre 14). Par exemple, le serveur est endormi dans l'attente d'un signal. Lorsqu'un client communique avec le serveur, il envoie les données à communiquer puis réveille le serveur via l'utilisation d'un signal utilisateur. Cependant, cette méthode n'est valable que si les processus se situent tous sur la même machine (ce qui n'est pas le cas d'une architecture client / serveur réseau).

Dans les possibilités que nous venons d'énoncer, aucune ne permet donc vraiment de résoudre parfaitement le problème de plusieurs entrées / sorties synchrones.

18.2 L'appel système `select()`

La fonction `select()` permet de surveiller plusieurs descripteurs de fichier afin de mettre en attente le processus tant que l'état d'au moins un de ces descripteurs n'a pas changé. Il est ensuite possible de savoir quels sont les descripteurs dont l'état a été modifié. Elle permet donc de gérer simultanément plusieurs entrées / sorties synchrones. De plus, elle est extrêmement pratique puisqu'elle peut s'utiliser avec tout type de descripteurs de fichier (socket, tuyaux, entrée standard...).

On commence par indiquer, au moyen d'une variable de type `fd_set`, quels descripteurs nous intéressent :

```
fd_set rset ;

FD_ZERO ( &rset ) ;
FD_SET  ( fileno_client01 , &rset ) ;
FD_SET  ( fileno_client02 , &rset ) ;
...
```

La fonction `FD_ZERO()` met la variable `rset` à zéro, puis on indique l'ensemble des descripteurs de fichier à surveiller avec la fonction `FD_SET()`.

On appelle ensuite la fonction `select()` qui attendra que des données soient disponibles sur l'un de ces descripteurs :

```
if ( select ( fileno_max + 1 , &rset , NULL , NULL , NULL ) == -1 )
{
   /* erreur */
}
```

La fonction `select()` prend cinq arguments :

1. le plus grand numéro de descripteur de fichier à surveiller plus un. Lorsque les descripteurs sont ajoutés dans le tableau `rset`, il est donc nécessaire, en plus, de garder le descripteur de fichier le plus élevé ;
2. un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels on veut lire, il est possible d'utiliser un pointeur nul lorsque cette possibilité n'est pas utilisée ;
3. un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels on veut écrire, il est possible d'utiliser un pointeur nul lorsque cette possibilité n'est pas utilisée ;

4. un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels peuvent arriver des conditions exceptionnelles, il est possible d'utiliser un pointeur nul lorsque cette possibilité n'est pas utilisée ;
5. un pointeur vers une structure de type `timeval` représentant une durée après laquelle la fonction `select()` doit rendre la main si aucun descripteur n'est disponible. Dans ce cas, la valeur retournée par `select()` est 0, il est possible d'utiliser un pointeur nul lorsque cette possibilité n'est pas utilisée.

La fonction `select()` renvoie -1 en cas d'erreur, 0 au bout du temps spécifié par le cinquième paramètre et le nombre de descripteurs prêts sinon.

La fonction `FD_ISSET` permet de déterminer si un descripteur est prêt :

```
if ( FD_ISSET ( fileno_client01 , &rset ) )
{
   /* lire sur le descripteur de fichier correspondant au client 01 */
}

if ( FD_ISSET ( fileno_client02 , &rset ) )
{
   /* lire sur le descripteur de fichier correspondant au client 02 */
}
```

Enfin, il est important de remarquer que les trois ensembles de descripteurs de fichier passés à la fonction `select()` sont modifiés par celle-ci. En effet, lorsque la fonction rend la main, seul les descripteurs dont l'état a changé sont conservés dans les ensembles passés en paramètre. C'est la raison pour laquelle il est nécessaire de reconstruire ces ensembles avant chaque appel.

Plus d'informations sont disponibles concernant la fonction `select()` dans les pages `man` de `select` et `select_tut`.

18.3 Exercices

Question 1

Écrire un programme créant `NB_FILS` processus fils (`NB_FILS` étant une constante définie au moyen du préprocesseur C). Chaque processus fils sera relié au processus père au moyen d'un tuyau, dirigé du fils vers le père. Les processus fils effectueront tous la même action, contenue dans une fonction séparée. Pour ce premier exercice, ils se contenteront d'écrire en boucle infinie leur PID dans le tuyau, en séparant les écritures d'un nombre de secondes choisi au hasard entre 2 et 10 (à cet effet, vous utiliserez la fonction `random()`. Le processus père effectuera une attente passive sur l'ensemble des tuyaux au moyen de la fonction `select()` et affichera les informations lues dès qu'elles seront disponibles.

Question 2

Reprendre le programme précédent en modifiant quelque peu le comportement des processus fils. Ils seront désormais reliés au processus père au moyen de deux tuyaux, un dans chaque sens. Le processus père enverra par ses tuyaux descendants une chaîne

de caractères à chaque processus fils. Chacun d'eux attendra un nombre de secondes compris entre 2 et 10 avant de renvoyer cette chaîne au processus père au moyen du tuyau remontant. Dès réception d'une chaîne, le processus père l'affichera, la renverra au même fils et ainsi de suite.

Question 3

Reprendre le programme précédent en plaçant le code des processus fils dans un exécutable séparé qui sera lancé par `exec()`. la fonction `dup()` permettra aux processus fils de relier les tuyaux à leurs entrée et sortie standard.

18.4 Corrigés

Listing 18.1 – *Corrigé du premier exercice*

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

#define NB_FILS 5

struct identification
{
  pid_t   pid ;
  FILE    *tuyau ;
} ;

struct identification enfant[NB_FILS] ;

void cree_fils ( ) ;
void fils ( FILE *tuyau ) ;

int main ( int argc , char **argv )
{
  int i , plus_grand = 0 ;

   cree_fils ( ) ;

   for (i = 0;i< NB_FILS;i++) {
     if (fileno(enfant[i].tuyau) > plus_grand) {
       plus_grand = fileno(enfant[i].tuyau) ;
     }
   }

   for (;;)  {
      fd_set rset ;

      FD_ZERO ( &rset ) ;
      for (i = 0;i < NB_FILS;i++) {
        FD_SET (fileno(enfant[i].tuyau),&rset);
      }

      if (select(plus_grand+1,&rset,NULL,NULL,NULL) == -1) {
        perror ( "Erreur dans select()" );
        exit ( EXIT_FAILURE );
      }

      for (i = 0;i < NB_FILS;i++) {
        if (FD_ISSET(fileno(enfant[i].tuyau),&rset)) {
          char ligne[100] ;

          if (fgets(ligne,sizeof(ligne),enfant[i].tuyau) == NULL) {
            perror("Erreur dans fgets()");
            exit(EXIT_FAILURE);
            }
            printf("Recu du fils %d : %s",enfant[i].pid,ligne);
         }
      }
```

```
   }
}

void cree_fils()
{
   int i ;

   for(i = 0;i < NB_FILS;i++) {
     int    tuyau[2] ;
     FILE   *ecriture ;
     if (pipe(tuyau) == -1) {
         perror("Erreur dans pipe()");
         exit(EXIT_FAILURE);
     }

     switch (enfant[i].pid = fork()) {
     case -1 :   /* erreur */
       perror("Erreur dans fork()");
       exit(EXIT_FAILURE);
     case 0 :   /* processus fils, ecrivain */
       close (tuyau[LECTURE]) ;
       ecriture = fdopen(tuyau[ECRITURE],"w");
       if (ecriture == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
       }
       fils (ecriture);
       break ;
     default :   /* processus pere, lecteur */
       printf ("Creation du fils %d\n",enfant[i].pid);
       close(tuyau[ECRITURE]);
       enfant[i].tuyau = fdopen(tuyau[LECTURE],"r");
       if (enfant[i].tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
       }
     }
   }
}


void fils ( FILE *tuyau )
{
   pid_t pid = getpid();

   srandom (time(NULL)^pid);

   for (;;) {
     sleep(2+random()%9);

     printf("[fils %d]\n",pid);
     fprintf(tuyau,"%d\n",pid);
     fflush(tuyau);
   }
}
```

Listing 18.2 – *Corrigé du deuxième exercice*

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

#define NB_FILS 5

struct identification
{
   pid_t   pid ;
   FILE    *lecture ;
   FILE    *ecriture ;
} ;

struct identification enfant[NB_FILS] ;

void cree_fils ( ) ;
void fils ( FILE *lecture , FILE *ecriture ) ;


int main ( int argc , char **argv )
{
   int i , plus_grand = 0 ;

   cree_fils();

   for ( i = 0;i < NB_FILS;i++) {
     fprintf(enfant[i].ecriture,"message au fils %d\n",enfant[i].pid);
     fflush(enfant[i].ecriture);
   }

   for (i = 0;i < NB_FILS;i++) {
     if (fileno(enfant[i].lecture) > plus_grand) {
       plus_grand = fileno(enfant[i].lecture);
     }
   }

   for (;;) {
     fd_set rset ;

     FD_ZERO(&rset);
     for (i = 0;i < NB_FILS;i++) {
       FD_SET(fileno(enfant[i].lecture),&rset);
     }

     if (select(plus_grand+1,&rset,NULL,NULL,NULL) == -1) {
       perror("Erreur dans select()");
       exit(EXIT_FAILURE);
     }

     for (i = 0;i < NB_FILS;i++) {
       if (FD_ISSET(fileno(enfant[i].lecture),&rset)) {
         char ligne[100] ;

         if (fgets(ligne,sizeof(ligne),enfant[i].lecture) == NULL) {
           perror("Erreur dans fgets()");
           exit(EXIT_FAILURE);
         }

         printf("Recu du fils %d : %s",enfant[i].pid,ligne);
         fprintf(enfant[i].ecriture,"%s",ligne);
         fflush(enfant[i].ecriture);
       }
     }
```

```
   }
}

void cree_fils()
{
   int i ;

   for (i = 0;i < NB_FILS;i++) {
     int    tuyau1[2] , tuyau2[2] ;
     FILE   *lecture , *ecriture ;

     if (pipe(tuyau1) == -1) {
       perror("Erreur dans pipe()");
       exit(EXIT_FAILURE);
     }

     if (pipe(tuyau2) == -1) {
       perror("Erreur dans pipe()");
       exit(EXIT_FAILURE);
     }

     switch (enfant[i].pid = fork()) {
     case -1 :   /* erreur */
       perror("Erreur dans fork()");
       exit(EXIT_FAILURE);

     case 0 :   /* processus fils */
       close(tuyau1[LECTURE]);
       ecriture = fdopen(tuyau1[ECRITURE],"w");
       if (ecriture == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
       }

       close(tuyau2[ECRITURE]);
       lecture = fdopen(tuyau2[LECTURE],"r");
       if (lecture == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
       }

       fils(lecture,ecriture);
       break;

     default :   /* processus pere */
       printf("Creation du fils %d\n",enfant[i].pid);

       close(tuyau1[ECRITURE]);
       enfant[i].lecture = fdopen(tuyau1[LECTURE],"r");
       if (enfant[i].lecture == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
       }

       close(tuyau2[LECTURE]);
       enfant[i].ecriture = fdopen(tuyau2[ECRITURE],"w");
       if (enfant[i].ecriture == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
       }
     }
   }
}
```

```
void fils ( FILE *lecture , FILE *ecriture )
{
   pid_t pid = getpid ( ) ;

   srandom ( time ( NULL ) ^ pid ) ;

   for ( ; ; )
   {
      char ligne[100] ;

      if ( fgets ( ligne , sizeof ( ligne ) , lecture ) == NULL )
      {
         perror ( "Erreur dans fgets()" ) ;
         exit ( EXIT_FAILURE ) ;
      }

      printf ( "[fils %d] lu du pere : %s" , pid , ligne ) ;

      sleep ( 2 + random ( ) % 9 ) ;

      printf ( "[fils %d] envoi au pere : %s" , pid , ligne ) ;
      fprintf ( ecriture , "%s" , ligne ) ;
      fflush ( ecriture ) ;
   }
}
```

Listing 18.3 – *Corrigé du troisième exercice*

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

#define NB_FILS 5

struct identification
{
  pid_t   pid ;
  FILE    *lecture ;
  FILE    *ecriture ;
} ;

struct identification enfant[NB_FILS] ;

void cree_fils ( ) ;

int main(int argc,char **argv)
{
  int i , plus_grand = 0 ;

  cree_fils();

  for (i = 0;i < NB_FILS;i++) {
    fprintf(enfant[i].ecriture,"message au fils %d\n",enfant[i].pid);
    fflush(enfant[i].ecriture);
  }

  for (i = 0;i < NB_FILS;i++) {
```

```
    if (fileno(enfant[i].lecture) > plus_grand) {
      plus_grand = fileno(enfant[i].lecture);
    }
  }

  for (;;) {
    fd_set rset ;

    FD_ZERO(&rset);
    for (i = 0;i < NB_FILS;i++) {
      FD_SET(fileno(enfant[i].lecture),&rset);
    }

    if (select(plus_grand+1,&rset,NULL,NULL,NULL) == -1) {
      perror("Erreur dans select()");
      exit(EXIT_FAILURE);
    }

    for (i = 0;i < NB_FILS;i++) {
      if (FD_ISSET(fileno(enfant[i].lecture),&rset)) {
        char ligne[100] ;

        if (fgets(ligne,sizeof(ligne),enfant[i].lecture) == NULL) {
          perror("Erreur dans fgets()");
          exit(EXIT_FAILURE);
        }

        printf("Recu du fils %d : %s",enfant[i].pid,ligne);
        fprintf(enfant[i].ecriture,"%s",ligne);
        fflush(enfant[i].ecriture);
      }
    }
  }
}

void cree_fils ( )
{
  int i ;

  for (i = 0;i < NB_FILS;i++) {
    int    tuyau1[2] , tuyau2[2] ;

    if (pipe(tuyau1) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
    }

    if (pipe(tuyau2) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
    }

    switch (enfant[i].pid = fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);

    case 0 :   /* processus fils */
      close(tuyau1[LECTURE]);
      if (dup2(tuyau1[ECRITURE],STDOUT_FILENO) == -1) {
        perror("Erreur dans dup2()");
        exit(EXIT_FAILURE);
      }
```

```
      close(tuyau2[ECRITURE]);
      if (dup2(tuyau2[LECTURE],STDIN_FILENO) == -1) {
        perror("Erreur dans dup2()");
        exit(EXIT_FAILURE);
      }

      if (execl("select03-fils","select03-fils",NULL) == -1 ) {
        perror("Erreur dans exec()");
        exit(EXIT_FAILURE);
      }

    default :   /* processus pere */
      printf("Creation du fils %d\n",enfant[i].pid);

      close(tuyau1[ECRITURE]);
      enfant[i].lecture = fdopen(tuyau1[LECTURE],"r");
      if (enfant[i].lecture == NULL) {
        perror("Erreur dans fdopen()");
        exit(EXIT_FAILURE);
      }

      close(tuyau2[LECTURE]);
      enfant[i].ecriture = fdopen(tuyau2[ECRITURE],"w");
      if (enfant[i].ecriture == NULL ) {
        perror("Erreur dans fdopen()");
        exit(EXIT_FAILURE);
      }
    }
  }
}
```

Listing 18.4 – *Une autre version du troisième exercice*

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main(int argc,char **argv)
{
  pid_t pid = getpid();

  srandom (time(NULL)^pid);

  for (;;) {
    char ligne[100] ;

    if (fgets(ligne,sizeof(ligne),stdin) == NULL) {
      perror("Erreur dans fgets()");
      exit(EXIT_FAILURE);
    }
    sleep(2+random()%9);
    printf("%s",ligne);
    fflush(stdout);
  }
}
```

19

Utilisation d'un débogueur

19.1 Introduction

Le développement en informatique fonctionne principalement par tentatives / échecs. Ainsi, le cycle usuel réalisé par un développeur pour concevoir un programme est composé des étapes suivantes :

1. développement d'une nouvelle fonctionnalité ;
2. test de cette fonctionnalité :
   - si les tests sont validés alors retour à l'étape 1 ;
   - sinon correction des erreurs puis retour à l'étape 2.

La correction d'erreurs dans un logiciel fait donc généralement partie intégrante de son cycle de développement et de maintenance. Une erreur en informatique est appelée *bug*, mot anglais faisant référence aux insectes qui se logeaient entre les contacteurs des tous premiers ordinateurs à lampes et créaient ainsi des dysfonctionnements [1].

Les bugs (ou bogues, selon la terminologie officielle française) étant quasiment inévitables en informatique, la qualité d'un développeur se manifeste certes par la qualité du code écrit mais aussi par sa capacité à détecter et corriger des bugs. Dans ce but, il existe des outils, appelés débogueurs (*debugger*), facilitant la tâche. Un débogueur est une application permettant de contrôler l'exécution d'un programme.

Ce chapitre indique comment utiliser concrètement un débogueur et il est fortement conseillé de prendre le temps de lire les quelques pages qui suivent : la maîtrise d'un débogueur permet de faire gagner beaucoup de temps dans la mise au point des programmes !

Rappelons néanmoins que l'utilisation d'un débogueur est l'étape ultime pour tester du code qui ne fonctionne pas et que cet outil ne dispense pas d'écrire du code propre, par petits ajouts, testé régulièrement.

Le débogueur aide le bon programmeur, mais ne le remplace pas !

1. Cette origine est parfois contestée et des références au mot *bug* auraient été trouvées bien avant cette époque, dans le milieu de l'édition et de l'impression. On peut aisément imaginer que l'idée est la même : un insecte qui vient se glisser sous la presse Gutenberg au moment où celle-ci est sur le point d'imprégner d'encre une feuille de papier.

19.2 gdb, xxgdb, ddd et les autres

L'utilisateur du débogueur peut, par exemple :
- interrompre un programme en cours d'exécution et regarder l'état courant des variables de ce programme ;
- exécuter pas à pas tout ou partie du programme (instructions ou fonctions) ;
- surveiller le contenu de la mémoire ou des registres du microprocesseur ;
- évaluer des expressions de ce programme.

Par la suite, nous détaillerons l'utilisation d'un débogueur appelé `gdb` [2] développé et maintenu par la communauté GNU [3]. `gdb` est un logiciel libre disponible sous Unix et Windows.

`gdb` s'utilise « en ligne de commande », sans interface graphique, et c'est cette utilisation que nous allons détailler dans les sections suivantes : l'utilisation en ligne de commande permet de bien comprendre le fonctionnement d'un débogueur et les apports de celui-ci dans la mise au point de code. Il existe néanmoins différents surcouches graphiques pour **`gdb`**, comme par exemple **`xxgdb`** ou **`ddd`**, et un utilisateur pressé pourra utiliser ces verions plus accessibles de prime abord. Il faut cependant retenir que ce ne sont que des surcouches, qui se contentent d'interpréter des actions à la souris (clic sur un bouton, sélection d'une zone, etc.) pour les envoyer à **`gdb`**

Il existe d'autres débogueurs, commerciaux ou non, dont certains sont intégrés dans des environnement complet de programmation (`IDE` ou *integrated development environment*). L'expérience montre que les développeurs utilisant des IDE ont tendance à méconnaître le fonctionnement réel des compilateurs, débogueurs et autres outils de développement, pour se reposer exclusivement sur leur unique IDE. Ce constat est particulièrement flagrant pour les développeurs ayant appris à programmer directement sous un IDE et, par voie de conséquence, les auteurs du présent document conseillent vivement l'apprentissage indépendant des différents outils de développement.

C'est pourquoi le fonctionnement et l'utilisation de **`gcc`**, **`gdb`**, **`make`**, etc. sont présentés séparément dans ce document, alors qu'il aurait été possible de proposer une vision intégrée, par exemple au travers de l'IDE **`eclipse`** (voir `www.eclipse.org`).

19.3 Utilisation de gdb

Afin d'illustrer le fonctionnement de **`gdb`**, l'exemple suivant sera utilisé :

Listing 19.1 – *Le programme de test*

```
#include <stdio.h>
#include <stdlib.h>

int somme(int a, int b) {
  return a + b;
}
```

2. `http://www.gnu.org/software/gdb/gdb.html`
3. `http://www.gnu.org`

```
int main(int argc, char *argv[]) {
  int i, s;
  s=0;
  for (i=1; i<argc; i++) {
    s = somme(s,atoi(argv[i]));
  }
  printf("la somme est: %d\n", s);
  exit(EXIT_SUCCESS);
}
```

Ce programme calcule la somme des arguments passés lors du lancement du programme.

Compilation

gdb est capable de déboguer n'importe quelle application, même si celle-ci n'a pas été prévu pour. Cependant, pour que les informations affichées par **gdb** soit compréhensibles et puissent se rattacher aux fichiers sources qui ont été compilés, il est nécessaire de spécifier au compilateur de générer des informations de débogage dans les fichiers objets produits. Lors de l'exécution du programme, le débogueur peut ainsi connaître le nom des variables, la ligne correspondante de l'instruction en cours dans le fichier source, ... Avec **gcc**, les informations de débogage s'ajoutent avec l'option `-g`, par exemple :

```
localhost> gcc -g -Wall ./gdb1.c -o ./gdb1
```

gcc permet de générer des informations de débogage spécifiques à **gdb**. Pour cela, il faut utiliser l'option `-ggdb` plutôt que `-g`, par exemple :

```
localhost> gcc -ggdb -Wall ./gdb1.c -o ./gdb1
```

Évidemment, plus le compilateur ajoute des informations de débogage aux fichiers compilés, plus les fichiers ainsi générés sont grands. Afin d'enlever des informations de débogage à un fichier exécutable (bibliothèque ou application), la commande `strip` peut être utilisée.

> Attention, **gcc** est l'un des rares compilateurs permettant d'ajouter des informations de débogage en générant du code optimisé (option `-O`). Il est **déconseillé** de déboguer du code optimisé par le compilateur. En effet, pendant la phase d'optimisation, le compilateur a pu supprimer des variables, remplacer des instructions, rendant le code incohérent avec le source qui l'a produit.

Exécution

Pour lancer **gdb**, il suffit de taper la commande :

```
localhost> gdb
Current directory is /home/degris/Poly/TP/Debugger/
GNU gdb 6.2-2mdk (Mandrakelinux)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu".
(gdb)
```

Un message indiquant le répertoire courant et la version courante de **gdb** s'affiche alors, indiquant que **gdb** est prêt à être utilisé.

> Astuce : il est possible de lancer **gdb** directement depuis `Emacs` en faisant `M-X gdb`. **gdb** se manipule alors exactement de la même façon que depuis la console. L'utilisation du débogueur depuis l'éditeur permet notamment un accès facilité à certaines commandes du débogueur par l'intermédiaire des menus déroulants de l'éditeur. Ces commandes restent naturellement accessibles directement depuis la ligne de commandes !

Avant de commencer l'exécution d'un programme, il est nécessaire de charger celui-ci depuis **gdb**. Ceci est réalisé par la commande `file`, par exemple pour charger le programme `gdb1` :

```
(gdb) cd /local/tmp/degris/gdb
Working directory /local/tmp/degris/gdb.
(gdb) file gdb1
Reading symbols from /local/tmp/degris/gdb/gdb1...done.
Using host libthread_db library "/lib/tls/libthread_db.so.1".
```

Attention, si les informations de débogage n'ont pas été générées, **gdb** n'affiche pas systématiquement un message d'alerte.

Une fois le programme chargé, il peut être exécuté par la commande `run` avec la liste des arguments à passer aux programmes, par exemple :

```
(gdb) run 1 2
Starting program: /local/tmp/degris/gdb/gdb1 1 2
la somme est: 3

Program exited normally.
```

gdb lance alors l'exécution du programme qui se termine normalement si aucun événement (par exemple un signal en provenance du système) n'est survenu. Une fois l'exécution terminée, **gdb** affiche un message le confirmant. Si une valeur autre que 0 est passée à la fonction `exit()` lors de la fin de l'exécution du programme, alors **gdb** affiche cette valeur de retour.

Contrôle de l'exécution

Souvent, il est nécessaire d'interrompre l'exécution d'un programme afin de pouvoir suivre pas à pas son déroulement. Ceci est réalisé avec les points d'arrêt. Par exemple,

il est possible d'associer un point d'arrêt à la fonction `somme`. Dans ce cas, **gdb** interrompra l'exécution du programme à chaque appel de la fonction :

```
(gdb) break somme
Breakpoint 1 at 0x80483df: file /home/degris/Poly/Src/gdb1.c, line 5.
(gdb) run 1 2
Starting program: /local/tmp/degris/gdb/gdb1 1 2

Breakpoint 1, somme (a=0, b=1) at /home/degris/Poly/Src/gdb1.c:5
5         return a + b;
(gdb)
```

gdb nous signale alors qu'il a suspendu l'exécution du programme à l'instruction correspondant à la ligne 5 dans le fichier source `gdb1.c`. De plus, il nous signale que la fonction `somme` a été appelée avec pour valeurs `a=0` et `b=1`. Enfin, **gdb** affiche la ligne correspondant à la prochaine instruction à exécuter. Pour les (heureux) utilisateurs de **gdb** depuis Emacs, cette ligne est remplacée par un pointeur affiché directement dans une fenêtre affichant le code en cours de débogage, rendant l'utilisation de **gdb** beaucoup plus intuitive.

Si l'on continue l'exécution du programme avec la commande `cont`, **gdb** arrêtera l'exécution lors du deuxième appel de la fonction `somme` :

```
(gdb) cont
Continuing.

Breakpoint 1, somme (a=1, b=2) at /home/degris/Poly/Src/gdb1.c:5
5         return a + b;
```

Il est possible d'enlever ce point d'arrêt avec la commande `delete` en passant en argument le numéro associé au point d'arrêt. D'autre part, avec la même commande `break`, il est possible de définir un point d'arrêt avec le nom du fichier source et un numéro de ligne :

```
(gdb) delete 1
(gdb) break gdb1.c:15
Breakpoint 2 at 0x804844a: file /home/degris/Poly/Src/gdb1.c, line 15.
(gdb) cont
Continuing.

Breakpoint 2, main (argc=3, argv=0xbfffee54) at /home/degris/Poly/Src/gdb1.c:15
15        printf("la somme est: %d\n", s);
```

L'exécution du programme est alors suspendue à l'instruction correspondant à la ligne 15 du fichier `gdb1.c`. Lorsque l'exécution est suspendue, il est parfois utile d'exécuter la suite du programme pas à pas. La commande `next` permet de poursuivre l'exécution ligne par ligne. La commande `step` poursuit l'exécution ligne par ligne mais en entrant dans les fonctions si une fonction est appelée, par exemple :

```
Breakpoint 3, main (argc=3, argv=0xbfffee54) at /home/degris/Poly/Src/gdb1.c:12
12        for (i=1; i<argc; i++) {
(gdb) next
13          s = somme(s,atoi(argv[i]));
```

```
(gdb) next
12        for (i=1; i<argc; i++) {
(gdb) step
13          s = somme(s,atoi(argv[i]));
(gdb) step
somme (a=1, b=2) at /home/degris/Poly/Src/gdb1.c:5
5         return a + b;
(gdb)
```

De plus, lorsque le programme est suspendu, il est possible de changer l'adresse du pointeur d'instruction avec la commande `jump`. En effet, cette fonction permet de spécifier la prochaine commande à exécuter. Elle prend en argument une adresse ou un numéro de ligne correspondant à un fichier source. Par exemple, si le programme est suspendu juste avant de se terminer :

```
(gdb) break gdb1.c:16
Breakpoint 1 at 0x804845d: file /home/degris/Poly/Src/gdb1.c, line 16.
(gdb) run 2 3 4
Starting program: /local/tmp/degris/gdb/gdb1 2 3 4
la somme est: 9

Breakpoint 1, main (argc=4, argv=0xbfffee54) at /home/degris/Poly/Src/gdb1.c:16
16        exit(EXIT_SUCCESS);
```

La somme vaut actuellement 9. Si l'on recommence l'exécution de la boucle `for` :

```
(gdb) jump gdb1.c:12
Continuing at 0x804840a.
la somme est: 18

Breakpoint 1, main (argc=4, argv=0xbfffee54) at /home/degris/Poly/Src/gdb1.c:16
16        exit(EXIT_SUCCESS);
```

La somme vaut maintenant 18 puisque la boucle a été exécutée deux fois.

Évaluation d'expressions

La commande `print` permet d'évaluer une expression spécifiée en argument dans le langage débogué (dans notre cas, la langage C). L'évaluation d'une expression peut être notamment utilisée pour connaître le contenu d'une variable. Dans ce cas, seules les variables locales au contexte courant peuvent être utilisées dans une expression. Par exemple, la variable `s` est locale au contexte d'appels de la fonction `main`. Elle n'est donc pas accessible directement si l'exécution est suspendue dans la fonction `somme`. Cependant, les commandes `up` et `down` permettent de changer le contexte courant dans `gdb` et la commande `bt` permet d'afficher la pile d'appels. Par exemple, si le programme est suspendu dans la fonction `somme` :

```
#0  somme (a=1, b=2) at /home/degris/Poly/Src/gdb1.c:5
5         return a + b;
```

Affichage de la pile d'appels :

```
(gdb) bt
#0  somme (a=1, b=2) at /home/degris/Poly/Src/gdb1.c:5
#1  0x0804843d in main (argc=3, argv=0xbfffee54)
    at /home/degris/Poly/Src/gdb1.c:13
```

gdb affiche une erreur concernant l'affichage de la valeur de s puisque cette variable n'est pas visible depuis le contexte de la fonction somme :

```
(gdb) print s
No symbol "s" in current context.
```

On se déplace donc dans la pile d'appels pour trouver un contexte où la variable s est visible (fonction main). Une fois le contexte correct, la valeur de la variable est affichée :

```
(gdb) up
#1  0x0804843d in main (argc=3, argv=0xbfffee54)
    at /home/degris/Poly/Src/gdb1.c:13
13          s = somme(s,atoi(argv[i]));
(gdb) print s
$1 = 1
```

Cependant, les variables qui étaient visibles dans le contexte de la fonction somme (par exemple, l'argument a) ne le sont plus dans ce contexte. Il faut donc se déplacer dans la pile d'appels une nouvelle fois pour accéder au contexte de la fonction somme où l'argument a est visible :

```
(gdb) print a
No symbol "a" in current context.
(gdb) down
#0  somme (a=1, b=2) at /home/degris/Poly/Src/gdb1.c:5
5         return a + b;
(gdb) print a
$2 = 1
```

Lorsque l'exécution est suspendue, il est possible d'évaluer des expressions faisant appel à des fonctions avec la commande print, par exemple :

```
(gdb) print somme(34,56)
$3 = 90
(gdb) print getpid()
$4 = 6018
```

Une expression peut aussi être utilisée pour modifier le contenu d'une variable pendant l'exécution :

```
(gdb) print s
$2 = 18
(gdb) print s=4
$3 = 4
(gdb) print s
$4 = 4
```

La commande `display` permet d'évaluer une expression à chaque pas d'exécution, par exemple :

```
(gdb) break gdb1.c:13
Breakpoint 1 at 0x8048419: file /home/degris/Poly/Src/gdb1.c, line 13.
(gdb) run 1 2 3 4 5
Starting program: /local/tmp/degris/gdb/gdb1 1 2 3 4 5

Breakpoint 1, main (argc=6, argv=0xbfffee44) at /home/degris/Poly/Src/gdb1.c:13
13          s = somme(s,atoi(argv[i]));
(gdb) display s
1: s = 0
(gdb) cont
Continuing.

Breakpoint 1, main (argc=6, argv=0xbfffee44) at /home/degris/Poly/Src/gdb1.c:13
13          s = somme(s,atoi(argv[i]));
1: s = 1
(gdb) cont
Continuing.

Breakpoint 1, main (argc=6, argv=0xbfffee44) at /home/degris/Poly/Src/gdb1.c:13
13          s = somme(s,atoi(argv[i]));
1: s = 3
(gdb)
```

Gestion des signaux

Lorsqu'un signal est envoyé par le système à l'application, celui-ci est intercepté par `gdb`. Le comportement de `gdb` est alors défini par trois paramètres booléens lorsqu'un signal est reçu :

- l'application est suspendue ou non ;
- `gdb` écrit un message lors de la réception ou non ;
- le signal est passé à l'application ou non.

Il est possible de connaître l'état courant de ces paramètres avec la commande `info signals` :

```
(gdb) info signals
Signal        Stop      Print   Pass to program Description

SIGHUP        Yes       Yes     Yes             Hangup
SIGINT        Yes       Yes     No              Interrupt
SIGQUIT       Yes       Yes     Yes             Quit
SIGILL        Yes       Yes     Yes             Illegal instruction
SIGTRAP       Yes       Yes     No              Trace/breakpoint trap
SIGABRT       Yes       Yes     Yes             Aborted
SIGEMT        Yes       Yes     Yes             Emulation trap
SIGSEGV       Yes       Yes     Yes             Segmentation fault
...
```

Par exemple, on peut constater que le signal SIGINT (le signal envoyé lorsque `ctrl-c` est utilisé) n'est pas envoyé au programme. En effet, il est utilisé par `gdb` pour suspendre l'exécution du programme en cours.

Concernant le signal d'erreur de segmentation (`SIGSEGV`), on peut constater que `gdb` suspend le programme, affiche la réception du signal et envoie le signal à l'application. Ceci est très utile pour déboguer une application recevant un tel signal puisque `gdb` suspend l'application lors de la réception du signal.

Le programme suivant (`gdb2.c`) est utilisé pour illustrer comment `gdb` peut faciliter la recherche des causes d'une erreur de segmentation (par exemple) :

Listing 19.2 – *Recherche d'une violation mémoire*

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  char* s = NULL;
  // Erreur de segmentation puisque s pointe sur 0
  sprintf(s, " ");
  exit(EXIT_SUCCESS);
}
```

Lorsque le programme est exécuté, il est interrompu par `gdb` lors de la réception du signal :

```
(gdb) file gdb2
Reading symbols from /local/tmp/degris/gdb/gdb2...done.
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) run
Starting program: /local/tmp/degris/gdb/gdb2

Program received signal SIGSEGV, Segmentation fault.
0x4008d025 in _IO_str_overflow () from /lib/tls/libc.so.6
```

Le signal a été reçu lors de l'appel dans une bibliothèque. Afin de pouvoir accéder au contenu des variables de notre application, il est nécessaire de remonter la pile d'appels jusqu'au contexte qui nous intéresse :

```
(gdb) bt
#0  0x4008d025 in _IO_str_overflow () from /lib/tls/libc.so.6
#1  0x4008bcdf in _IO_default_xsputn () from /lib/tls/libc.so.6
#2  0x40066b17 in vfprintf () from /lib/tls/libc.so.6
#3  0x40081afb in vsprintf () from /lib/tls/libc.so.6
#4  0x4006f5db in sprintf () from /lib/tls/libc.so.6
#5  0x080483df in main (argc=1, argv=0xbfffee64)
    at /home/degris/enseignement/ensta2007/Poly/Src/gdb2.c:7
(gdb) up 5
#5  0x080483df in main (argc=1, argv=0xbfffee64)
    at /home/degris/enseignement/ensta2007/Poly/Src/gdb2.c:7
7         sprintf(s, " ");
```

Maintenant, le pointeur `s` est devenu visible, on peut vérifier sa valeur, cause de l'erreur de segmentation :

```
(gdb) print s
$1 = 0x0
```

Gestion de plusieurs processus

Lorsque le programme débogué crée un processus fils via la fonction `fork`, le processus fils est alors exécuté sans débogueur. Si l'utilisateur a paramétré des points d'arrêt dans le code du processus fils, ceux-ci ne seront donc pas pris en compte. L'exemple `extp2_2.c` du chapitre 10 sur les processus est utilisé à titre d'illustration :

Listing 19.3 – *Gestion du père et du fils*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int i;

  printf("[processus %d] je suis avant le fork\n", getpid());
  i = fork();
  printf("[processus %d] je suis apres le fork, il a retourne %d\n",
         getpid(), i);

  exit(EXIT_SUCCESS);
}
```

L'exécutable correspondant à l'exemple est chargé dans `gdb`. Un point d'arrêt est défini sur la ligne 11 (cette ligne est exécutée par le processus père et le processus fils), puis le programme est démarré :

```
(gdb) break extp2_2.c:11
Breakpoint 1 at 0x804844c: file /home/degris/Poly/Src/extp2_2.c, line 11.
(gdb) run
Starting program: /local/tmp/degris/gdb/extp2_2
[processus 27343] je suis avant le fork
Detaching after fork from child process 27346.
[processus 27346] je suis apres le fork, il a retourne 0

Breakpoint 1, main (argc=1, argv=0xbfffee54)
  at /home/degris/enseignement/ensta2007/Poly/Src/extp2_2.c:11
11 printf("[processus %d] je suis apres le fork, il a retourne %d\n", getpid(), i);
(gdb) print getpid()
$1 = 27343
```

On peut remarquer que seul le processus père a été suspendu et que le processus fils a terminé son exécution.

Afin de déboguer le processus fils, la commande `attach` peut être utilisée. Elle permet « d'attacher » `gdb` à n'importe quel processus en cours d'exécution [4]. Le `PID` du processus auquel `gdb` doit s'attacher est passé en paramètre de la commande. De plus, la commande `attach` provoque la fin de l'exécution du programme en cours de débogage. Par conséquent, si le processus père est déjà exécuté dans un premier débogueur, il est nécessaire d'en utiliser un second.

Par exemple, dans le corrigé `tuyau1.c` du chapitre 15, le processus fils ne se termine pas immédiatement puisqu'il attend une entrée sur l'entrée standard. Il est

4. À condition d'avoir des droits d'accès suffisants sur ce processus.

alors possible d'attacher gdb sur le processus fils. Par exemple, si le programme est lancé depuis un premier débogueur :

```
(gdb) run
Starting program: /local/tmp/degris/gdb/tuyau1
Detaching after fork from child process 28806.
Je suis le fils, tapez des phrases svp
```

Malgré le message affiché, gdb est attaché au processus père. Afin de déboguer le processus fils, on utilise un deuxième débogueur que l'on attache au processus fils :

```
localhost> gdb
GNU gdb 6.2-2mdk (Mandrakelinux)
(gdb) attach 28806
Attaching to process 28806
Reading symbols from /local/tmp/degris/gdb/tuyau1...done.
Using host libthread_db library "/lib/tls/libthread_db.so.1".
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe410 in ?? ()
(gdb) print getpid()
$1 = 28806
```

Cette méthode est générale pour déboguer un processus en cours d'exécution.

Utilisation d'un fichier *core*

Lorsqu'un programme est interrompu par une erreur de segmentation, il est possible de générer un fichier core. Ce fichier est utilisé pour enregistrer l'état courant du programme lorsque celui-ci effectue l'erreur de segmentation. Afin d'activer cette fonctionnalité, la commande **ulimit** du *shell* est utilisé. Sous **bash**, **ulimit -c unlimited** active la génération d'un fichier core, **ulimit -c 0** la désactive. Une fois la fonctionnalité activée, un message s'affiche lors de l'erreur de segmentation, indiquant l'écriture du fichier core. Par exemple, si on lance le programme compilé à partir de l'exemple gdb2.c depuis une invite de commande :

```
localhost> ./gdb2
Segmentation fault (core dumped)
```

Le fichier généré s'appelle en général core ou core.<pidduprocessus> et se situe dans le répertoire courant du programme provoquant l'erreur [5]. Sous NetBSD ce fichier est généralement nommé <nomduprogramme>.core. Ce nom est paramétrable à l'aide de la commande **sysctl** :

```
localhost#> sysctl  -w kern.defcorename = %n.core
```

5. À condition d'avoir les droits en écriture sur ce répertoire.

Une fois le fichier généré, il peut être lu par **gdb** avec la commande `core-file` afin de consulter l'état des variables lorsque l'erreur est survenue. Une fois chargé, l'état de la pile d'appels et l'état des variables peut être consulté. En reprenant le même exemple :

```
localhost> ./gdb
$ gdb
(gdb) file gdb2
Reading symbols from /local/tmp/degris/gdb/gdb2...done.
(gdb) core-file core.24513
Core was generated by `./gdb2'.
Program terminated with signal 11, Segmentation fault.

Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x4008d025 in _IO_str_overflow () from /lib/tls/libc.so.6
(gdb) bt
#0  0x4008d025 in _IO_str_overflow () from /lib/tls/libc.so.6
#1  0x4008bcdf in _IO_default_xsputn () from /lib/tls/libc.so.6
#2  0x40066b17 in vfprintf () from /lib/tls/libc.so.6
#3  0x40081afb in vsprintf () from /lib/tls/libc.so.6
#4  0x4006f5db in sprintf () from /lib/tls/libc.so.6
#5  0x080483df in main (argc=1, argv=0xbfffee74)
    at /home/degris/enseignement/ensta2007/Poly/Src/gdb2.c:7
(gdb) up 5
#5  0x080483df in main (argc=1, argv=0xbfffee74)
    at /home/degris/enseignement/ensta2007/Poly/Src/gdb2.c:7
7         sprintf(s, " ");
(gdb) info locals
s = 0x0
```

Enfin, comme c'est réalisé dans l'exemple, il est nécessaire de charger le fichier exécutable avec la commande **`file`** avant de charger le fichier `core`. En effet, le fichier `core` ne contient aucune information concernant les symboles de débogage nécessaire à son interprétation, contrairement au fichier exécutable.

Affichage d'informations sur l'état courant de l'application et du système

Il est possible d'obtenir d'autres informations sur l'état courant du programme avec l'aide de **gdb**, en voici quelques exemples :

– `info all-registers` : affiche la liste de l'état des registres du processeur :

```
(gdb) info all-registers
eax            0xfffffe00       -512
ecx            0x40019000       1073844224
edx            0x400    1024
ebx            0x0      0
esp            0xbfffec44       0xbfffec44
...
```

– `info locals` : affiche l'état des variables locales au contexte courant :

```
(gdb) info locals
tuyau =   {7,
  8}
```

```
mon_tuyau = (FILE *) 0x804a008
ligne =   "\230I..."
```

– `ptype` : affiche la définition d'un type de donnée :

```
(gdb) ptype FILE
type = struct _IO_FILE {
    int _flags;
    char *_IO_read_ptr;
    ...
    int _fileno;
    ...
}
```

D'une manière générale, il est possible d'obtenir une liste de l'ensemble des commandes ainsi que leur descriptif via l'aide de **gdb** accessible avec la commande `help`. De plus, un manuel est disponible à l'adresse suivante : `http://www.gnu.org/manual/`. Enfin, plusieurs interfaces graphiques existent pour faciliter l'utilisation de **gdb**. Nous citerons notamment `ddd` [6] ou `Eclipse` [7] avec son extension `CDT` [8].

6. `http://www.gnu.org/software/ddd/`
7. `http://www.eclipse.org`
8. `http://www.eclipse.org/cdt`.

Troisième partie

Projets

20

Projet de carte bleue

20.1 Introduction

L'objectif du projet proposé est de simuler les échanges entre banques permettant à un particulier de payer ses achats avec sa carte bancaire (sa « carte bleue »), même si celle-ci n'est pas émise par la même banque que celle du vendeur.

Avant de présenter le sujet, examinons le fonctionnement du paiement par carte bancaire.

Le principe du paiement par carte bancaire

Le paiement par carte bancaire met en relation plusieurs acteurs :
- le client, qui souhaite régler un achat avec la carte bancaire qu'il possède et qui lui a été fournie par sa banque (le Crédit Chaton) ;
- le commerçant, qui est équipé d'un terminal de paiement fourni par sa propre banque (la Bénépé) ;
- la banque du commerçant (la Bénépé) à laquelle vient se connecter le terminal de paiement ;
- la banque du client (le Crédit Chaton) qui va dire si la transaction est autorisée (donc si le compte de son client est suffisamment provisionné ou non).

Le terminal du commerçant est relié à la banque Bénépé grâce à une simple ligne téléphonique. La banque Bénépé est connectée à toutes les autres banques installées en France, et notamment au Crédit Chaton, grâce à un réseau dédié : le réseau interbancaire (voir figure 20.1).

Supposons maintenant que le client lambda se rend chez son revendeur de logiciels préféré pour acheter la toute dernière version du système d'exploitation « FENETRES ». Au moment de passer en caisse, il dégaine sa carte bancaire, le caissier l'insère dans son terminal de paiement et le client doit, après avoir regardé au passage la somme qu'il s'apprête à débourser, saisir son code confidentiel.

Ce code est directement vérifié par la carte (plus exactement par la puce contenue dans la carte). Si le code est erroné, la transaction s'arrête là. Si le code est bon, en revanche, les opérations suivantes ont lieu :

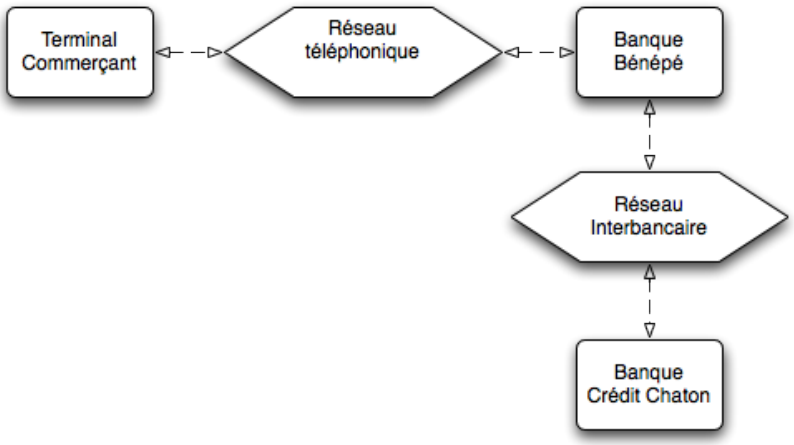


FIGURE 20.1 – *Principe de connexion des terminaux et des banques*

1. Le terminal se connecte (via le réseau téléphonique) au serveur de la banque Bénépé et envoie le numéro de la carte bancaire du client ainsi que le montant de la transaction.
2. Le serveur de la banque Bénépé regarde le numéro de la carte et, se rendant compte qu'il ne s'agit pas d'une des cartes qu'il a émises, envoie le numéro de carte avec le montant de la transaction au serveur de la banque Crédit Chaton, via le réseau interbancaire permettant de relier les différentes banques.
3. Le serveur de la banque Crédit Chaton prend connaissance du numéro de la carte bancaire et vérifie que le compte correspondant à ce numéro dispose d'un solde suffisant pour honorer la transaction.
4. Si c'est le cas, il répond à la banque Bénépé (toujours via le réseau interbancaire) que le paiement est autorisé. Si ce n'est pas le cas, il répond le contraire.
5. Enfin, le serveur de la banque Bénépé transmet la réponse au terminal du commerçant.
6. La transaction est validée (« paiement autorisé ») ou refusée (« paiement non autorisé »).

La demande d'autorisation

La suite des opérations décrites ci-dessus se nomme la « demande d'autorisation » et a essentiellement pour but de vérifier que le compte du client est bien provisionné (ou qu'il a une autorisation de découvert). Cette demande d'autorisation n'est pas

systématique et dépend du terminal [1], lequel prend par exemple en compte le montant de la transaction.

La demande d'autorisation transite via deux serveurs différents :

Le serveur d'acquisition - Il s'agit du serveur de la banque du commerçant auquel se connecte le terminal via le réseau téléphonique. Une fois connecté, le terminal envoie au serveur d'acquisition toutes les informations concernant la transaction, notamment le montant, le numéro de la carte et des données permettant d'assurer la sécurité de la transaction.

Le serveur d'autorisation - Il s'agit du serveur de la banque du client auquel le serveur d'acquisition transmet l'autorisation de paiement émise par le terminal. La réponse à la demande suit le chemin inverse, à savoir serveur d'autorisation de la banque du client $\rightarrow$ serveur d'acquisition de la banque du commerçant $\rightarrow$ terminal du commerçant.

Le routage

Pour effectuer le routage des demandes d'autorisation, c'est-à-dire pour déterminer à quelle banque chaque demande d'autorisation doit être transmise, le serveur d'acquisition utilise les premiers numéros de chaque carte bancaire concernée : ceux-ci indiquent la banque ayant émis cette carte. Dans ce projet, nous partirons des principes suivants :

- un numéro de carte est constitué de seize chiffres décimaux ;
- les quatre premiers correspondent à un code spécifique à chaque banque ;
- les serveurs d'acquisition des banques sont directement reliés au réseau interbancaire.

Chaque serveur d'acquisition analyse donc le numéro de la carte qui figure dans la demande d'autorisation qu'il reçoit, puis :

- si le client est dans la même banque que le commerçant (et que le serveur d'acquisition), il envoie la demande directement au serveur d'autorisation de cette banque ;
- si le client est dans une autre banque, le serveur d'acquisition envoie la demande sur le réseau interbancaire, sans se préoccuper de la suite du transit.

Le réseau interbancaire n'est donc pas un simple réseau physique : il doit aussi effectuer le routage des demandes d'autorisation, c'est-à-dire analyser les demandes qui lui sont fournies, envoyer chaque demande vers le serveur d'autorisation de la banque correspondante et, enfin, prendre en charge la transmission de la réponse lorsqu'elle lui revient.

20.2 Cahier des charges

L'objectif de ce projet est de simuler les mécanismes décrits ci-dessus, c'est-à-dire :

1. Et bientôt aussi des cartes bancaires.

– le terminal envoyant une demande d'autorisation au serveur d'acquisition de sa banque ;
– le serveur d'acquisition effectuant le routage de la transaction vers le bon serveur d'autorisation et effectuant le routage des réponses qu'il reçoit en retour vers les terminaux ;
– le réseau interbancaire auquel sont connectés les différents serveurs d'acquisition, capable d'effectuer le routage des demandes et des réponses relayées par les serveurs d'acquisition ;
– le serveur d'autorisation fournissant la réponse à la demande d'autorisation.

Remarque : cet exercice est avant tout « académique », mais n'est pas dénué d'intérêt ; en effet, des sociétés commercialisent toutes sortes de simulateurs pour tester le fonctionnement des nouveaux composants des systèmes monétiques, afin de valider leur fonctionnement avant leur mise en production.

Le cahier des charges fonctionnel précise l'architecture générale, les fonctionnalités devant être programmées ainsi que les contraintes fonctionnelles à respecter. Le cahier des charges technique fournit les restrictions concernant la mise en œuvre.

Cahier des charges fonctionnelles

Architecture fonctionnelle

Le schéma 20.2 précise celui qui a été présenté plus haut et retranscrit la description que nous avons fournie ci-dessus.

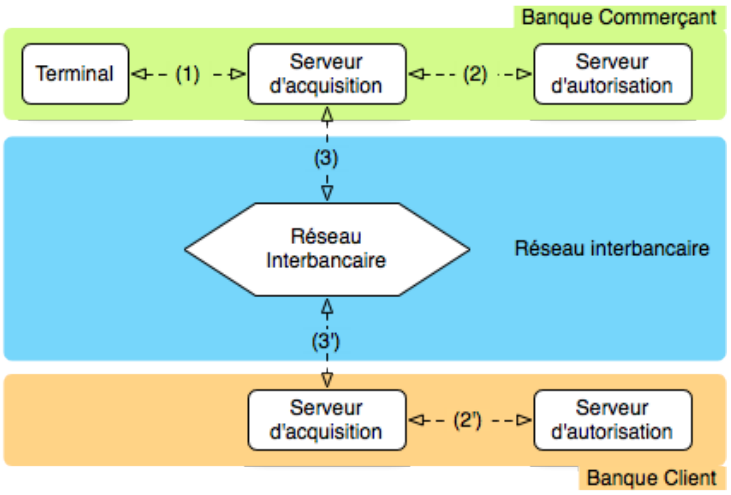


FIGURE 20.2 – *Architecture fonctionnelle du projet.*

Le terminal est relié via le réseau téléphonique (1) au serveur d'acquisition de

la banque du commerçant. Celui-ci est connecté au sein de la banque (2) au serveur d'autorisation de cette même banque.

Le réseau interbancaire relie (3, 3') les serveurs d'acquisition des différentes banques. Toutes les autres banques de la place sont également reliées au réseau interbancaire, mais ne sont pas représentées sur ce schéma.

Le terminal

Dans le cadre de ce projet, il n'est pas question d'utiliser de vrais terminaux ou de vraies cartes. Le terminal étant le moyen d'envoyer aux programmes des demandes d'autorisation, il sera simulé pour ce projet par une interface utilisateur en mode texte permettant simplement de saisir un numéro de carte et un montant de transaction.

Chaque terminal devra envoyer les informations saisies au serveur d'acquisition, devra attendre la réponse du serveur d'acquisition et l'afficher à l'écran (*i.e.* paiement autorisé ou non).

Les échanges entre le terminal et le serveur d'acquisition ont lieu suivant un protocole bien déterminé : les informations sont formatées d'une certaine façon. Ce sont les constructeurs de terminaux qui imposent leurs protocoles et ce sont les serveurs d'acquisition qui doivent s'adapter pour « parler » les protocoles des différents terminaux qui lui sont connectés.

Afin de simplifier le projet et de garantir l'interopérabilité des différents projets en eux (voir en annexe pour une explication de ce point), un seul protocole de communication sera utilisé. Celui-ci est décrit en annexe de ce document.

Le serveur d'acquisition

Le serveur d'acquisition n'a qu'une fonction de routage :

- il doit pouvoir accepter des demandes d'autorisation provenant de terminaux et du réseau interbancaire ;
- il doit pouvoir effectuer le routage des demandes d'autorisation vers le serveur d'autorisation de la banque ou bien vers le réseau interbancaire ;
- il doit pouvoir accepter les réponses provenant du réseau interbancaire ou du serveur d'autorisation de la banque ;
- il doit pouvoir envoyer les réponses vers le réseau interbancaire ou le terminal (en étant capable d'apparier chaque réponse à la demande initiale).

Le serveur d'acquisition doit être capable d'utiliser le protocole de communication employé par les terminaux, ainsi que le protocole du réseau interbancaire et le protocole du serveur d'autorisation (voir les protocoles définis en annexe).

Afin de pouvoir effectuer correctement le routage des messages, le serveur d'acquisition doit connaître les 4 premiers chiffres des numéros des cartes de sa banque.

Le serveur d'autorisation

Le serveur d'autorisation doit être capable de fournir une réponse à une demande d'autorisation. Pour fonctionner, le serveur d'autorisation doit donc avoir accès aux soldes des comptes des clients de la banque référencés par numéro de carte.

Dans le cadre de cette simulation, nous utiliserons une méthode simple : le serveur d'autorisation possède la liste des numéros de cartes émises par la banque, auxquels sont associés les soldes des comptes adossés à ces cartes ; lorsqu'une demande d'autorisation lui parvient, le serveur vérifie que le numéro de carte figure bien dans sa liste. Il contrôle alors que le solde de compte est suffisant pour effectuer la transaction, si c'est le cas il répond oui, sinon, il répond non.

Le réseau interbancaire

Le réseau interbancaire n'a qu'une fonction de routage :
– il doit pouvoir accepter les messages provenant des serveurs d'acquisition ;
– il doit pouvoir analyser le contenu des messages pour déterminer vers quel serveur d'acquisition il doit les envoyer.

Pour fonctionner, le réseau interbancaire doit posséder la liste des codes de 4 chiffres figurant en en-tête des cartes et les banques associées.

Cahier des charges techniques

Contraintes générales

L'ensemble de la simulation fonctionnera sur une seule machine.

Les programmes seront écrits en langage `C`, en mettant en œuvre les solutions et techniques apprises pendant les TD du cours et en proscrivant toute autre méthode.

Nombre de processus

Chaque composant « fonctionnel » tel qu'il a été présenté dans le cahier des charges fonctionnel correspondra à un processus. On trouvera donc un processus pour le terminal (que l'on nommera `Terminal`), un processus pour le serveur d'acquisition (`Acquisition`), un processus pour le serveur d'autorisation (`Autorisation`) et un processus pour le réseau interbancaire (`Interbancaire`).

La simulation pouvant mettre en jeu plusieurs terminaux, plusieurs banques, etc. et on trouvera en fait un processus `Terminal` par terminal, un processus `Acquisition` par serveur d'acquisition et un processus `Autorisation` par serveur d'autorisation.

Paramètres

Les paramètres nécessaires au fonctionnement des différents processus seront fournis via « la ligne de commande », à l'exception des paramètres suivants qui seront fournis sous forme de fichier :

– la liste des banques et de leur code à 4 chiffres ;
– la liste des cartes bancaires de chaque banque et le solde du compte associé.

Ces fichiers seront au format texte, chaque ligne contenant les 2 informations demandées (code sur 4 chiffres et banque associée ou numéro de carte et solde du compte associé), séparées par une espace.

Gestion des échanges par tuyau

Les terminaux connectés au même serveur d'acquisition (donc les terminaux d'une même banque) utiliseront chacun une paire de tuyaux pour dialoguer avec ce serveur. Le serveur d'acquisition aura donc comme rôle d'orchestrer simultanément les lectures et les écritures sur ces tuyaux. La synchronisation (demande, autorisation d'écriture, ordre de lecture) sera mise en œuvre par l'intermédiaire de l'appel système `select()` (voir protocole de communication défini en annexe).

Les échanges entre un serveur d'acquisition et un serveur d'autorisation seront possibles au travers d'une paire de tuyaux, fonctionnant d'une façon très classique, selon le procédé vu en PC.

Pour ce qui concerne le réseau interbancaire et les serveurs d'acquisition, la problématique est strictement identique à celle des terminaux : il faudra utiliser une paire de tuyaux pour connecter chaque serveur d'acquisition au réseau interbancaire. Ceci confère au processus `Interbancaire` un rôle de chef d'orchestre et implique de maintenir une table de routage.

Comment tester sans s'y perdre ?

Le schéma proposé ci-dessus comporte un petit défaut tant que les communications entre processus se feront sur la même machine (donc sans utiliser de *socket*, développement proposé en option) : chaque commerçant (en fait l'utilisateur, donc vous ou bien le MC qui corrigera votre projet) va en pratique saisir ses demandes d'autorisation sur un processus `Terminal`. Or, tous ces processus auront été lancés à partir du même terminal de commande (du même *shell*, de la même fenêtre `xterm`). Nous ne disposerons donc que d'une seule et unique fenêtre (d'un seul terminal de commande) pour simuler plusieurs terminaux bancaires et pour afficher ou saisir les informations de ces terminaux bancaires...

Afin de faire bénéficier à chaque processus `Terminal` d'une entrée et d'une sortie standard qui lui sont propres, l'astuce suivante peut être utilisée : lors de la création d'un nouveau `Terminal` par le processus `Acquisition` [2], recouvrir le nouveau fils du processus `Acquisition` non pas par `Terminal`, mais par **`xterm -e Terminal`** [3].

2. Ou par un ancêtre commun au processus `Acquisition` et à chaque processus `Terminal`.
3. La commande `xterm -e prog` lance une fenêtre **`xterm`** qui exécute en premier lieu l'exécutable **`prog`** ; cette astuce n'est pas très esthétique et n'est qu'un intermédiaire permettant de tester le fonctionnement du projet avant la mise en place des communications par *socket*

Les commerçants pourront ainsi taper des demandes, puis lire les réponses obtenues sans que les affichages des différents terminaux n'interfèrent entre eux.

Livrables

Doivent être livrés pour constituer ce projet :

- un ensemble de fichiers C, amplement commentés, correspondant aux différents programmes constituant la simulation ;
- un fichier `Makefile` permettant de compiler aisément ces programmes afin de créer les exécutables correspondants ;
- des exemples de fichiers de configuration, permettant de faire fonctionner un réseau interbancaire auquel seront connectées deux banques possédant chacune deux terminaux ;
- un mode d'emploi expliquant comment obtenir une application opérationnelle, comment l'exécuter et comment la tester ; ne pas oublier à ce sujet de préciser tous les paramètres à modifier ou à communiquer aux programmes pour le faire fonctionner dans un environnement différent de celui de l'ENSTA ParisTech [4] ;
- un manuel technique détaillant l'architecture, le rôle de chaque composant, expliquant comment tester leur bon fonctionnement de façon indépendante et justifiant les choix techniques effectués ; ce manuel devra en particulier bien préciser comment le projet répond au cahier des charges (ce qu'il sait faire, ce qu'il ne sait pas faire et mettre en exergue ses qualités ainsi que ses défauts).

Tous les documents à produire s'adressent à des ingénieurs généralistes et les rédacteurs doivent donc veiller à donner des explications concises mais claires.

Aucun fichier exécutable, fichier objet ou fichier de sauvegarde ne devra être transmis avec les livrables. Aucun fichier superflu ou inutile ne devra être transmis avec les livrables.

Tous les fichiers constituant les livrables sont au format texte et doivent être lisibles par tout lecteur ou éditeur de texte. Les schémas (et uniquement les schémas) au format `PDF` sont tolérés.

20.3 Conduite du projet

Les étapes qui vous sont suggérées dans cette partie permettent une approche « sereine » de la programmation de la version minimale du projet.

4. Le correcteur n'utilisera en effet probablement pas une station de travail de l'ENSTA ParisTech. Ce n'est pas au correcteur (ou client) de faire des efforts pour comprendre l'intérêt de votre projet (produit), mais à vous de faire des efforts pour le convaincre. Si le correcteur est obligé de chercher longuement comment faire fonctionner votre projet, la note attribuée risque de refléter sa faible conviction.

Introduction

Un projet en plusieurs étapes

Le développement de ce simulateur va être ici présenté en plusieurs étapes qui devraient vous amener vers une application fonctionnelle minimale.

Ce découpage conduit le développeur à écrire des programmes indépendants les uns des autres qui vont communiquer entre eux par l'intermédiaire de tuyaux ou/et *socket*, souvent après redirection des entrées et sorties standard. Ces programmes devront rester assez génériques pour répondre essentiellement à la fonctionnalité pour laquelle ils ont été créés.

Méthode : « divide and conquer »

La constitution de l'application globale par écriture de « petits » programmes indépendants qui communiqueront ensuite entre eux est un gage de réussite : chaque « petit » programme générique peut être écrit, testé et validé indépendamment et la réalisation du projet devient alors simple et très progressive.

La meilleure stratégie à adopter consiste à écrire une version minimale de chaque brique du projet, à faire communiquer ces briques entre elles, puis, éventuellement, à enrichir au fur et à mesure chacune des briques. La stratégie consistant à développer à outrance une des briques pour ensuite s'attaquer tardivement au reste du projet conduit généralement au pire des rapports résultat / travail.

Avant toute programmation, conception ou réalisation de ce projet, il est fortement conseillé de lire l'intégralité de l'énoncé, y compris les parties que vous ne pensez pas réaliser, et de s'astreindre à comprendre la structuration en étapes proposée ici. La traduction de cet énoncé sous forme de schéma est un préalable indispensable.

En particulier, il est particulièrement important de définir dès le départ l'ensemble des flux de données qui vont transiter d'un processus à un autre, de déterminer comment il est possible de s'assurer que ces flux sont bien envoyés et reçus, puis de programmer les fonctions d'émission et de réception correspondantes. Ces fonctions seront ensuite utilisées dans tous les processus du projet.

Un schéma clair et complet associé à des communications correctement gérées garantissent la réussite de ce projet et simplifient grandement son développement ! C'est dit !

Étape 1 : les fonctions de communication

La première étape consiste à programmer les différentes fonctions de communication permettant de lire et écrire des demandes d'autorisation et des réponses selon les protocoles définis en annexe.

Les problèmes de synchronisation seront abordés dans les étapes suivantes et il s'agit pour l'instant uniquement de traduire sous forme de programmes (de fonctions

en `C`) les protocoles de communication : formatage des messages selon la structure définie en annexe, récupération des informations stockées sous cette forme, etc.

Une fois ces fonctions écrites, elles seront utilisées par tous les processus constituant le projet.

Étape 2 : réalisation de programmes indépendants

Dans cette seconde étape, il s'agit de mettre en place les différentes briques du projet et de pouvoir les tester indépendamment.

Processus : `Autorisation`

Le processus `Autorisation` recevra sur son entrée standard des demandes d'autorisation auxquels il devra répondre sur sa sortie standard.

L'exécutable `Autorisation` prendra sur sa ligne de commande le nom du fichier de paramètres, c'est-à-dire celui comprenant la liste des soldes des comptes des clients de la banque, référencés par leurs numéros de cartes.

Le processus `Autorisation` pourra donc être testé indépendamment des autres processus.

Processus : `Acquisition`

Le processus `Acquisition` recevra des messages de demande d'autorisation en provenance soit des terminaux, soit du réseau interbancaire, ou bien des réponses en provenance du serveur d'autorisation ou du réseau interbancaire. Ces messages seront redirigés (« routés ») vers le réseau interbancaire, vers le serveur d'acquisition ou bien vers les terminaux, conformément au cahier des charges. À cette phase du projet on pourra utiliser :

- l'entrée et la sortie standard pour simuler le dialogue avec le serveur d'autorisation (à la main) ;
- des fichiers dans lesquels le processus `Acquisition` ira écrire lorsqu'il est supposé envoyer un message ;
- des fichiers dans lesquels le processus `Acquisition` ira lire les messages qu'il s'attend à recevoir ; ces derniers seront préparés à la main.

Ces fichiers permettront de simuler les échanges avec les processus `Terminal` et `Interbancaire`.

Le programme devra accepter sur sa ligne de commande au moins un paramètre représentant les 4 premiers chiffres des cartes de la banque à laquelle il appartient.

Processus : `Terminal`

Le processus `Terminal` offrira à l'utilisateur l'interface définitive permettant la saisie des informations pour une transaction (montant et numéro de carte).

Afin de pouvoir tester le fonctionnement du `Terminal`, les messages de demande d'information à destination du serveur d'acquisition pourront être écrits dans un fichier. Les réponses seront également lues dans un fichier qui aura été préparé à l'avance.

Etape 3 : création du réseau bancaire

Préparation de la communication par tuyaux

En pratique, le processus `Acquisition` et chaque processus `Terminal` vont communiquer via une paire unique de tuyaux. Une fois ces 2 tuyaux créés, il est possible de modifier simplement les programme `Terminal` et `Acquisition` pour qu'ils utilisent non pas des fichiers mais ces 2 tuyaux.

Pour cela, on complétera le programme `Terminal` en permettant de lui passer sur la ligne de commande les descripteurs des tuyaux à utiliser en lecture et en écriture. On modifiera le code pour que ces tuyaux soit utilisés en lieu et place des fichiers employés à la seconde étape.

Raccordement

Pour terminer la mise en place des communications au sein d'une même banque, il faut maintenant créer, d'une part, une paire de tuyaux entre `Acquisition` et chaque `Terminal` (il y aura autant de paires de tuyaux que de terminaux), d'autre part, une paire de tuyaux entre `Acquisition` et `Autorisation`, après avoir redirigé leurs entrées et sorties standard.

Écrire un programme `Banque` acceptant sur sa ligne de commande le nombre de terminaux de la banque.

Le programme devra également accepter sur sa ligne de commande les paramètres suivants :

- le nom de la banque à simuler ;
- les 4 chiffres associés à cette banque ;
- le nom d'un fichier contenant les soldes des comptes des clients ;
- le nombre de terminaux à créer.

Créer les tuyaux nécessaires, opérer les clonages et recouvrement nécessaire pour créer les processus `Terminal` et `Autorisation` en nombre suffisant.

Enfin recouvrir `Banque` par le programme `Acquisition` sans oublier d'effectuer les redirections nécessaires des tuyaux avec les entrées et sorties standards.

Etape 4 : création du réseau interbancaire

Processus : `Interbancaire`

Chaque serveur d'acquisition sera relié au processus `Interbancaire` par une paire de tuyaux. Celle-ci permettra à `Interbancaire` de recevoir les messages de demande d'autorisation et de transmettre les réponses en retour, après les avoir routés.

L'architecture à mettre en place entre `Interbancaire` et les différents processus `Acquisition` sera similaire à celle mise en place entre chaque processus `Acquisition` et les processus `Terminal` qui y sont reliés.

Dans un premier temps, les messages transitant par `Interbancaire` seront simplement lus sur l'entrée standard et écrits sur la sortie standard (ou lus et écrits dans des fichiers, sans qu'aucune communication inter-processus ne soit mise en place.

Raccordement

On créera un programme `Démarrage` qui devra accepter sur sa ligne de commande un fichier de configuration dans lequel on trouvera les informations suivantes :

- le nom d'un fichier contenant toutes les banques et les 4 chiffres associés à chaque banque ;
- les noms des fichiers nécessaires au fonctionnement de chaque banque ;
- le nombre de terminaux pour chaque banque.

Le processus `Démarrage` devra créer les tuyaux, effectuer les clônages et les recouvrements nécessaires afin de créer l'ensemble des processus nécessaires à la simulation.

20.4 Évolutions complémentaires et optionnelles

Multi-demandes

Dans cette version les processus `Acquisition` et `Interbancaire` doivent pouvoir gérer plusieurs demandes en parallèle. Il faudra donc utiliser toutes les possibilités d'identification des messages proposés par les protocoles.

Cumul des transactions

On demande de rendre la simulation plus réaliste en permettant au niveau du serveur d'autorisation de comptabiliser l'ensemble des paiements effectués par une carte afin de proposer une réponse à la demande d'autorisation qui tienne compte du solde du compte, mais également de l'encours carte.

Délai d'annulation (*time out*)

Dans cette version les processus `Acquisition` et `Interbancaire` doivent pouvoir gérer une fonction d'annulation : une transaction est annulée lorsqu'il s'est écoulé plus de 2 secondes depuis le relais de la demande, sans qu'une réponse ne soit parvenue. Si une réponse parvient ultérieurement, elle sera ignorée.

Ceci suppose que les processus `Acquisition` et `Interbancaire` conservent une trace datée de toutes les demandes qu'ils traitent et qu'ils vérifient régulièrement la date de péremption de chaque demande.

Attention : l'annulation d'une demande pour délai trop important ne dispense pas les processus `Acquisition` et `Interbancaire` d'envoyer une réponse à l'émetteur de la demande.

Utilisation des *sockets*

L'utilisation des communications entre machines ne fait pas partie des objectifs de ce cours. Cependant, afin de rendre le projet plus vivant et plus attractif, nous proposons aux plus débrouillards d'utiliser des *sockets* de sorte que la simulation soit plus réaliste :

- chaque terminal communique avec son serveur d'acquisition par des *sockets* ;
- chaque serveur d'acquisition communique avec le réseau interbancaire par *sockets*.

Les informations nécessaires à l'utilisation des *sockets* sont données dans le polycopié. Une introduction aux *sockets* est faite en annexe.

Le travail de mise en œuvre des communications par *socket* ne doit être entrepris **seulement après avoir réussi à programmer une application complètement opérationnelle avec des tuyaux**.

Attention : **ne jamais modifier un programme qui fonctionne et toujours travailler sur une copie de celui-ci**.

20.5 Annexes

De l'intérêt des standards pour assurer une meilleure interopérabilité

Le recours à des protocoles « standardisés » (ou pour le moins communs) a un double intérêt :

- il permet de gagner du temps sur le développement de ce projet et d'illustrer par la pratique la façon dont les problèmes sont traditionnellement résolus en informatique ;
- il permettra de mélanger les projets entre eux afin de tester le respect du protocole et, peut-être, d'aider certains binômes à avancer (il suffira d'utiliser les processus d'un autre binôme [5]).

La capacité ainsi esquissée à mélanger des composants issus de programmeurs ou prestataires différents pour constituer un système d'information unique se nomme « l'interopérabilité » et est un des enjeux majeurs de l'informatique actuelle.

5. Précisons tout de suite que cette utilisation ne peut s'entendre qu'à des fins de mise au point...

Spécifications des protocoles de communication

Les messages sont constitués de caractères `ASCII`[6]. Chaque message est constitué de différents champs séparés par les caractères `'|'` et terminés par un caractère de fin de message `'\n'`. Le premier caractère indique toujours la nature du message.

Le récapitulatif des échanges est présenté sur la figure 20.3.

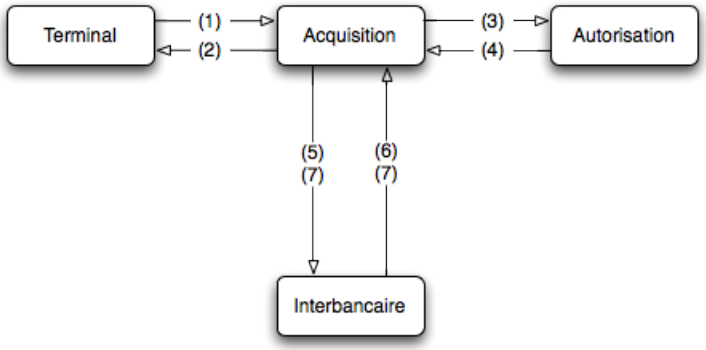


FIGURE 20.3 – *Récapitulatif des échanges.*

Remarque : afin de simplifier au maximum le protocole, on considère qu'il n'est pas possible que plus d'une demande d'autorisation ou réponse, correspondant à une transaction faite avec une carte donnée, circule sur le réseau. Cette hypothèse (plutôt réaliste) permettra de simplifier l'appariement des demandes et des réponses au niveau des processus `Acquisition` et `Interbancaire`.

Un format de message unique

Quel que soit le message, son format est toujours le même :

`|I...I|C...C|A...A|B...B|\n` avec :

- `I...I` est l'identifiant de l'envoyeur du message (longueur strictement supérieur à 0 et indéterminée a priori) ;
- `C...C` est le nom de la commande décrite dans le message (longueur strictement supérieur à 0 et indéterminée a priori) ;

6. L'utilisation de caractères accentués dans les échanges entre processus et dans les fichiers de configuration est strictement déconseillée. En fonction des paramétrages des stations de travail, l'utilisation de ce type de caractères peut entraîner des complications et imposer une programmation en C plus complexe. Rappelons que le cours IN201 n'est pas un cours de programmation en C et que l'important est de réussir à mettre en oeuvre des échanges structurés entre processus.

– `A...A` est le premier argument (optionnel) de la commande (longueur indéterminée a priori) ;
– `B...B` est le deuxième argument (optionnel) de la commande (longueur indéterminée a priori).

Ce format devrait permettre de traiter l'ensemble des cas possibles. Afin de simplifier la mise en œuvre, on supposera qu'un message comporte au plus 255 caractères.

La suite des spécifications du protocole consiste à définir l'utilisation de ce format spécifiquement au contexte des communications énumérées dans le schéma récapitulatif ci-dessus.

Protocole du terminal

Les échanges avec le terminal utilisent deux types de message, *la demande d'autorisation* et *la réponse à la demande d'autorisation*.

(1) Message de demande d'autorisation

Ce message est envoyé par le terminal et réceptionné par le serveur d'acquisition.

Format : `|I...I|DemandeAutorisation|A...A|B...B|\n`

Chaque lettre représente un caractère du message.

– `I...I` est l'identifiant du terminal (par exemple son PID) ;
– `A...A` correspond au montant de la transaction exprimé en centimes d'euros ;
– `B...B` correspond au numéro de la carte sur 16 chiffres décimaux (longueur constante).

(2) Message de réponse à la demande d'autorisation

Ce message est envoyé par le serveur d'acquisition et réceptionné par le terminal.

Format : `|I...I|ReponseAutorisation|A...A||\n` avec :

– `I...I` est l'identifiant du serveur d'acquisition (peut être son PID) ;
– `A...A` est la chaîne de caractères :
  – `oui` si la demande est acceptée,
  – `non` si la demande est refusée,
  – `timeout` si le délai imparti pour réceptionner la réponse est écoulé,
  – `probleme` si un problème inattendu est survenu ;

Protocole du réseau interbancaire

(5) (7) Message de demande d'autorisation

Ce message peut être émis par le réseau interbancaire et réceptionné par un serveur d'acquisition ou bien émis par un serveur d'acquisition et réceptionné par le réseau interbancaire.

Format : `|I...I|DemandeAutorisation|A...A|B...B|\n` avec :

– `I...I` est l'identifiant de l'envoyeur du message (peut être son PID) ;
– `A...A` correspond au montant de la transaction exprimé en centimes d'euros ;
– `B...B` correspond au numéro de la carte sur 16 chiffres décimaux (longueur constante).

(6) (8) Message de réponse

Ce message est fourni par un serveur d'acquisition au réseau interbancaire ou par le réseau interbancaire au serveur d'acquisition.

Format : `|I...I|ReponseAutorisation|A...A|B...B|\n` avec

- `I...I` est l'identifiant de l'envoyeur du message (peut être son PID) ;
- `A...A` est la chaîne de caractères :
  - `oui` si la demande est acceptée,
  - `non` si la demande est refusée,
  - `timeout` si le délai imparti pour réceptionner la réponse est écoulé,
  - `probleme` si un problème inattendu est survenu ;
- `B...B` est le numéro de carte (16 chiffres) sur laquelle la transaction a été passée.

Le serveur d'autorisation

Message de demande d'autorisation

Ce message est envoyé par le serveur d'acquisition au serveur d'autorisation.

Format : `|I...I|DemandeAutorisation|A...A|B...B|\n` avec :

- `I...I` est l'identifiant du serveur d'acquisition (peut être sont PID) ;
- `A...A` correspond au montant de la transaction exprimé en centimes d'euros ;
- `B...B` correspond au numéro de la carte sur 16 chiffres décimaux (longueur constante).

Message de réponse

Ce message est envoyé par le serveur d'autorisation au serveur d'acquisition.

Format : `|I...I|ReponseAutorisation|A...A|B...B|\n` avec

- `I...I` est l'identifiant du serveur d'autorisation (peut être son PID) ;
- `A...A` est la chaîne de caractères :
  - `oui` si la demande est acceptée,
  - `non` si la demande est refusée,
  - `timeout` si le délai imparti pour réceptionner la réponse est écoulé,
  - `probleme` si un problème inattendu est survenu ;
- `B...B` est le numéro de carte (16 chiffres) sur laquelle la transaction a été passée.

Les *socket*

Dans cette partie, vous trouverez quelques explications simples sur le principe de fonctionnement des *socket* et sur les quelques programmes fournis.

Qu'est-ce qu'une *socket* ?

Un chapitre complet du polycopié (hors programme du cours) décrit le fonctionnement et la mise en œuvre des *socket* et les élèves qui souhaitent réellement comprendre doivent s'y référer. Les explications ci-dessous ne sont qu'un éclairage rapide et, pour

rester concis, nous utiliserons l'analogie entre les communications par *socket* et les communications téléphoniques.

Une *socket* est l'équivalent de l'ensemble constitué par un poste téléphonique et par le câble de téléphone qui y arrive. Contrairement aux installations téléphoniques, que l'on a tendance à ne pas faire et défaire tout le temps, les *socket* sont créées à la demande et détruites dès qu'on ne s'en sert plus.

Pour utiliser un poste téléphonique, il faut disposer d'une ligne, référencée par un numéro unique (le numéro de téléphone). Idem pour les *socket*, même s'il est possible dans certains cas d'utiliser une *socket* sans qu'elle n'ait de numéro associé.

Le numéro de téléphone qui est associé à une ligne donnée dépend de la localisation de cette ligne : les lignes téléphoniques d'un même quartier sont gérées par le même central téléphonique (aussi appelé autocommutateur ou PABX) et la transmission des communications de central à central s'effectue justement en repérant les premiers chiffres du numéro de téléphone : tous les numéros en 01 45 52 MC DU, par exemple, sont gérés par le central de la cité de l'air [7]. Les chiffres MCDU sont alors déterminés en fonction des disponibilités, souvent de façon incrémentale, mais sans que cela n'ait réellement de sens : 54 01 pour le standard de l'ENSTA, 44 06 pour le DFR, 44 24 pour le DFR/A, etc.

Pour les *sockets*, le rôle du central téléphonique de quartier est tenu par le système d'exploitation de l'ordinateur sur lequel la *socket* est créée. L'équivalent du numéro de ligne s'appelle un numéro de port [8] et l'équivalent des 6 premiers chiffres du numéro de téléphone est donné par le nom de l'ordinateur, ces deux éléments étant par convention séparés par le caractère « : ». Par exemple, une *socket* pourra être accessible via le « numéro de ligne » `menthe2.ensta.fr:1234`.

Il est à noter que, en pratique, tous les ordinateurs reliés à l'Internet (et de façon générale tous les ordinateurs communiquant par protocole IP) sont désignés par un numéro unique [9] appelé « adresse IP ». Par exemple, menthe2 a pour adresse IP 147.250.9.32, menthe3 a pour adresse IP 147.250.9.33, ivoire3 a pour adresse IP 147.250.9.103, www.ensta.fr a pour adresse 147.250.7.70, www.sncf.com a pour adresse 195.25.238.132, etc. La correspondance entre l'adresse IP d'une machine et son nom est établi par une machine particulière, le DNS (pour *Domain Name System*),

7. En pratique, les choses sont un tout petit peu plus compliquées que ça : un numéro de téléphone à 10 chiffres est de la forme E Z ABPQ MCDU, où E désigne l'opérateur longue distance chargé d'acheminer la communication (0 pour l'opérateur par défaut de boucle locale), Z désigne la zone concernée (1 à 5 et 9 pour les fixes, 6 ou 7 pour les mobiles, 8 pour les numéros spéciaux), ABPQ désigne le commutateur de rattachement et MCDU désigne le numéro de la ligne sur le commutateur de rattachement (Milliers Centaines Dizaines Unités).

8. Le terme de port est fréquemment utilisé en informatique. De façon générale, il représente un point d'entrée ou de sortie pour un échange de données entre deux zones informatiques différentes. À rapprocher des ports maritimes qui sont les points d'entrée et de sortie des échanges de marchandises entre la zone terrestre et la zone maritime. De même que certains ports maritimes sont spécialisés dans certains types de marchandises, certains ports informatiques sont dédiés à certains types d'échange de données (mail, WWW, etc.).

9. Au moins sur le réseau sur lequel est connecté cette machine.

et le réseau Internet contient un certain nombres d'ordinateurs assurant ensemble et de façon synchrone ce service pour l'ensemble des machines connectées à l'Internet.

L'équivalent du numéro de téléphone pour les *socket* est donc, en réalité, non pas le nom de la machine suivi du numéro de port, mais l'adresse IP de la machine suivie du numéro de port : `147.250.9.32:1234` pour menthe2, par exemple.

Nous conserverons par la suite le premier formalisme car il est plus simple et qu'il offre un avantage majeur : la correspondance entre le nom d'une machine et son adresse IP (la « résolution de nom » en jargon informatique) étant effectuée à chaque utilisation du nom, l'adresse IP de l'ordinateur peut être modifiée dans le temps sans que cela n'affecte les communications.

Comment communique-t-on par *socket* ?

La communication par *socket* est similaire à celle par téléphone : l'un des postes téléphoniques doit émettre un appel, en indiquant le numéro de la ligne à atteindre, et l'autre poste doit attendre l'appel, puis décrocher pour établir la communication. Idem pour les *socket*, chaque action à entreprendre (composer le numéro, attendre la sonnerie, décrocher, etc.) étant réalisée au travers d'un appel système demandé par un programme particulier.

La façon dont les communications par *socket* sont généralement établies et traitées s'apparente cependant plutôt à une communication téléphonique entre un particulier et une entreprise : le particulier appelle le standard de l'entreprise afin de parler avec M. Untel, par exemple, la standardiste répond et transfère la communication sur le poste de M. Untel, ce qui lui permet de rester disponible pour d'autres appels.

Ainsi, lorsque le programme ayant créé la *socket* que l'on cherche à joindre (programme que nous nommerons « serveur » pour la suite de l'explication) accepte d'établir la communication avec le programme qui l'a demandée (programme que nous nommerons « client » pour la suite de l'explication), le serveur crée une autre *socket* et lui transmet la communication, de façon totalement similaire à ce qu'aurait fait la standardiste (à ceci près que les *sockets* sont créées à la demande et détruites dès qu'elles ne sont plus utilisées).

Le plus souvent, ceci est effectué via la création d'un fils (`fork()`) totalement dédié à cette communication, ce qui permet au père (le serveur) de rester disponible pour d'autres communications (d'autres connexions, dans le langage informatique). C'est ainsi que fonctionne la plupart des serveurs WWW par exemple.

21

Le Scaphandre et le Papillon

21.1 Énoncé

Ce projet est librement adapté de l'écriture du livre *Le Scaphandre et le Papillon*. L'auteur de ce livre, Jean-Dominique Bauby, était atteint du *locked-in syndrome* qui le paralysait presque totalement et il a dicté l'intégralité du texte par le seul battement de sa paupière gauche. Afin de l'aider dans cette tâche, une personne énumérait à voix haute les lettres de l'alphabet dans un ordre déterminé et attendait le signal de l'auteur pour sélectionner une lettre. Parfois, cette personne complétait d'elle-même les mots et proposait le mot entier pour confirmation.

Le but du projet est de faire faire par un ordinateur le travail de la personne chargée de prendre la dictée, c'est-à-dire :

- énumérer les lettres de l'alphabet dans un ordre donné (qui peut éventuellement changer au fur et à mesure) ;
- détecter les battements de paupière de l'utilisateur (qui seront modélisés au moyen d'une touche du clavier) et arrêter l'énumération ;
- recueillir la lettre ainsi désignée pour la concaténer au mot courant ;
- consulter plusieurs dictionnaires afin de proposer une terminaison probable au mot courant ;
- prévenir l'utilisateur en cas d'erreur probable, notamment si aucune terminaison n'a pu être trouvée, et intervenir pour proposer une rectification ;
- passer au caractère suivant.

21.2 Contraintes générales

La réalisation de ce projet ne doit pas être vue comme la simple mise en œuvre des notions abordées tout au long de ce document : chaque projet doit prendre en compte le besoin réel du handicapé et doit proposer des solutions qui peuvent effectivement être appliquées. Avant toute considération technique concernant les systèmes d'exploitation et la programmation en `C`, il convient donc d'effectuer une analyse complète des besoins du handicapé et de définir les différentes fonctionnalités qui doivent être prises en charge.

En particulier, il est capital de regrouper les fonctionnalités qui ne peuvent être dissociées et de bien séparer les fonctionnalités qui n'ont pas de liens directs. Ceci permettra de développer des projets modulaires qui pourront s'adapter facilement aux conditions réelles d'emploi et aux évolutions des matériels utilisés. Ainsi, il serait peu judicieux de considérer que la détection des clignements de paupière sera effectuée par le même biais (par le même processus en l'occurrence) que l'affichage du mot courant...

Formulé différemment, il est capital de ne pas faire de présupposé sur l'équipement dont dispose le handicapé pour dicter son texte. Notons que ceci ne correspond pas à une volonté pédagogique de la part de l'équipe de ce cours, mais bien à une approche pratique des problèmes qui se posent en réalité : la tâche correspondant à l'affichage des lettres, par exemple, est totalement indépendante de celle visant à compléter les mots et l'indépendance effective de ces deux tâches permet de réduire les coûts de développement. Ainsi, si le handicapé utilise dans un premier temps un minitel pour communiquer avec le programme principal et si, convaincu de l'utilité d'un tel programme, il décide de s'acheter un PC et un modem pour dialoguer avec lui, seuls les codes des tâches d'affichage et de communication sont à réécrire.

Ce raisonnement peut encore être développé : si les tâches sont véritablement indépendantes et si elles dialoguent en utilisant un protocole de communication prédéfini (par exemple, un protocole standard), les différents programmes permettant d'afficher, de communiquer, etc. peuvent être choisis « sur étagère », c'est-à-dire qu'il n'est pas nécessaire de les développer [1] : il suffit d'acheter au plus bas prix de tels programmes.

21.3 Contraintes techniques

Un certain nombre de contraintes techniques sont imposées afin de canaliser les efforts de développement qui peuvent être faits sur ce sujet. Certaines de ces contraintes découlent des contraintes générales décrites ci-dessus (il serait judicieux de trouver lesquelles et de voir pourquoi), d'autres visent simplement à limiter la quantité de travail à fournir.

Insistons sur le fait que la décomposition du projet en unités fonctionnelles indépendantes est un travail qui peut paraître pénible au premier abord mais qui permet rapidement de se dégager des inconvénients de développement des projets lourds : chaque unité fonctionnelle peut être développée indépendamment du reste du projet et, donc, peut être déboguée et testée indépendamment...

En pratique, les unités fonctionnelles évoquées ci-dessus seront des processus indépendants et le projet sera donc constitué d'un certain nombre de programmes différents, amenés à communiquer les uns avec les autres.

Les contraintes techniques imposées sont les suivantes :

– Les différents programmes seront écrits en `C` sans faire appel à des bibliothèques spécialisées. En particulier, aucune bibliothèque graphique n'est autorisée (la

1. Cette remarque s'adresse aux futurs ingénieurs, mais pas aux actuels élèves...

bibliothèque `termios` peut être utilisée à l'exception des fonctionnalités rendant la saisie non bloquante).
- Un seul processus dialoguera avec l'utilisateur. Ce processus se limitera à la détection des clignements (pourquoi ?).
- Tous les affichages à l'écran seront effectués par un seul et unique processus, dont ce sera la seule et unique fonction (pourquoi ?).
- La recherche dans les dictionnaires sera assurée par plusieurs autres processus totalement indépendants du processus principal et fonctionnant avec n'importe quel dictionnaire. Si le projet ne fonctionne qu'avec un format particulier de dictionnaires (dictionnaires triés par ordre alphabétique, par exemple), il conviendra de vérifier au moment de l'exécution que les dictionnaires fournis conviennent.
- Les mécanismes de communication et de synchronisation décrits dans ce document seront utilisés afin d'assurer que le handicapé est **toujours** prioritaire, c'est-à-dire qu'il n'a jamais besoin d'attendre quoi que ce soit (et surtout pas la fin d'une recherche dans un dictionnaire) pour dicter du texte.
- Toute particularité du projet qui empêcherait d'utiliser des modules déjà développés et répondant au cahier des charges (cf. remarque précédente sur les achats « sur étagère ») est à proscrire. De même, est à proscrire toute particularité qui empêcherait de transformer aisément ce projet en un projet distribué sur un réseau de machines. Formulé différemment, tous les processus mis en œuvre par ce projet tourneront sur la même machine afin de simplifier les développements (sauf pour les élèves ayant choisi d'utiliser des *sockets* au lieu de tuyaux de communication), mais il est important de concevoir une architecture qui ne tire pas parti de cette simplification et qui considère que chaque processus peut être sur une machine différente. Ainsi, en particulier, toute utilisation de fichier sur support local (disque dur par exemple) comme moyen de communicatione est prohibée.
- L'utilisation de fonctions comme `sleep()`, `system()` ou tout autre fonction des bibliothèques du langage `C` pouvant se substiter aux appels système décrits dans ce document est **très fortement** déconseilée car les programmes utilisant à la fois de telles fonctions et les appels système en question ont un comportement parfois difficilement compréhensible pour le néophyte.
- L'architecture de votre programme tiendra compte des services à mettre en œuvre, même si le temps imparti ne permet pas de les programmer. Notons que cette dernière contrainte n'en est pas une si vous appliquez les règles édictées ci-dessus...

21.4 Services à mettre en œuvre

Le projet comportera au minimum les services décrits dans l'énoncé (section 21.1), à savoir l'énumération et la sélection de lettres, la consultation d'au moins 2 dictionnaires pour proposer des terminaisons et la concaténation des éléments sélectionnés

par le handicapé (lettre ou terminaison) pour créer des mots.

Vous programmerez aussi dans l'ordre les services décrits ci-dessous.

Correction des erreurs de dictée

Il est probable que des erreurs surviennent au cours de la dictée (arrêt tardif de l'énumération, changement d'avis, étourderie) et il convient donc de laisser la possibilité à l'utilisateur d'effectuer des rectifications.

Dans le cas de Jean-Dominique Bauby, ces erreurs étaient traitées par des mimiques (regard horrifié pour signaler une erreur, condescendant pour approuver...) et simplifiées par la psychologie de l'assistante. Le but étant de s'en passer, vous aller au contraire proposer au handicapé de rectifier lui-même le texte, avec des fonctions de correction (annulation du dernier caractère et du dernier mot). Vous vous limiterez à trois touches, la troisième touche correspondant au clignement des deux paupières...

Utilisation d'un alphabet spécialisé

Parcourir l'alphabet n'est pas toujours la meilleure solution pour dicter un mot (calculez le nombre d'énumérations nécessaires pour dicter le mot xylophone !). Pour améliorer le procédé, vous utiliserez un alphabet spécialisé reflétant la probabilité d'utilisation des lettres de l'alphabet dans le texte dicté.

Voici par exemple l'ordre utilisé par Claude Mendibil et Jean-Dominique Bauby (qui correspond en fait à la fréquence d'apparition des lettres dans la langue française) :

E S A R I N T U L O M D P C F B V H G J Q Z Y X K W

Utilisation d'un alphabet dynamique

L'utilisation d'un alphabet spécialisé permet probablement de gagner du temps en moyenne, mais peut s'avérer peu efficace dans certains cas (refaire le test avec l'alphabet proposé ci-dessus et le mot xylophone).

Afin d'augmenter encore l'efficacité du logiciel, vous utiliserez des alphabets dynamiques dont l'ordre d'apparition des lettres sera recalculé à chaque sélection d'une lettre par le handicapé, par exemple en fonction des terminaisons probables du mot courant.

Attention, cette amélioration doit tenir compte de deux points importants : le programme doit pouvoir fonctionner avec tout type de dictionnaire et il ne faut pas empêcher le handicapé de créer des néologismes.

Utilisation d'un dictionnaire spécialisé

Avec le dictionnaire français courant, votre programme ne devinera jamais des mots comme « ensta » ou « babasse » pratiqués régulièrement par un handicapé. À

partir de maintenant, quand vous rencontrerez un mot nouveau, après la confirmation d'usage, vous proposerez d'insérer ce mot dans un dictionnaire personnel...

Ce dictionnaire se présentera comme les autres du point de vue de la recherche lexicographique, mais sera plus subtil, puisqu'il pourra s'enrichir au fur et à mesure de l'exécution du programme.

Aide d'un opérateur humain

Cette extension s'adresse aux programmeurs avertis qui voudraient essayer la programmation des *sockets*.

La terminaison automatique à l'aide d'un dictionnaire n'est pas simple pour les mots conjugués ou les néologismes. Dans ce cadre là, la participation d'un opérateur humain (qui peut deviner l'intention du handicapé) est un atout important. Néanmoins, il est toujours difficile de trouver des volontaires qui acceptent de se rendre au chevet du handicapé...

L'idéal serait donc de permettre au « dictionnaire humain » d'intervenir à distance, via un réseau de communication, au travers d'une interface similaire à celle utilisée par le handicapé.

21.5 Précisions

Des dictionnaires sont facilement accessibles sur la toile.

Le point important de ce projet est d'écrire des programmes indépendants les uns des autres qui vont communiquer entre eux par l'intermédiaire de tuyaux. Il n'est pas nécessaire de spécialiser ces programmes et chaque programme devrait être interchangeable avec le programme écrit par un autre élève pour ce projet.

Signalons par ailleurs que ce projet est assez lourd et qu'il convient de le commencer très tôt. Ceci n'est possible que si le projet est conçu dès le départ de façon modulaire : certains outils, en effet, sont abordés très tard dans les TP et il serait dangereux d'attendre de tout maîtriser avant de débuter le projet. En particulier, il n'est pas nécessaire de comprendre le fonctionnement des tuyaux de communication pour écrire les différents programmes indépendants qui réaliseront les fonctions d'énumération, de détection de clignement, de recherche de terminaisons, etc.

D'un point de vue technique, tous les processus indépendants peuvent utiliser l'entrée standard et la sortie standard pour échanger des données. Ces entrée et sortie standard seront ensuite redirigées dans des tuyaux de communication, ce qui permettra d'établir des communications sans avoir à modifier ces programmes...

21.6 Déroulement du projet

Les conditions de remise du projet sont décrites sur la page WWW du cours.

22

Projet de Jukebox

Ce projet propose la conception d'un système de sélection et d'audition de morceaux numériques de musique en environnement multi-utilisateurs.

22.1 Énoncé

Le but de ce projet est de concevoir un système permettant d'écouter de la musique dans une salle informatique où plusieurs personnes travaillent (typiquement une salle de cours). Une seule machine est chargée de décoder les fichiers sons, stockés au format MPEG-I layer 3 (noté fréquemment MP3) et de les envoyer (via la carte son) à un amplificateur audio afin que tous en profitent.

Chaque utilisateur peut ainsi choisir depuis sa console quels morceaux de musique il souhaite écouter et il dispose en retour des informations concernant le morceau en cours (affichées sur son écran, avec au minimum le titre et un compteur de temps).

La machine effectuant le décodage des fichiers de musique devra ainsi communiquer avec les différentes personnes de la salle (directement sur leur console), décider des morceaux à jouer en fonction des désirs de chacun et renvoyer des informations en retour.

Le fonctionnement de l'ensemble sera le plus dynamique possible et, notamment, les deux cas suivants devront être traités :

- un utilisateur pourra arriver dans la salle alors que tout fonctionne, se connecter à un des postes de travail et donner ses desiderata ;
- un utilisateur déjà connecté et ayant effectué des demandes d'écoute de morceaux de musique pourra se déconnecter sans perturber les autres utilisateurs.

Afin d'avoir le maximum de souplesse, on supposera que la musique est stockée et accessible (sous forme de fichier) à partir de la machine qui effectue la lecture, mais pas nécessairement depuis les consoles des personnes présentes dans la salle.

22.2 Outils utilisés

Ce projet ne visant pas à l'écriture d'un décodeur MP3, un logiciel libre sera utilisé à cet effet : **`mpg123`**. Ce logiciel est fourni dans nombre de distributions de Linux et il

est installé sur toutes les machines Linux de l'ENSTA. Ses sources sont disponibles sur `http://www.mpg123.de` et il sera nécessaire de les parcourir : ce logiciel offre en effet l'avantage de pouvoir être piloté grâce à un jeu de commandes envoyées sur l'entrée standard (les retours de valeurs étant lues sur sa sortie standard), mais ces fonctionnalités ne sont pas détaillées dans la page de manuel qui s'attache plutôt à montrer les différentes options utilisables en ligne de commande (cf. 22.3).

Le contrôle du volume sonore (et éventuellement d'autres réglages, comme balance et graves/aigues) de la machine qui commande l'amplificateur audio pourra être fait directement avec les bons appels `ioctl()` appliqués au périphérique `/dev/mixer`. Ceci est spécifique à Linux et à rechercher dans les pages de manuel et les documentations de OSS, le gestionnaire de son utilisé sous Linux [1].

Dans un premier temps, on pourra se passer de commander la carte son ou on pourra utiliser le programme **`aumix`** (ou **`mixer`**) avec une ligne de commande adéquate. Il ne s'agit toutefois pas d'une solution élégante car **`aumix`** doit être exécuté pour chaque modification de volume...

Les autres éléments du projet sont à programmer en `C` directement. Les tests du projet se feront au casque sur une machine disposant d'une carte sonore adéquate. Dans le pire des cas, il est possible de faire fonctionner le programme **`mpg123`** sans carte son à des fins de vérification.

22.3 Commandes de `mpg123`

Pour commander **`mpg123`** depuis son entrée/sortie standard, il faut le lancer ainsi :

```
mpg123 --aggressive -R -
```

L'option « `--aggressive` » est facultative, mais demande au programme d'utiliser si possible les fonctionnalité d'ordonnancement temps réel [2] afin de garantir un décodage sans coupure de la musique (lire un fichier son est une tâche typiquement temps réel). « `-R` » dit au programme de fonctionner dans le mode télécommandé (**`remote`**). Le paramètre « - » est la façon habituelle sous Unix de dénommer l'entrée/sortie standard (cette option est présente uniquement parce que **`mpg123`** attend par défaut un nom de fichier).

Remarque : pour faire fonctionner le programme sans carte son, on peut utiliser l'option `-t` de test (mais les temps ne sont pas respectés) ou « `-w <fichier.wav>` » qui écrit le résultat dans un fichier.

Voici les commandes que l'on peut envoyer par l'entrée standard dans ce mode (il faut envoyer chaque commande sous la forme d'une ligne de texte complète) :

1. Voir `/usr/include/linux/soundcard.h` et les sources d'un programme de mixage simple comme **`mixer`**.
2. Il semble que c'est inutile à l'ENSTA car le programme ne dispose pas de droits suffisant pour utiliser ces fonctionnalités.

QUIT : arrêter tout et quitter le programme ;
LOAD <f> : charger et jouer le fichier <f> (fonctionne aussi avec une URL, mais on ne l'utilisera pas) ;
STOP : arrêter la lecture sans quitter le programme ;
PAUSE : mettre en pause / redémarrer la lecture ;
JUMP [+|-] <n> : déplacement dans le fichier de <n> trames.

Remarque : une trame MP3 fait 1152 échantillons sonores, donc à une fréquence d'échantillonnage de 44100 Hz (correspondant à un CD-audio), cela donne un peu plus de 38 trames/s. Quelques exemples pour JUMP :

JUMP +38 : avance de 38 trames (une seconde environ) ;
JUMP -64 : recule de 64 trames (revient au début du fichier si moins de 64 trames se sont écoulées) ;
JUMP 0 : revient au début du fichier courant ;
JUMP 200 : saute directement à la trame numéro 200.

Le programme **mpg123** retourne des réponses (et des lignes d'informations) dont le format suit. Les valeurs de retour sont envoyées sur la sortie standard, les erreurs sont renvoyées sur la sortie d'erreur (`stderr`), il faut donc observer les deux.

@R MPG123 : message envoyé au démarrage du programme.

@I <a> : Message envoyé au chargement d'un fichier, <a> est le nom du fichier sans son chemin ni extension. Cette réponse est renvoyée après un LOAD pour un fichier ne contenant pas d'informations ID3-tag (zone d'information réservée dans le fichier MP3).

@I ID3: <abcdef> : message envoyé au chargement d'un fichier contenant un ID3-tag ;
<a> : titre (exactement 30 caractères),
<b> : artiste (exactement 30 caractères),
<c> : album (exactement 30 caractères),
<d> : année (exactement 4 caractères),
<e> : commentaire (exactement 30 caractères),
<f> : genre du morceau (chaîne de longueur non-fixée).

@S <abcdefghijkl> : message donnant les principales informations extraites de l'en-tête du fichier lu, avec :
<a> : type de MPEG (string),
<b> : « layer » (int),
<c> : fréquence d'échantillonnage (int),
<d> : mode (string),
<e> : extension de mode (int),
<f> : taille de la trame (int),
<g> : stereo (int),
<h> : copyright (int),
<i> : code correcteur d'erreur (int),

`<j>` : préaccentuation (int),
`<k>` : débit (en kbits/s) (int),
`<l>` : extension (int).

`@F <abcd>` : message envoyé ***à chaque trame lue*** avec :
`<a>` : numéro de trame (`int`),
`<b>` : nombres de trames restantes pour ce fichier (`int`),
`<c>` : secondes (`float`),
`<d>` : secondes restantes (`float`).

`@P <a>` : message de changement d'état avec :
`0` : lecture arrêtée (commande STOP, ou à la fin normale du morceau),
`1` : en pause (commande PAUSE),
`2` : redémarrage après pause (commande PAUSE).

Voici un exemple de session, dans laquelle les chaînes envoyées vers l'entrée standard de **`mpg123`** sont précédées de `>>` :

```
Idefix> mpg123 --aggressive -R -
    @R MPG123
>> LOAD 03_Guide_vocal.mp3
  @I ID3:Guide vocal Genesis Duke
                  1983 AlternRock
  @S 1.0 3 44100 Stereo 0 418 2 0 0 0 128 0
  @F 1 3617 0.03 94.48
  @F 2 3616 0.05 94.46
  @F 3 3615 0.08 94.43
  @F 4 3614 0.10 94.41
  ...
  @F 94 3524 2.46 92.06
  @F 95 3523 2.48 92.03
>> JUMP +32
  @F 96 3522 2.51 92.00
  @F 129 3489 3.37 91.14
  @F 130 3488 3.40 91.12
  ...
  @F 192 3426 5.02 89.50
>> PAUSE
  @P 1
>> PAUSE
  @P 2
  @F 193 3425 5.04 89.47
  @F 194 3424 5.07 89.44
  ...
  @F 227 3391 5.93 88.58
```

```
>> STOP
  @P 0
>> QUIT
```

Comme il y a un retour de valeur à chaque trame décodée, il faudra commencer par écrire un programme capable de lancer **mpg123** et de dialoguer avec lui, en particulier pour absorber les valeurs de retour et stocker l'état du décodeur en interne. On gérera également les cas où le programme se termine avec une erreur, tout particulièrement lorsque l'on demande le chargement d'un fichier inexistant (bien que ce cas ne se produise normalement pas dans le projet final).

22.4 Les *ID3-tag*

Afin de proposer à l'utilisateur une liste de morceaux plus explicite que simplement les noms des fichiers, on pourra utiliser les *ID3-tags*. Ceux-ci sont retournés par **mpg123** au début de la lecture d'un morceau, mais pour pouvoir présenter l'ensemble des fichiers au choix, il faut faire soi-même la lecture, et ceci est très simple.

Les informations du *ID3-tag* sont stockées dans les 128 derniers octets du fichier MP3. Ces 128 derniers octets se répartissent comme suit dans l'ordre (c'est l'ordre dans lequel **mpg123** les retourne également) :

- Les trois caractères « TAG ». Ceci permet de savoir si le fichier dispose ou non d'un *ID3-tag* ;
- Le titre du morceau sous la forme d'un bloc de 30 caractères, le titre étant complété à 30 caractères par des espaces ;
- Le nom de l'artiste sur 30 caractères ;
- Le nom de l'album sur 30 caractères ;
- L'année (en texte) sous la forme de 4 caractères exactement ;
- Un commentaire sous la forme de 30 caractères (30 espaces s'il n'y a pas de commentaire) ;
- Un octet qui classe le type de musique selon une table standard (disponible sur la page du cours).

22.5 Suggestion d'un plan d'action

- Dans un premier temps, il faudra écrire un programme capable de lancer **mpg123** et d'interagir avec lui, en particulier pour récupérer en permanence les informations de trame lue (@F) sans bloquer le décodage, et pour gérer les arrêts intempestifs (fichier inexistant..) du programme. Les commandes de ce programme d'encapsulation seront prises également en entrée et sortie standard, avec une syntaxe que l'on pourra choisir similaire à celle de **mpg123**.
- Ensuite on choisira une politique simple pour décider du prochain morceau à jouer et on écrira l'interface. Une solution en texte est suffisante dans un premier

temps, avec par exemple des listes de choix et des numéros. Il faudra identifier clairement à ce niveau les communications et leur format entre les interfaces et le juke-box.
– On fera fonctionner l'ensemble avec plusieurs interfaces concurrentes. Pour cela, on pourra utiliser des tuyaux nommés pour faire communiquer les interfaces avec le serveur, en lançant tous les processus sur la même machine. Les plus courageux feront de la communication par *sockets*, ce qui permettra de faire fonctionner les interfaces sur des machines différentes du serveur.

Améliorations possibles

– Afficher la liste des morceaux grâce aux informations extraites des ID3-tags (et pas seulement les noms des fichiers).
– Ajouter la commande du volume, de la balance... C'est l'occasion également de chercher des politiques à ce niveau (le plus simple est par exemple de donner priorité à celui qui souhaite baisser la volume, ou faire une moyenne de chacune des interfaces...).
– Améliorer le système de choix des morceaux (minimisation de l'insatisfaction moyenne, avec un vote sur une liste de morceaux proposés par exemple). Créer une liste des morceaux prévus à la diffusion et dynamique, rafraîchie sur chaque interface lors des évolutions.
– Une jolie petite interface en `ncurses`, en `Tk` ou autres (mais ne pas passer trop de temps à faire du X11).

23

Administration distante

L'énoncé de ce projet a pour but de vous guider pas à pas dans la réalisation du projet. Les différents services qui doivent être rendus par vos programmes peuvent être facilement programmés de manière incrémentale et vérifiés au fur et à mesure.

> Ce projet a pour but d'introduire la notion d'administration à distance de plusieurs stations. Nous nous proposons de mettre en place un système simple permettant différentes actions : consulter une liste de fichiers, copier des fichiers d'un client à un autre ou d'un serveur central vers les clients... Ces commandes seront accessibles depuis un terminal client (appelé `Interface`) et devront être analysées puis relayées par un serveur d'administration (appelé `Administrateur`) et enfin être appliquer sur les différents clients (appelés `Machine`). L'intégralité du projet prendra place sur une seule machine et nous nous servirons des tuyaux pour simuler des connexions TCP entre les différents acteurs.
>
> Aucune contrainte d'organisation des différents éléments n'est donnée. Toutefois le cahier des charges décrit ci-après devra être scrupuleusement respecté. Le projet en version minimale est simple.
>
> Les éventuelles améliorations ne doivent être abordées qu'après avoir finalisé le projet dans sa version minimale. Si un projet est rendu dans une version améliorée incomplète (ou non fonctionnelle) seules les parties de la version minimale seront prises en compte pour la correction et l'établissement de la note.

23.1 Position du problème

On désire pouvoir administrer (dans un sens très restreint) plusieurs machines à partir d'un serveur central. Chaque machine sera simulée par un processus qui trouvera dans un fichier de configuration les informations relatives à son fonctionnement. De la même façon le serveur central sera simulé par un processus. Afin d'accéder au serveur central on utilisera un dernier processus qui simulera une interface de dialogue. On désire pouvoir accomplir les actions suivantes à partir de l'`Interface` :

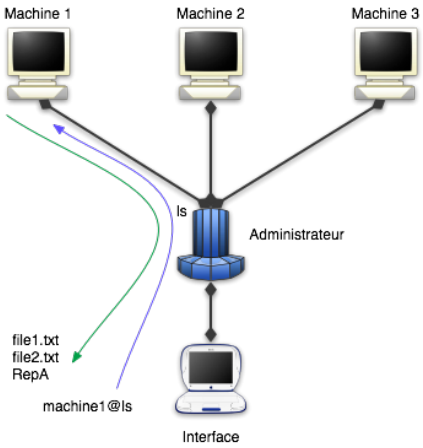


FIGURE 23.1 – *Architecture générale. L'`Interface` envoie des commandes à l'`Administrateur` qui réalise le routage et éventuellement le traitement de commandes plus complexes.*

– obtenir le contenu d'un répertoire d'une `Machine`,
– copier un fichier d'une `Machine` vers **toutes** les autres.

Ceci permet par exemple, à partir d'un fichier de configuration modèle `/etc/ssh/sshd.conf`, de placer une configuration générique sur toutes `Machines` nouvellement créées. De nombreuses commandes sont possibles mais nous nous limiterons dans un premier temps à celles décrites ci-dessus.

23.2 Le processus `Machine`

Configuration

Une machine distante est tout simplement un processus. Afin de simuler une machine, ce processus gère **un** répertoire qui sera représentatif du répertoire « racine » d'une machine réelle. Ce répertoire sera inscrit dans un fichier de configuration dont le nom sera donné par la ligne de commande. Voici un exemple :

```
$> more config1.conf
RACINE: /net/utilisateur/bcollin/IN201/projet2008/reptest1
...
$> more config2.conf
RACINE: /net/utilisateur/bcollin/IN201/projet2008/reptest2
...
$> machine config1.conf
...
$> machine config2.conf
```

...

La figure 23.2 donne une vue générale de l'ensemble des fichiers de configurations.

Le processus doit être capable de connaître les fichiers et les répertoires présents dans son répertoire « racine ». À cette fin on utilisera les fonctions systèmes décrites dans l'aide mémoire situé à la fin de ce document. Ces fonctions sont les suivantes :

- `DIRP *opendir(const char *pathname)`
- `struct dirent *readdir(DIRP *fd)`
- `int closedir(DIRP *fd)`

Les commandes à interpréter

Le processus `Machine` doit pouvoir dialoguer avec le serveur d'administration. Il devra lire les commandes sur son entrée standard (`stdin`) et écrire le résultat sur sa sortie standard (`stdout`). Le mécanisme de création de tuyaux et de duplication vu en petite classe permettra de mettre en place le dialogue à partir du processus d'administration.

Le processus `Machine` doit pouvoir comprendre *a minima* les commandes suivantes :

- `ls` : le processus retourne la liste des fichiers et des répertoires à l'endroit où il se trouve.
- `send nom_file` : le processus renvoie les données contenues dans le fichier `nom_file`
- `write nom_file size data` : le processus écrit les données `data` dont la taille totale est de `size` octets dans le fichier `nom_file` dans le répertoire où il se trouve.

Le dialogue doit pouvoir gérer l'envoi ou la réception de données quelconques. Il est donc impératif d'établir un protocole de dialogue. Ce protocole sera abordée ultérieurement.

23.3 Le processus `Administrateur`

Ce processus joue un rôle central d'aiguillage des commandes reçues depuis l'interface. Quand une commande est interprétée, il interagit avec les différentes `Machines`. L'administrateur reçoit ses commandes sur son entrée standard (`stdin`) et il envoie certains des résultats qu'il recevra des `Machines` sur sa sortie standard.

Configuration

Le processus `Administrateur` possède un fichier de configuration dans lequel il trouvera la liste des clients qu'il doit administrer ainsi que leur fichier de configuration. Les informations contenues dans ce fichier lui permettront ainsi de créer les différents processus `Machines`. Ce fichier peut être constitué de la manière suivante :

```
machine1 /net/utilisateur/bcollin/IN201/projet2008/config1.conf
machine2 /net/utilisateur/bcollin/IN201/projet2008/config2.conf
```

Les fichiers `config1.conf` et `config2.conf` sont ceux détaillés dans la partie précédente. Le fichier de configuration sera communiqué au processus sur sa ligne de commande :

```
$> ./administrateur maconf.test
```

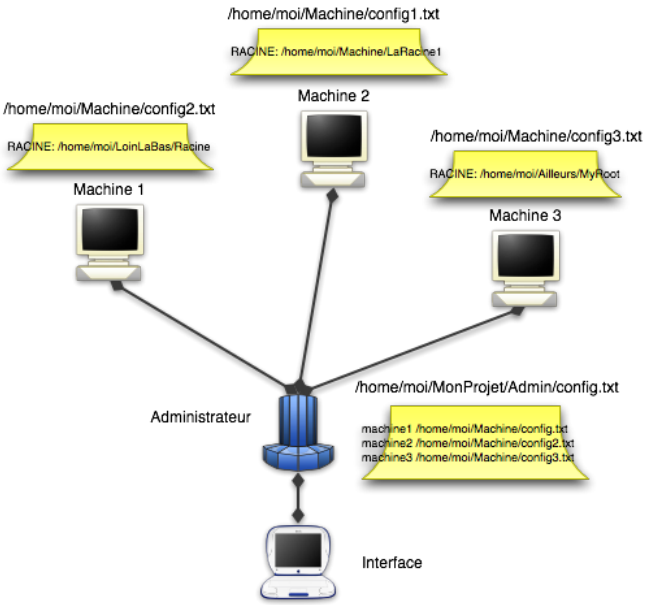


FIGURE 23.2 – *La répartition des fichiers de configuration. L'administrateur possède un fichier de configuration qui lui permet de savoir quelles sont les machines à lancer ainsi que l'emplacement de leur fichier de configuration. Chaque machine reçoit ainsi sur sa ligne de commande le nom de son fichier de configuration. Dans ce fichier se trouve inscrit le chemin de son répertoire « racine ».*

Les commandes à interpréter

L'administrateur jouant le rôle d'un chef d'orchestre, il doit être en mesure d'interagir avec les `Machines` mais aussi avec l'`Interface`. Comme nous l'avons évoqué, l'`Administrateur` dialogue avec l'`Interface` à partir de son entrée standard et de sa

sortie standard. Il dialogue avec les `Machines` à partir des tuyaux qu'il aura mis en place en créant les processus correspondants.

Puisqu'il dialogue avec les `Machines`, l'`Administrateur` doit être capable d'interpréter les résultats renvoyés par les `Machines` :

- lire une liste de fichiers et de répertoires retournée par une `Machine` (résultat de la commande `ls`),
- lire l'ensemble des données (quelconques mais de taille prévue) renvoyées par une `Machine` (résultat de la commande `send`),
- envoyer un ensemble de données (quelconques mais de taille prévue) à une `Machine` (commande `write`).

Il doit **surtout** aiguiller les commandes qu'il reçoit de l'interface. Ceci sera fait en plaçant « l'adresse » de la machine avant la commande destinée à être envoyée. On pourra adopter le protocole suivant : `machine1@ls` pour demander à l'`Administrateur` d'envoyer la commande à la machine 1. On veut surtout pouvoir établir un mécanisme de *broadcast* dans l'envoi de fichier, *i.e.* ne lire qu'une seule fois un fichier distant pour l'envoyer à plusieurs `Machines`. Cette commande venant de l'interface, on convient qu'elle pourra prendre la forme suivante : `machine1@send* nom_file`. Cette commande peut s'interpréter de la manière suivante : aller chercher le fichier `nom_file` sur la `machine1` et l'envoyer à toutes les autres. Nous devons aussi pouvoir fermer les « connexions » aux différentes machines. Ceci pourra prendre par exemple la forme suivante : `machine1@close`. Enfin nous devons pouvoir quitter l'`Administrateur` et terminer par le même coup l'`Interface`. On peut convenir de la commande suivante : `quit`.

Nous avons donc en résumé les dialogues suivants :

- Dialogue `Administrateur` / `Machine`
  - envoyer `ls` et lire le résultat,
  - envoyer `send` et lire le résultat,
  - envoyer `write`,
  - envoyer `close`,
- Dialogue `Administrateur` / `Interface`
  - `machine@ls` : envoyer la commande `ls` à `machine`, lire le résultat et le renvoyer à l'`Interface`,
  - `machine1@send* nom_file` : envoyer la commande `send` à `machine1` pour obtenir le fichier `nom_file`, lire le résultat et le renvoyer à `machine2` par la commande `write nom_file size data`, puis à la `machine3`,...
  - `machine@close` : signifier au processus `machine1` qu'il doit se terminer et fermer proprement la connexion.
  - `quit` : fermer les connexions aux différentes `Machines` puis fermer l'`Interface` et l'`Administrateur`.

23.4 L'interface

L'interface est créée par l'`Administrateur`. Elle nous permet d'interagir avec l'`Administrateur`. L'interface lit les commandes sur son entrée standard et affiche les résultats sur sa sortie standard. Les commandes saisies (par vos soins) sont donc celles énoncées dans la description du dialogue entre l'`Administrateur` et l'`Interface`.

La lecture des commandes sera faite ligne à ligne, de la même manière que le *shell*. L'interface pourrait donc simplement se comporter de la façon suivante :

- proposer une « invite », tel que % ou encore $,
- lire l'entrée standard et la transmettre à l'`Interface`.

Nous pourrions donc avoir l'affichage suivant :

```
Bienvenue dans l'interface d'administration
%machine1@ls
test1.txt
test2.txt
Mon_dossier
%machine2@ls
Sub1
Sub2
%machine1@send* test1.txt
ok
%machine2@ls
test1.text
Sub1
Sub2
%machine2@send* unknown.txt
error: file not found
%
```

23.5 Les dialogues

Établir un protocole

Comme nous l'avons vu, différents dialogues existent. Il y a d'une part l'envoi des commandes mais aussi la réception des données issues de l'exécution d'une commande ainsi que le traitement des erreurs (fichier inexistant ou machine inexistante). On doit donc impérativement établir un protocole de dialogue afin de différencier ce qui est une réponse correcte d'une erreur mais aussi prévoir la lecture de données quelconques.

La transmission des commandes n'est rien de plus que l'envoi d'un ensemble d'octets ! Il faut donc « encapsuler » ces octets dans quelque chose qui nous permettra de :

- savoir combien de données il convient de lire,
- si ce que nous devons lire est un résultat ou si il s'agit d'une erreur.

On peut convenir d'adopter le protocole qui va suivre. Chaque envoi est constitué de plusieurs champs permettant de décrire s'il s'agit d'une réponse, d'une erreur, quelle est la taille de ce qu'il faut lire et les données elles-mêmes. Un « paquet » aura donc la forme suivante :

```
MXXXXXXXXdata.............
```

Le premier octet de l'envoi contient le mode : 0 s'il s'agit d'une réponse, 2 s'il s'agit d'un envoi et 1 s'il s'agit d'une erreur. Les 8 octets suivants contiennent la taille des données à lire. Les données prendront place à partir du 10ième octet.

Voyons quelques exemples.

- envoi de la commande `ls` :

  `2␣␣␣␣␣␣␣2ls`

  soit 2 car il s'agit d'un envoi, 7 espaces suivies du caractère `2` pour dire que les données transmises tiennent sur 2 octets, et enfin les 2 caractères de la commandes `ls`.
- réception d'une liste de fichiers et de répertoires, par exemple `file1`, `fic3` et `Rep34` :

  `0␣␣␣␣␣␣16file1\nfic3\nRep34`

  Il s'agit d'une réponse (`0`), nous devrons lire 16 caractères et ces caractères sont « file1\nfic3\nRep34 »
- réception d'une erreur car le fichier `unknown.txt` n'existe pas.

  `1␣␣␣␣␣␣27fichier␣absent:␣unknown.txt`

  soit 1 car il s'agit d'une erreur, 6 espaces suivies des caractères « 27 » car nous devons lire les 27 caractères de la réponse.
- réception du contenu d'un fichier en retour de la commande `send`. On supposera que ce fichier contient les lettres de l'alphabet. Attention ce fichier se termine par un retour chariot !

  `0␣␣␣␣␣␣27abcdefghijklmnopqrstuvwxyz\n`

  soit 0 car il s'agit d'une réponse, 6 espaces suivies des caractères « 27 » car nous devons lire les 27 caractères de la réponse et enfin les données contenues dans le fichier.

Entrées / sorties des différents processus

On rappelle que pour tester vos différents processus mais aussi car cela est **imposé**, il est impératif que les échanges se fassent tel que décrit dans la figure 23.3.

23.6 Consignes et cahier des charges

Consignes

Une attention particulière sera prise pour soigner la présentation et la qualité du travail [1]. À cette fin, chaque projet sera envoyé en temps et en heure au MdC de petite classe du groupe auquel vous appartenez.

Un projet doit contenir les éléments suivants :

1. Si vous ne pouvez pas le faire bien, faites le beau !

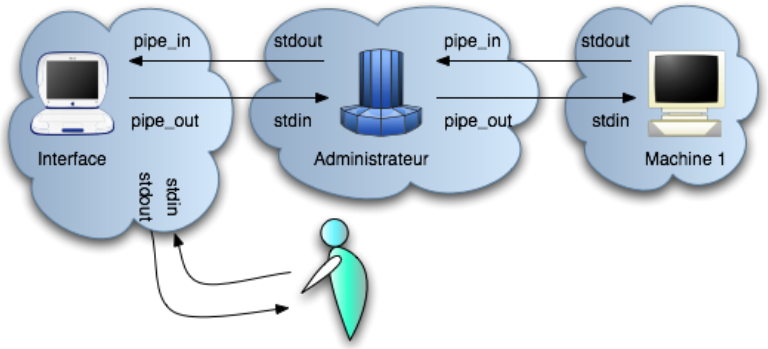


FIGURE 23.3 – *L'interface saisit sur « stdin » ce que vous rentrez et affiche les résultats sur « stdout ». Une machine lit les commandes sur « stdin » et écrit les résultats sur « stdout ». Enfin l'administrateur lit ses commandes sur « stdin » et retourne les résultats sur « stdout ».*

- l'intégralité des fichiers sources (`*.c` et `*.h`). Chaque fichier devrait (conditionnel à nature d'injonction forte) contenir des commentaires amples et variés permettant de comprendre facilement chaque fonction ainsi que les structures que vous jugerez bon de définir.
- un fichier `Makefile` permettant de compiler l'intégralité des programmes par la commande **`make`**.
- l'intégralité des fichiers de configuration nécessaires au fonctionnement de vos programmes,
- un document au format PDF uniquement décrivant votre projet. Ce document doit contenir une description de vos programmes, des flux qui circulent entre les processus, des exemples de fonctionnement, un exemple de ligne de commandes permettant d'exécuter votre projet, ainsi que les points forts et faibles de votre projet.

Cahier des charges

Le projet minimal doit faire fonctionner au moins trois clients, un administrateur et une interface. Ceci signifie qu'au total, en cours d'exécution, le projet minimal comprend 5 programmes communiquant entre eux par des tuyaux (on n'utilisera pas de tuyau nommé). Dans l'idéal, le fichier de configuration de l'`Administrateur` doit permettre de définir autant de machines que l'on souhaite.

La compilation des fichiers sources ne doit donner ni erreur (rédhibitoire) ni avertissement en utilisant l'option `-Wall`.

Les projets seront envoyés par mail aux MdC. À ce titre, l'intégralité des documents nécessaires sera placée dans une archive `tar`. On ne devra pas trouver dans cette archive des fichiers temporaires (`*~` ou `*.o`). Cette archive devra intégrer un répertoire portant le nom des deux élèves du binôme :

```
% mkdir lemeilleur_leplusbeaux
% cp -R tous_mon_projet/* lemeilleur_leplusbeaux/
%tar cvzf projet.tgz lemeilleur_leplusbeau/
```

Ainsi quand le MdC extrait la pièce jointe, soit `projet.tgz` et décompresse cette archive, il doit voir apparaître la structure suivante (il s'agit d'un exemple) :

```
% tar xvzf projet.tgz
lemeilleur_leplusbeau/
lemeilleur_leplusbeau/source/
lemeilleur_leplusbeau/source/admin.c
lemeilleur_leplusbeau/source/admin.h
lemeilleur_leplusbeau/source/client.c
lemeilleur_leplusbeau/source/client.h
lemeilleur_leplusbeau/source/ihm.c
lemeilleur_leplusbeau/source/ihm.h
lemeilleur_leplusbeau/source/Makefile
lemeilleur_leplusbeau/config/conf_adm
lemeilleur_leplusbeau/config/conf_cl1
lemeilleur_leplusbeau/config/conf_cl2
lemeilleur_leplusbeau/config/conf_cl3
lemeilleur_leplusbeau/doc/README
lemeilleur_leplusbeau/doc/projet.pdf
```

Veillez à remplacer `lemeilleur` et `leplusbeaux` par vos noms. Ce ne sera pas à vos futurs clients d'organiser les documents que vous leurs enverrez.

23.7 Améliorations possibles

Attention avant de vous lancer dans d'éventuelles améliorations, veillez à être en possession d'un projet fonctionnel.

- La première amélioration consiste naturellement à étendre le jeu de commandes disponibles depuis l'interface.
- Une deuxième amélioration consisterait à permettre la navigation dans des sous répertoires des différents clients.
- Une autre amélioration serait de protéger certains fichiers afin d'interdire leur accès distant. Un fichier `.praccess` situé à la racine de chaque répertoire pourrait contenir la liste des fichiers accessibles (politique du refus par défaut).
- Une amélioration de grande envergure serait de distribuer ce projet sur plusieurs machines en utilisant des *sockets*.

23.8 Aide mémoire

Lecture des entrées d'un catalogue

- `DIRP *opendir(const char *pathname)` : cette fonction permet d'ouvrir le répertoire `pathname` et d'associer un descripteur de répertoire. Elle agit sur un répertoire un peu de la même façon que la fonction `open()` agit avec les fichiers. On peut alors utiliser le descripteur retourné pour lire les différentes entrées présentes dans le répertoire.
- `struct dirent *readdir(DIRP *fd)` : cette fonction lit l'entrée courante et pointe sur la suivante. Si il n'y plus d'entrée, la fonction retourne `NULL`. Les données lues sont placées dans une structure `dirent`. Deux champs de cette structure nous intéressent :
  - `d_type` : ce champ contient le type de l'entrée, *i.e.* s'il s'agit d'un répertoire ou d'un fichier par exemple.
  - `d_name` : ce champ contient le nom de l'entrée, *i.e.* le nom du fichier ou du répertoire.
- `int closedir(DIRP *fd)` : cette fonction ferme le descripteur de répertoire.

Un répertoire est donc vu comme un fichier contenant différentes entrées qui seront dans notre cas :

- d'autres répertoires (type `DT_DIR`)
- des fichiers (type `DT_REG`)

L'extrait suivant donne un exemple de l'utilisation de ces fonctions pour lister le contenu d'un répertoire. La fonction `contenu_repertoire` prend en arguments :

- `char *rootdir` : une chaîne de caractères donnant le chemin **absolu** du répertoire dont on veut obtenir le contenu,
- `int *length` : un pointeur vers un entier (cet entier doit exister) qui contiendra la longueur de la chaîne allouée par la fonction pour retourner le contenu du répertoire.

Elle renvoie une chaîne de caractères dont la longueur est écrite dans `length`, cette chaîne étant allouée par la fonction.

```
char *contenu_repertoire(char *rootdir, int *length)
{
  DIR *dirp;
  struct dirent *nd;
  char *retval=NULL;

  if (rootdir == NULL || length == NULL) return NULL;
  *length = 0;
  if (rootdir[0] != '/' ) return NULL;
  if (strstr(rootdir,"..") != NULL) return NULL;

  if ((dirp = opendir(rootdir)) == NULL) {
    return NULL;
  }
  while((nd = readdir(dirp)) != NULL) {
    if (nd->d_type == DT_REG || nd->d_type == DT_DIR) {
      if (strncmp(nd->d_name, ".", 1) && strncmp(nd->d_name, "..", 2)) {
```

```
        retval = realloc(retval,(retval==NULL?0:strlen(retval))+
                         strlen(nd->d_name)+2);
        strncat(retval,nd->d_name,NAME_MAX);
        strcat(retval,"\n");
      }
    }
  }
  closedir(dirp);
  *length = strlen(retval) + 1;
  return retval;
}
```

Ecriture d'un « paquet » de données

La fonction qui suit permet d'envoyer des données sur un flux, en organisant les données au sein d'un paquet. La fonction prend en arguments :

- `FILE *flux` : le descripteur de fichier vers lequel réaliser l'écriture,
- `int mode` : le mode d'envoi dont il est question, *i.e.* une réponse, un envoi ou une erreur,
- `int size` : le nombre d'éléments contenus dans le tableau `data`,
- `unsigned char *data` : l'adresse du tableau de données

La fonction retourne `0` quand l'écriture a eu lieu, et un entier non nul en cas d'erreur.

```
int write_paquet(FILE *flux, int mode, int size, unsigned char *data)
{
  unsigned char entete[10];
  size_t twri;
  /* Verification flux et donnees */
  if (flux == NULL || data == NULL) return -1;
  /* Verification du mode */
  if (mode < 0 || mode > 2) return -2;
  /* Verification de la taille */
  if (size <= 0) return -3;
  /* Creer l'entete */
  snprintf((char *)entete,10,"%c%8d",mode+'0',size); entete[9] = '\0';
  if ((twri = fwrite((void *)entete,1,9,flux)) != 9) return -4;
  if ((twri = fwrite((void *)data,size,1,flux)) != 1) return -5;
  fflush(flux);
  return 0;
}
```

Lecture d'un « paquet » de données

Le court extrait qui suit est un exemple de fonction permettant de lire, en l'analysant, un « paquet » sur un flux.

La fonction prend comme arguments :

- `FILE *flux` : le descripteur de fichier à partir duquel réaliser la lecture,
- `int *size` : l'adresse d'un entier (qui doit exister) dans lequel la fonction placera le nombre d'octets occupés par les données,
- `int *type` : l'adresse d'un entier (qui doit exister) dans lequel la fonction placera le type d'envoi (0,1 ou 2).

La fonction retourne l'adresse d'un tableau de caractères alloué et contenant les données. Ce tableau possède une taille égale au nombre d'octets de données augmenté de un. Le dernier élément du tableau est le caractère nul `\0`. Le fait de placer ce caractère spécial permet de lire ce tableau de caractères comme une « chaîne », et donc d'utiliser des fonctions telles `strncmp()` ou encore `sscanf()`.

```
unsigned char *read_paquet(FILE *flux, int *size, int *type)
{
  unsigned char *data=NULL;
  unsigned char buf[9];
  size_t tread;
  if (flux == NULL || size == NULL || type == NULL) {
    return NULL;
  }
  /* Lecture du mode*/
  if ((tread = fread(buf,1,1,flux)) != 1) {
    return NULL;
  }
  if (buf[0] == 0 || buf[0] == '0') *type = 0;
  else if (buf[0] == 1 || buf[0] == '1') *type = 1;
  else if (buf[0] == 2 || buf[0] == '2') *type = 2;
  else return NULL;

  /* Lecture de la taille */
  if ((tread = fread(buf,8,1,flux)) != 1) return NULL;
  buf[8] = '\0'; *size = atoi((char *)buf);if (*size <=0) return NULL;
  /* Lecture des donnees */
  if ((data = malloc(*size+1)) == NULL) {
    *size = 0; return NULL;
  }
  if ((tread = fread(data,1,*size,flux)) != *size) {
    free(data); *size = 0; return NULL;
  }
  data[*size] = '\0';
  return data;
}
```

Lecture binaire d'un fichier

Les fichiers à transmettre doivent être lus non pas ligne à ligne mais de manière binaire. La fonction suivante permet de lire la totalité d'un fichier et de placer les données lues dans un tableau de caractères.

Elle prend comme arguments :

- `char *name` : le chemin associé au fichier
- `int *size` : l'adresse d'un entier (qui doit exister) dans lequel la fonction placera le nombre d'octets occupés par les données,

La fonction retourne un tableau de caractères dont la taille est égale au nombre d'octets lus augmenté de un. Le dernier élément du tableau est le caractère nul `\0`.

```
unsigned char *read_fichier(char *name, int *size)
{
  FILE *flux;
  struct stat buf;
  unsigned char *data;
  size_t tread;
```

```
  if (name == NULL || size == NULL) return NULL;
  if (stat(name,&buf)) return NULL;
  *size = (int)(buf.st_size);
  if (*size == 0) return NULL;
  if ((flux = fopen(name,"r")) == NULL) {
    *size = 0; return NULL;
  }
  if ((data = malloc(*size+1)) == NULL) {
    *size = 0; return NULL;
  }
  if ((tread = fread(data,1,*size,flux)) != *size) {
    free(data); *size = 0; return NULL;
  }
  fclose(flux); data[*size] = '\0';
  return data;
}
```

Séparation d'une commande de l'`Interface`

La fonction qui suit permet de séparer les différents champs d'une commande de type `machine@command`. Attention, cette fonction modifie la chaîne qu'on lui passe en entrée.

Elle prend pour arguments :

- `char *cmd_in` : la chaîne de caractères à analyser qui sera modifiée,
- `char **cmd_out` : l'adresse d'un tableau de caractères qui pointera vers la sous-chaîne correspondant à la commande,
- `char **machine` : l'adresse d'un tableau de caractères qui pointera vers la sous-chaîne de la machine.

La fonction renvoie 0 si tout se passe bien et 1 dans le cas contraire.

```
#define MAX_STRING_LENGTH 256
int split_commande(char *cmd_in, char **cmd_out, char **machine)
{
  int s;
  if (cmd_in == NULL || cmd_out == NULL || machine == NULL) return -1;
  if (strlen(cmd_in) >= MAX_STRING_LENGTH) return -1;
  *machine = cmd_in; s = 0;
  while(cmd_in[s] != '@' && s<strlen(cmd_in)) s++;
  if (s==strlen(cmd_in)) return -1;
  cmd_in[s] = '\0'; *cmd_out = cmd_in + s + 1;
  return 0;
}
...
char cmd[256];
char *cmd_out=NULL, *machine=NULL;

if (split_commande(cmd, &cmd_out, &machine)) {
   fprintf(stderr,"Erreur\n");
}
...
```

Séparation des arguments au sein d'un paquet

On suppose avoir récupéré le contenu d'un paquet et donc une chaîne de caractères représentant une commande. Nous traiterons ici de la commande `write` qui est constituée comme suit :

```
write nom_file size data
```

Nous devons donc séparer les arguments d'une chaîne de la forme :

```
write test.txt 26 abcdefghijklmnopqrstuvwxyz!
```

Pour mémoire, le paquet intégrant cette chaîne serait de la forme suivante :

```
2      45write test.txt 26 abcdefghijklmnopqrstuvwxyz!
```

La fonction prend en arguments :

- `unsigned char *data` : les données issues de la lecture du paquet par la fonction `read_paquet()`,
- `char *filename` : l'adresse d'un tableau de caractères dont la longueur doit être au moins égale à une constante macro-définie (comme on a pu le voir dans la fonction précédente),
- `int *datasize` : l'adresse d'un entier (qui doit exister) dans lequel la fonction placera la taille du tableau de données qu'elle retournera.

La fonction renvoie l'adresse d'un tableau d'octets qui contiendra les données destinées à être écrites dans le fichier `filename`.

```
unsigned char *split_command_write(unsigned char *data,
                                   char *filename,
                                   int *datasize)
{
  int s,i,l;
  unsigned char *wdata=NULL;
  if (data == NULL || filename == NULL || datasize == NULL) {
     return NULL;
  }
  *datasize = -1;
  s = sscanf((char *)data,"write %s %d",filename,datasize);
  if (s !=2) {
    *datasize = 0;
    return NULL;
  }
  if (*datasize<=0) {
    *datasize = 0;
    return NULL;
  }
  if ((wdata = malloc(*datasize)) == NULL) {
    *datasize = 0;
    return NULL;
  }
  l = strlen((char *)data)-*datasize;
  for(i=0;i<*datasize;i++) {
    wdata[i] = data[l+i];
  }
  return wdata;
```

```
}

/* Exemple d'utilisation */
  FILE *fp;
  char filename[256];
  int psize, ptype, datasize;
  pdata = read_paquet(stdin, &psize, &ptype);
  ...
  wdata = split_command_write(pdata, filename, &datasize);
  if ((fp = fopen(filename,"w+")) == NULL) {
   ....
  }
  if (fwrite((void *)wdata,datasize,1,fp) != 1) {
    ....
  }
  fclose(fp);
  free(wdata);
```

24

Parcours multi-agents dans un labyrinthe

Ce projet a pour but d'introduire la notion de système multi-agents sur un exemple simple. Un système multi-agents, dans l'acceptation qui nous intéresse, est une technique de résolution de problème complexe qui permet d'explorer un ensemble de solutions (optimum local) beaucoup plus large que ne pourrait le faire un système séquentiel simple.

Largement utilisée en traitement d'images, cette technique permet par exemple de réaliser des tâches d'analyse de scène en découpant le problème en sous-tâches plus simples et en laissant le soin à des programmes très spécialisés (et donc relativement efficaces) de résoudre ces « mini » problèmes. On peut ainsi trouver un agent qui va estimer la nature du bruit présent dans certaines parties de l'image et qui laissera, en fonction du résultat obtenu (gaussien, multiplicatif, additif), un autre (d'autres) agent(s) procéder au débruitage.

Cette population d'agents doit naturellement posséder le moyen de communiquer et surtout de s'organiser. Un des processus fondamentaux d'un système multi-agents est le contrôleur qui a la charge de **créer**, de **dialoguer** et de **tuer** les agents.

Nous allons mettre en œuvre un système multi-agents simple pour résoudre la découverte d'un chemin dans un labyrinthe. Une technique simple pour obtenir la solution est d'utiliser un programme récursif. À chaque embranchement, un tel programme appelle la fonction de résolution de chemin et si celle-ci débouche sur une impasse, l'appel précédent poursuit sa marche dans l'autre branche. Cette technique est tout à fait correcte mais au lieu d'utiliser un seul processus pour avancer dans le labyrinthe, nous allons demander à un contrôleur de gérer une population d'agents qui progresseront dans les différents chemins [1].

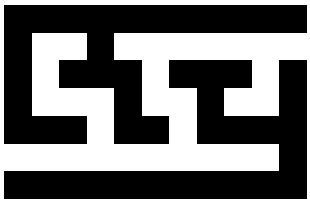
1. Si l'on effectue le parallèle avec les parcours d'arbre, la technique récursive est un parcours en profondeur d'abord, la technique par agent un parcours en largeur d'abord.

24.1 Position du problème

Définitions liées au labyrinthe

Un labyrinthe peut se présenter sous différentes formes. Nous n'utiliserons que des labyrinthes simples sans boucle et pour lesquels il n'existe qu'une seule solution. Les chemins présents dans ces labyrinthe seront de largeur égale à un. Toutefois, les curieux et les curieuses pourront « lâcher » leurs agents dans des labyrinthes plus complexes.

Nos labyrinthes seront des tableaux à deux dimensions dans lesquels chaque case sera représentée par le caractère [2] « 1 » ou « 0 » selon qu'il s'agit d'un mur ou d'une case libre. Par convention, **et pour l'intégralité du projet**, nous conviendrons qu'une case à « 0 » est une case libre, alors qu'une case de valeur « 1 » est un mur. La figure 24.1 donne un exemple de labyrinthe. **Dans la suite de ce document, les symboles « 0 », « 1 » et « 2 » désignent les caractères `'0'`, `'1'` et `'2'`.**



1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1	1	0	1
1	0	0	0	1	0	0	1	0	0	1
1	1	1	0	1	1	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1

FIGURE 24.1 – *Une représentation sous forme d'« image » d'un labyrinthe. Les chemins sont représentés en blanc. Le fichier contient pour sa part les caractères `'0'` et `'1'`.*

Les labyrinthes utilisés seront dits « 4-connexes ». Les chemins définis à l'intérieur de tels labyrinthes ne peuvent suivre que les quatre directions haut, bas, gauche et droite. Tout parcours diagonal est donc impossible. La figure 24.2 détaille les différents chemins. Enfin, pour nous concentrer sur la gestion du système multi-agents, nous supposerons que l'entrée du labyrinthe se trouve sur le mur extérieur de gauche (mur ouest). Par souci de simplification, vous pourrez aussi supposer si cela vous arrange que la sortie se trouve sur le mur extérieur de droite (mur est).

La répartition des tâches

Le système multi-agents est composé de trois entités différentes :

– le contrôleur : il connaît le labyrinthe et organise la vie des agents. Il a la charge de répondre à leurs demandes. Il obtiendra la solution du labyrinthe en remontant

2. On rappelle que dans le contexte du codage ASCII, un caractère est en fait représenté dans la machine sous la forme d'une valeur entière, ainsi le caractère `'0'` (zéro) a la valeur 49 et le caractère `'1'` la valeur 50.

FIGURE 24.2 – *Les flèches sur la figure de gauche représentent les chemins possibles pour certains embranchements. Toute traversée diagonale est interdite car elle brise la 4-connexité. Ces traversées diagonales interdites sont représentées sur la figure de droite.*

le chemin suivi par l'agent arrivé sur la case de sortie. Il tient à jour la progression des agents. C'est ce que représente la figure 24.8.

– l'agent : il n'a du labyrinthe qu'une connaissance très restreinte, à savoir son environnement proche qui se résume aux huit cases qui l'entourent. L'agent sait aussi d'où il vient, *i.e.* la case où il était au tour d'avant. L'agent ne peut pas rebrousser chemin, il doit toujours avancer. À partir de son environnement il choisit le chemin qu'il suit, que celui-ci soit unique ou qu'il y ait plusieurs possibilités. Il dialogue avec le contrôleur pour compléter sa vision du labyrinthe et aussi pour donner son déplacement.
– l'interface : elle a en charge de transmettre initialement au contrôleur le labyrinthe et de recevoir de ce dernier les informations concernant la progression des différents agents. Cette interface doit de plus permettre un fonctionnement « pas à pas » du système. Elle dialogue donc avec le contrôleur et ce dialogue bloquant doit permettre de synchroniser le contrôleur et les agents. On pourrait très bien utiliser les signaux pour réaliser cette synchronisation, mais on préfère, dans le cadre de ce projet retenir une solution de lecture bloquante.

Les échanges dans le système

Les dialogues entre les agents et le contrôleur sont tous bidirectionnels : l'agent requiert des informations et il communique sa position, son intention de mourir ou une demande de création lors de la découverte d'un embranchement. Le contrôleur envoie la nature des cases que les agents découvrent au fur et à mesure de leurs progressions.

Afin d'avoir un système modulaire, nous devons figer les formats d'échange qui seront utilisés. Voici les recommandations qu'il faudra suivre.

Dialogue Contrôleur $\longleftrightarrow$ Interface

L'interface est le processus qui se chargera de lancer tous les autres. L'interface doit donc transmettre au processus contrôleur le labyrinthe dans son intégralité. Cette

transmission ne sera pas faite sur la ligne de commande ni par un simple envoi d'un nom de fichier que le contrôleur irait ensuite charger. Le labyrinthe, *i.e.* sa largeur, sa hauteur et les caractères qui le composent seront envoyés via un tuyau sur l'entrée standard (`stdin`) du contrôleur. Ainsi la première tâche du contrôleur sera-t-elle de récupérer cette structure. Cet envoi prendra donc la forme suivante :

```
largeur\n
hauteur\n
111111111\n
..0011...\n
.........\n
111111111\n
```

Lors d'une avance pas à pas, l'interface envoie un message pour débloquer le contrôleur (nous verrons ceci plus en détail dans le paragraphe lié à l'organisation). Ce message prendra la forme suivante : « `OK\n` ». Le contrôleur qui était auparavant bloqué par un appel de lecture dans un tuyau vide, va pouvoir ainsi passer à l'étape de gestion de ses fils.

Le contrôleur doit rendre compte à l'interface et donc envoyer, après chaque étape de gestion de ses agents, les identifiants (typiquement le `pid` et un numéro d'ordre de création) de ses agents ainsi que leur position. Les positions sont numérotées de gauche à droite pour les `x` et de haut en bas pour les `y`. Le numéro d'agent sera pris à 0 pour l'agent initial, et chaque nouvel agent prendra l'entier immédiatement supérieur. La structure du message sera la suivante :

```
AGT pid_agent numero_agent x_agent y_agent\n
```

Pour signifier que tous les agents ont été transmis, le contrôleur signale à l'interface qu'il passe en mode d'écoute des agents par le message `RUN`. On obtient donc une suite de messages de la forme :

```
AGT pid_agent_1 numero_agent_1 x_agent_1 y_agent_1\n
AGT pid_agent_2 numero_agent_2 x_agent_2 y_agent_2\n
RUN\n
```

Dialogue Agent $\longrightarrow$ Contrôleur

Un agent, une fois qu'il connaît son environnement 3×3 peut choisir quel chemin suivre. Il annonce alors au contrôleur son choix tout en faisant une demande de nouvel environnement :

- pour un déplacement vers l'ouest : `HO`
- pour un déplacement vers l'est : `HE`
- pour un déplacement vers le sud : `HS`
- pour un déplacement vers le nord : `HN`

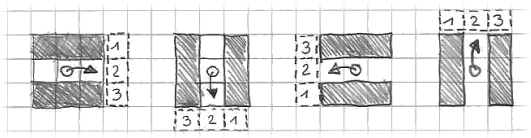


FIGURE 24.3 – *Les quatre demandes d'environnement. Lorsque l'agent choisit de se déplacer, un tiers de son nouvel environnement lui fait défaut. Selon son déplacement, il réclame donc au contrôleur les trois cases qui lui manquent. Cette demande permet en outre au contrôleur de connaître le chemin suivi par l'agent.*

Si plusieurs chemins sont possibles (l'agent atteint une bifurcation ou un croisement), l'agent annonce son choix et transmet au contrôleur les autres chemins. La requête prend alors les formes suivantes :

- deux choix sont possibles : `BXY` avec `X` donnant la direction choisie par l'agent, soit `O`, `E`, `N` ou `S` et `Y` la direction laissée. Le contrôleur devra réagir en créant un nouvel agent qui prendra le chemin `Y`.
- trois choix sont possibles : `CXYZ`, `X` donnant la direction choisie par l'agent et `Y` et `Z` les directions laissées sous le contrôle du contrôleur pour la création de deux agents.

Si un agent arrive dans une impasse, *i.e.* il n'a plus aucune solution pour avancer, il doit signaler son envie pressante de mourir. Le message qu'il envoie prend la forme suivante : `T`.

L'agent qui reçoit l'environnement `222` a atteint la sortie (voir ci-dessous pour la définition de l'environnement). Il envoie alors au contrôleur un message de réussite. Ce message prend la forme suivante : `F`.

Dialogue Contrôleur $\longrightarrow$ Agent

Le contrôleur envoie aux agents lorsqu'ils en font la demande l'environnement que ces derniers doivent connaître. Cet environnement se limite aux trois cases qui jouxtent la nouvelle position comme le montre la figure 24.3. **Attention, les trois cases sont envoyées dans le sens horaire**. Ainsi pour une demande d'environnement est, sont envoyées les valeurs des cases nord-est, est puis sud-est. Pour une demande d'environnement ouest, sont envoyées les cases sud-ouest, ouest puis nord-ouest. Si une ou plusieurs des cases sortent du labyrinthe, le contrôleur retourne la valeur « 2 », dans les autres cas le contrôleur retourne la valeur de la case, à savoir 0 si la case est libre et 1 s'il s'agit d'un mur. Un cas typique pourrait être par exemple pour un

environnement est se prolongeant en ligne droite : `101`. Pour la case de sortie par exemple, l'environnement sera obligatoirement `222`.

Le message d'environnement prendra la forme suivante : `Habc`, où `a`, `b` et `c` prennent une des trois valeurs 0, 1 ou 2.

Le contrôleur peut aussi envoyer des messages de terminaison aux agents qui vivent encore lorsque l'un d'entre eux a trouvé la sortie. Ce message prendra la forme suivante : `K`.

La création d'un agent n'occasionnera pas d'envoi de message, l'environnement 3×3 complet étant passé sur la ligne de commande de l'agent. Ceci sera abordé ultérieurement.

Récapitulatif des messages

Ce paragraphe résume les différents messages échangés. Les messages (excepté l'envoi du labyrinthe qui est sur trois lignes) seront tous sous la forme d'une ligne de texte terminée par un `\n` [3].

Agent vers Contrôleur

Déplacement (et demande d'environnement)			
ouest	est	nord	sud
`HO`	`HE`	`HN`	`HS`
Accès à la case de sortie			
`F`			
Arrivée dans une impasse			
`T`			
Arrivée à une bifurcation			
`BOE`	`BEO`	`BNS`	...
Arrivée à un croisement			
`CNES`	`CONE`	`CNOS`	...

Contrôleur vers Agent

Réponse d'environnement			
ouest	est	nord	sud
`H???`	`H???`	`H???`	`H???`
`?` étant le caractère « 0 », « 1 » ou « 2 »			
Envoi de la requête de terminaison			
`K`			

3. Ainsi vous pourrez tester les différentes briques (agent puis contrôleur) en interagissant directement avec eux au clavier.

Interface ↔ Contrôleur
Envoi du labyrinthe (I→C) `L\nH\n11..\n..1\n`
Avance pas à pas (I→C) `OK`
Arrêt immédiat (I→C) `FIN`

Progression (C→I)
`AGT pid num x y`
...
`RUN`

Solution (C→I) `SOL x y`

24.2 L'organisation générale

La figure 24.4 donne une vue d'ensemble de l'organisation des différents processus et des communications qui prennent place.

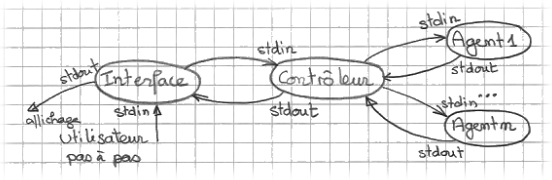


FIGURE 24.4 – *Les échanges au sein du système.*

Processus

Comme le suggère son nom, le contrôleur a une place centrale. C'est lui qui a la charge de créer les agents au fur et mesure du déroulement de la recherche de solution et de communiquer les déplacements à l'interface, il joue donc le rôle de chef d'orchestre. Néanmoins, il sera créé en tant que fils du processus interface afin que le processus interface puisse réaliser ses affichages sur la console [4]

Le processus interface recevra sur sa ligne de commande le nom du fichier contenant le labyrinthe ainsi que la sélection du déroulement (mode pas à pas ou mode

4. Cela pourra permettre également de remplacer l'interface simple que vous allez développer par un équivalent graphique sans nécessiter d'autre modification. Une telle interface ne fait pas partie de ce projet.

automatique). Il créera le processus contrôleur en lui transmettant sur sa ligne de commande le type de mode choisi (pas à pas ou automatique).

Le processus contrôleur devra impérativement réceptionner le labyrinthe sur son entrée standard avant de créer le premier agent. La création de l'agent est abordée ci-après. Une fois l'agent créé, et selon le mode de fonctionnement, le processus contrôleur entamera deux boucles de gestion. La création du premier agent ne rentre pas dans la boucle de gestion des agents.

- Gestion des agents
  1 le contrôleur lit les requêtes des agents existants.
  2 le contrôleur répond à ces requêtes et crée si besoin est de nouveaux agents. Il met à jour la position des agents.
- Dialogue avec l'interface
  1 le contrôleur envoie à l'interface le `pid` et la position des différents agents. Il signale à l'interface que la position du dernier agent a été envoyée par le message `RUN`.
  2 le contrôleur lit la requête de l'interface et prend les mesures qui s'imposent (uniquement en mode pas à pas).

Chaque agent sera un fils du contrôleur. L'agent adopte toujours le même schéma de fonctionnement, à savoir la réception de son environnement et de la direction de démarrage puis une boucle de gestion organisée comme suit :

1 l'agent décide son chemin,
2 l'agent envoie sa requête au contrôleur (il reste bloqué tant que le contrôleur ne la lit pas),
3 l'agent lit la réponse du contrôleur puis il revient à l'étape 1.

Les dialogues se font entre le contrôleur et les agents comme le symbolise la figure 24.5. Lorsque le contrôleur crée un agent, il lui fait parvenir sur la ligne de commande l'ensemble des paramètres dont il a besoin, à savoir l'environnement 3×3 complet et une direction de recherche [5]. Le programme « agent » aura donc une ligne de commande de la forme : « `> agent 111000101 E` ».

L'environnement prendra la forme d'une chaîne de 9 caractères. Le premier agent créé recevra par principe l'environnement suivant : `221200221` ce qui représente l'entrée du labyrinthe située comme nous l'avons précisé sur le mur ouest. L'environnement sera constitué par les trois lignes lues de gauche à droite et de bas en haut de la fenêtre 3×3 entourant l'agent. La figure 24.6 donne un exemple de création d'agent et de la première chaîne de caractères passée en ligne de commande.

Pour simplifier le développement, il sera possible de transmettre un environnement différent du labyrinthe en interdisant (par le caractère `'2'` certaines cases afin de contraindre le démarrage de l'agent dans une direction bien précise. Dans ce cas, transmettre la direction sur la ligne de commande ne sera plus nécessaire.

5. Cette direction sera un caractère `'N'`=nord, `'E'`=est, `'S'`=sud, `'O'`=ouest

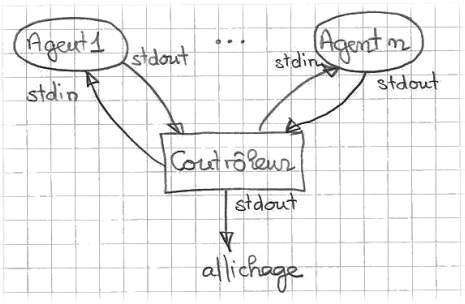


FIGURE 24.5 – *Chaque agent reçoit des informations sur son entrée standard et émet des requêtes sur sa sortie standard.*

Recherche du chemin par l'agent

À sa création l'agent a reçu son environnement ainsi qu'une direction de recherche. Le labyrinthe étant construit en 4-connexité, un agent a au plus quatre possibilités de déplacement. La direction qu'il reçoit à sa création lui bloque une de ces possibilités, celle qui le ferait revenir en arrière. Il lui reste donc trois possibilités :

- avancer dans la même direction, *i.e.* aller « tout droit »
- tourner à gauche
- tourner à droite.

La figure 24.7 résume la technique de recherche de l'agent.

Dialogue

Le dialogue entre le contrôleur et un processus fils se fera au moyen d'une paire de tuyaux. Les agents liront les messages sur leur entrée standard (`stdin`) et enverront les requêtes sur leur sortie standard (`stdout`). Il faudra donc veiller à rediriger les entrées/sorties des processus agents avant de les créer.

Les agents doivent progresser et lire leur entrée standard. Le fait de rendre la lecture bloquante permet de synchroniser tous les agents.

Toute demande de l'agent doit recevoir une réponse, l'agent restant bloqué tant qu'il n'a pas cette réponse. Si le contrôleur répond par le message `K`, l'agent doit se terminer immédiatement et le contrôleur surveillera la bonne terminaison de son fils.

Affichage

Le contrôleur envoie les différentes positions des agents à l'interface. Celle-ci affichera le labyrinthe entier avec la position des agents sur sa sortie standard (donc ici la console). Afin de rendre l'affichage lisible, on se fixe le format suivant :

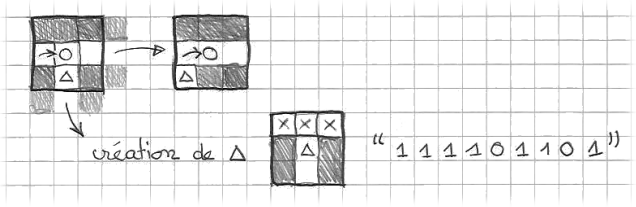


FIGURE 24.6 – *Demande de création d'agent. Sur la figure de gauche l'agent se déplace vers l'est. Il arrive donc dans la case où figure un cercle. En découvrant son environnement, il peut prendre deux directions, continuer vers l'est ou descendre vers le sud. En choisissant de continuer vers l'est il signale en plus au contrôleur qu'il faut créer un agent partant de la case marquée d'un triangle (message envoyé* `BES`*). Le contrôleur transmet à ce nouvel agent l'environnement complet qui l'entoure ainsi que la direction dans laquelle il doit essayer de progresser. Il bloque naturellement les cases que l'agent ne peut pas prendre soit en raison d'un chemin inverse, soit car d'autres agents prendront ces directions lors d'une création multiple en raison d'une bifurcation ou d'un croisement. La chaîne passée sur la ligne de commande de ce nouvel agent sera donc celle représentée sur la droite, soit* `222101101`*. En plus de cette chaîne le contrôleur transmet la direction sud soit le caractère* `'S'`*. Le nouvel agent, après réception de ces deux données n'a qu'une seule solution de déplacement, celle d'aller vers le sud.*

- une case mur sera affichée par un caractère dièse #
- une case libre sera affichée par une espace
- un agent sera rendu par un chiffre représentant son numéro. Afin de rester sur un caractère, les numéros à partir de 10 seront représentés par les lettres de l'alphabet minuscule.

Il faudra bien entendu s'assurer que la console utilise une police dont tous les caractères font la même largeur (police de type *fixed*).

Le chemin solution

Si le contrôleur note pour chaque déplacement des agents le chemin suivi avec un code dans une carte d'étiquettes, il peut, en partant de l'agent ayant atteint la case de sortie, revenir pas à pas en arrière et obtenir ainsi l'intégralité du chemin solution du labyrinthe. C'est ce que montre la figure 24.8.

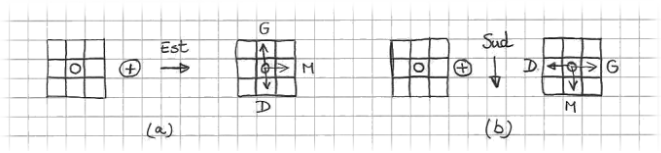


FIGURE 24.7 – *L'agent ne possède que 4 directions possibles pour se déplacer dans un environnement 4-connexe. Dans le cas de la figure (a), ayant reçu la direction est il ne peut aller que vers le nord, l'est ou le sud. Dans le cas de la figure (b), les seuls choix qui lui restent sont l'est, le sud ou l'ouest. Il doit ensuite, à partir de ces trois directions en choisir une, choix qui dépend bien sûr des cases (libres ou murées) du labyrinthe.*

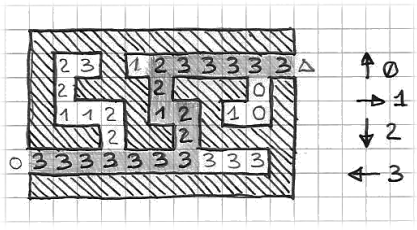


FIGURE 24.8 – *Lorsqu'un agent atteint la case de sortie, il suffit d'inverser le code de direction pour remonter le trajet jusqu'à la case de départ. Sur le schéma sont représentés les codes inversés qui permettent de remonter de proche en proche. Ce schéma sera construit et conservé par le contrôleur au fur et à mesure de la progression des agents dans le labyrinthe.*

24.3 Mise en œuvre du projet

Les fonctions disponibles

Afin de vous éviter un travail fastidieux (mais pourtant à votre portée !), certaines fonctions de lecture de message vous sont fournies à titre gracieux et officiel sous la forme d'un fichier d'en-tête `fonction.h` et d'un fichier objet `fonction.o` correspondant.

Structure `Labyrinthe`

```
struct _labyrinthe {
  int largeur,hauteur;
```

```
  char *carte;
};
typedef struct _labyrinthe Labyrinthe;
```

Cette structure définit un labyrinthe comme étant composé de 2 entiers et d'un tableau monodimensionnel donnant la carte. On accède à la valeur de la case de coordonnées `x,y` en utilisant par exemple :

```
#define CASE_LAB(L,x,y) (*((L)->carte+(y)*(L)->largeur+(x)))
```

Enumération `Direction`

```
enum _direction_lab {N=0,E=1,S=2,O=3};
typedef enum _direction_lab Direction;
```

`char2dir()`

```
Direction char2dir(char dir)
```

Cette fonction convertit le caractère de direction `dir` en entier, soit `N` $\rightarrow 0$, `E` $\rightarrow 1$, `S` $\rightarrow 2$ et `O` $\rightarrow 3$.Si la direction n'est pas correcte, la fonction retourne -1.

`lab_alloc()`

```
Labyrinthe *lab_alloc(int l, int h)
```

Cette fonction alloue une structure `Labyrinthe` de largeur `l` et de hauteur `h`. Elle retourne l'adresse de cette structure ou l'adresse `NULL` en cas d'erreur.

`lab_free()`

```
Labyrinthe *lab_free(Labyrinthe *L)
```

Cette fonction libère la mémoire associée à l'adresse `L`. Cette fonction retourne toujours une adresse `NULL`.

`lab_load()`

```
Labyrinthe *lab_read(FILE *fp)
```

Cette fonction retourne l'adresse d'une structure `Labyrinthe` dûment allouée et contenant le labyrinthe (ensemble de case de valeur 0 ou 1 lu dans le flux `fp`. En cas d'échec la fonction retourne une adresse `NULL`. Le fichier est organisé comme suit :

```
Largeur\n
Hauteur\n
111111111111\n
011001110011\n
011001110011\n
.....
111111111111\n
```

lab_write()

```
int lab_write(FILE *fp, Labyrinthe *L)
```

Cette fonction permet à l'interface d'écrire dans le flux `fp` le labyrinthe `L`. Cette fonction est compatible avec la lecture de la fonction précédente.

agent_write_mesg()

```
int agent_write_mesg(char *mesg, FILE *fp)
```

Cette fonction se contente d'écrit dans le flux `fp` le message `mesg` en lui rajoutant un retour chariot (`\n`) puis force l'écriture du tampon associé à `fp`. La fonction renvoie 0 si tout s'est bien déroulé et un code non nul dans le cas contraire.

agent_read_mesg()

```
int agent_read_mesg(char *en, int dir, FILE *fp)
```

Cette fonction lit un message (en provenance du contrôleur) dans le flux `fp`. Selon la nature du message, la fonction met à jour le tableau de 9 caractères (minimum) `env` en lisant 3 caractères sur `fp` dans le bon sens grâce à la direction `dir`. Si le message du contrôleur est de type `'H???'` la fonction retourne 1 après avoir fait les mises à jour si tout s'est bien passé. Sinon elle retourne un code négatif.
Si le message du contrôleur est de type `'K'` la fonction retourne immédiatement le code `0` (zéro).
Cette fonction est parfaitement compatible avec la fonction qui suit.

ctrl_write_env()

```
int ctrl_write_env(Labyrinthe *L,
                   int x, int y,
                   int dir, FILE *fp)
```

Cette fonction permet au contrôleur d'écrire dans le flux `fp` la lettre `'H'` suivie des trois cases (dans le sens horaire) qui compose l'environnement à découvrir dans la direction `dir` lorsque l'on est au point de coordonnées `x,y` du labyrinthe `L`.
Il s'agit donc

– des cases `(x-1,y-1)`, `(x,y-1)` et `(x+1,y-1)` pour la direction nord,
– des cases `(x+1,y-1)`, `(x+1,y)` et `(x+1,y+1)` pour la direction est,
– des cases `(x+1,y+1)`, `(x,y+1)` et `(x-1,y+1)` pour la direction sud,
– des cases `(x-1,y+1)`, `(x-1,y)` et `(x-1,y-1)` pour la direction ouest,

Livrables et modalités

Le projet doit être rendu avant l'heure qui sera indiquée sur le site consacré à l'IN201

`http://www.ensta.fr/ bcollin`

Chaque jour de retard entraîne une pénalité selon le barême qui suit :

1 jour	2 jours	3 jour	4 jours	5 jours	6 jours et au-delà
1 point	2 points	4 points	8 points	16 points	20 points

Le projet doit être envoyé par courrier électronique au MdC qui a encadré le groupe dans lequel vous vous trouvez. Cet envoi prendra la forme d'une archive au format `tar` laquelle sera composée des éléments suivants :

– Les programmes sources et en-têtes (`.c` et `.h`) amplement commentés,
– un fichier `Makefile` permettant de compiler vos fichiers et d'obtenir les trois exécutables (le contrôleur, l'interface et l'agent),
– un fichier `LISEZMOI` décrivant les fichiers qui composent l'archive et décrivant la procédure d'installation (quelle commande pour la compilation, quelle commande pour exécuter le projet,...). Ce fichier doit être au format texte.
– un rapport décrivant les différents programmes que vous aurez rédigés, les structures manipulées, l'organisation globale, les communications, les forces et faiblesses de votre projet, les différents tests que l'utilisateur pourra mettre en œuvre pour effectuer les vérifications de bon fonctionnement et des illustrations pouvant éclaicir vos propos. Ce rapport devra impérativement être remis dans un format lisible, à savoir `PDF`, `HTML` ou format texte seul.

On rappelle que l'étape de compilation est importante et qu'il serait souhaitable que vos programmes puissent compiler lorsque les options `-Wall -ansi -pedantic` sont retenues. Il sera peut-être utile d'utiliser la définition

```
#define _GNU_SOURCE
```

pour obtenir certaines définitions de prototypes des fonctions de la bibliothèque standard.

Développements subsidiaires

– utilisation de la fonction système `select()` pour réaliser les écoutes sur les différents tuyaux mis en œuvre afin de pouvoir rajouter un temps d'attente

aléatoire différent dans chaque agent et de vérifier que les agents réussissent à avancer (en mode automatique) sans se préoccuper de leurs congénères ;
– généralisation du système de base pour gérer des labyrinthes complexes à base de boucles et chemin de largeur supérieure à 1.

24.4 Conseils en vrac

Quelques conseils qui survivent d'année en année... Ne vous jetez pas dans le code, prenez le temps de réfléchir et d'organiser votre développement.

Ne vous lancez pas dans les développements subsidiaires avant d'avoir réalisé un projet de base en état de marche !

Évitez les programmes dont l'exécutable s'appelle `test`, `ls`, `rm` !

Chaque composant de ce projet peut être développé indépendamment des autres, c'est l'intérêt de fonctionner par duplication de l'entrée et de la sortie standard.

Méfiez-vous de la ressemblance troublante en le symbole `O` (la lettre O) et `0` (le chiffre 0).

Faites de petites fonctions ne sachant pas faire grande chose, mais le faisant bien, plutôt qu'une énorme fonction capable de tout mais fonctionnant à la manière d'un clignotant (un coup ça marche, un coup ça marche pas !). Ne pensez pas avoir trouvé une erreur dans `malloc` ou `free`, faites tout d'abord votre *mea culpa* avant l'*habeas corpus*. Ne pensez pas que l'on travaille mieux et plus vite en situation de stress intense, *i.e.* la veille du jour de remise du projet, d'autant que ce jour là tata Hortense va sûrement vous téléphoner pour prendre de vos nouvelles...

25

Course de voitures multi-*threads*

Résumé

Ce projet a pour but de découvrir la programmation par *threads* au travers de l'implémentation d'une course de voitures. À l'issue de votre travail vous devrez rendre un ensemble de fichiers sources ainsi qu'un rapport.

25.1 Présentation

Généralités

La programmation *multi-threads* offre la possibilité d'utiliser efficacement les architectures multi-processeurs ou multi-cœurs. Très souvent, dans une programmation « classique », les différents programmes qui échangent des informations (au moyen de *pipe* ou de *socket*) sont synchronisés par des mécanismes de lecture et d'écriture bloquants. Au sein des différents *threads* issus d'un processus, et comme cela a été vu en cours, l'accès à la mémoire virtuelle du processus est intégral (tout comme les accès aux descripteurs de fichiers). Il paraît donc assez naturel de se passer d'une communication externe et de faire en sorte que chaque *thread* écrive les différentes informations nécessaires au déroulement du programme dans une sorte de tableau noir partagé par tous (principe du *black board* dans un système multi-agents). Mais il devient impératif de protéger ces accès concurrents à la mémoire puisque nous ne disposerons pas (nous ne souhaitons pas disposer !) de la synchronisation par lecture/écriture.

Nous allons donc mettre en place une course de voitures dans laquelle chaque voiture sera animée par un *thread* tandis qu'un gestionnaire central aura la tâche d'afficher les positions de chaque voiture et d'effectuer certains contrôles.

Afin de garantir la portabilité de notre code source, nous utiliserons les *threads* `POSIX` (la bibliothèque `libpthread.so`) et nous veillerons à ce que nos programmes compilent sans avertissement avec la directive `-posix` doublée des directives `-ansi`, `-pedantic` et `-Wall`. Les annexes donnent, à ce sujet, quelques compléments d'informations nécessaires à une utilisation sous Linux [1].

1. Vous pouvez programmer votre projet chez vous sur votre ordinateur. Cependant votre évaluation sera réalisée sur les ordinateurs de l'ENSTA ParisTech et eux seuls.

Vue d'ensemble

La course se déroule sur une piste en boucle. Chaque voiture est en fait un *thread* dont le but est d'avancer sur la piste. Un $\mathtt{T_v}$ (les *threads* gérant les voitures seront dénommés ainsi) regarde si la piste devant lui est libre afin de gagner une nouvelle position. Il garde en mémoire sa nouvelle position et accède à la piste pour effacer son numéro de son ancienne position et s'inscrire dans sa nouvelle position. Selon les impératifs de course dictés par l'opérateur (en fait l'utilisateur au travers d'une interface), les $\mathtt{T_v}$ pourront réaliser d'autres actions. Ces dernières seront listées plus loin.

La vue d'ensemble de la course sera réalisée par un autre *thread*, appelé le $\mathtt{T_o}$. Ce dernier contrôle le déroulement de la course et interagit avec l'utilisateur par le biais d'une interface. Le contrôle de la course signifie que ce *thread* va réguler l'avancée des voitures. De plus, au travers de l'interface, l'utilisateur peut demander le départ de la course, la mise en pause de la course, l'arrêt de la course,... Il peut naturellement demander l'affichage des différentes positions.

On distingue donc :
- un *thread* par voiture (nommé $\mathtt{T_v}$),
- un *thread* pour le contrôle (nommé $\mathtt{T_o}$),
- le *thread* principal, dont le rôle est simplement de créer le $\mathtt{T_o}$ puis d'attendre sa mort.

Structures générales

On peut distinguer plusieurs structures informatiques, mais nous ne rentrerons dans les détails que de deux d'entre elles. L'une permet de conserver les informations relatives aux voitures (`voiture`) et l'autre permet d'obtenir une vision globale de la course (`course`).

L'opérateur

La course de voitures est organisée sur une piste en boucle qui sera en fait un simple tableau à deux dimensions [2] de largeur et de longueur fixées par la ligne de commande.

Chaque case de ce tableau contiendra :
- le nombre -1 : la case ne contient aucune voiture et la piste est donc considérée comme libre à cet endroit,
- le nombre -2 : la case contient une voiture accidentée, la piste est donc occupée à cet endroit,
- un numéro compris entre 0 et 255 : il s'agit du numéro de la voiture occupant cette position. On convient de limiter le nombre maximum de voitures à 256.

2. L'annexe donne aussi un exemple de tableau à une dimension auquel on accède par deux indices.

La piste est une donnée qui sera **partagée en lecture et en écriture** entre le T_o et les T_v :

- le T_o accède à la piste en lecture pour pouvoir réaliser un affichage sur la demande de l'opérateur.
- les T_v accèdent à la piste en lecture pour voir si les cases qu'ils convoitent sont libres mais surtout en écriture pour effacer leur ancienne position et y inscrire la nouvelle.

Le T_o accède de plus à une structure de course lui permettant de démarrer, mettre en pause, arrêter la course... Cette structure n'est accèdée qu'en lecture par les T_v qui peuvent ainsi prendre connaissance de ce qu'ils doivent faire. On peut donc voir la course comme une structure informatique qui pourrait contenir les champs suivants, parmi lesquels on retrouve la piste (ce qui suit est donné à titre d'exemple) :

```
struct course {
  int etat_course; /* pause, start, stop*/

  int largeur_piste;
  int longueur_piste;
  int *piste;

  int nb_voiture;
  pthread_t *thread_ident;
  ...
};
```

Le T_o pourra être constitué d'une phase d'initialisation (création de la piste et des voitures connues, ...), d'une boucle infinie qui règle la course et interagit avec l'utilisateur et d'une phase de terminaison. Le programme principal ne doit pas commander l'arrêt par un simple `Ctrl-C` ! On devra normalement se servir de la fonction `select()` présentée dans les annexes de manière à gérer les entrées utilisateur sur `stdin` sans pour autant bloquer le déroulement de la course.

Le T_o régule la circulation. Il ne doit pas permettre à une voiture d'avancer de plus d'une case par seconde, mais plusieurs voitures doivent pouvoir avancer toutes les secondes ! La temporisation doit être faite au sein du T_o par l'utilisation de la fonction `nanosleep()` dont on trouvera un exemple d'utilisation en annexe. Il est impératif que la consommation de CPU soit aussi minimaliste que possible. Les T_v ne doivent pas se servir des fonctions de sommeil telles que `nanosleep()`, `usleep()` ou `sleep()`.

Le T_o doit permettre de saisir *a minima* les commandes suivantes :

- `ajout` : permet d'ajouter une voiture dans la course. Si la course est déjà démarrée, la voiture entre dans sa boucle de circulation immédiatement. Sinon, comme les autres, elle attend la condition de démarrage.
- `depart` : permet de signaler le départ de la course, les T_v peuvent donc entrer dans leur boucle infinie de circulation.
- `pause` : permet de signaler la mise en pause de la course, les T_v voiture sont alors bloqués en attendant la reprise.
- `voir` : permet de voir l'occupation de la piste par les différentes voitures. L'affichage devra affecter une lettre de l'alphabet à chaque voiture (limitant par là

même le nombre de voiture aux alentours de 52 !) et représenter les cases vides de la piste par une espace.
- `fauxdepart` : les voitures doivent s'arrêter et se repositionner dans la configuration initiale. La course ne reprendra qu'à l'issue du signal de départ. Les voitures arrivées en cours de course seront éliminées.
- `stop` : permet d'arrêter la course en signifiant aux différents *threads* qu'ils doivent se terminer. Chaque *thread* affichera en se terminant la distance parcourue par la voiture qu'il représente. On veillera à ce que **les affichages n'interfèrent pas les uns avec les autres** [3].

Les voitures

Un T_v doit conserver son numéro (ce qui distingue les voitures les unes des autres et permettra un affichage agréable) sa position ainsi que son état (nous introduirons la notion d'accidents). Nous pourrons aussi ajouter certaines informations telles que, la distance parcourue, le nombre d'accès à la structure de course, *etc*.

Ces données sont intrinsèques à chaque T_v et n'ont donc pas vocation à être partagées. Elle pourront (devront !) être allouées au sein de chaque T_v. Nous pouvons donc décrire cette structure par quelque chose comme ce qui suit (encore une fois, cette structure est donnée à titre d'exemple) :

```
struct voiture {
  int numero;
  int etat;
  int x,y;
  int round;
  float distance;

  ...
};
```

D'autres éléments doivent probablement figurer dans cette structure, mais nous vous laissons le soin de les découvrir dans votre élaboration du projet.

Lorsqu'une voiture peut avancer (son *thread* est actif et en phase de boucle de course), elle doit avant tout veiller aux consignes de courses (par exemple la consigne faux départ la remet à sa position d'origine sur la grille de départ comme nous l'avons vu). Par contre un T_v ne doit pas utiliser d'attente bloquante comme par exemple `nanosleep()`. Seuls les mécanismes de blocage par `mutex` sont autorisés au sein des T_v.

Dans la boucle infinie de parcours de piste d'un T_v, et quand ce dernier est actif, un tirage aléatoire est réalisé afin de simuler un accident. Selon le résultat de ce tirage la voiture peut prendre un état accidenté. Dans ce cas, elle ne pourra pas avancer pendant un certain temps (ce temps est mesuré en secondes et défini par une macro définition dans un fichier d'en-têtes). Cette voiture bloque ainsi la progression des autres.

3. On rappelle qu'une écriture, lorsqu'elle peut être réalisée de manière concurrente, doit impérativement être protégée par un mécanisme garantissant un seul accès à la fois donc par l'utilisation de `mutex`

Si la voiture n'est pas accidentée, elle essaye d'avancer tout droit. Si cela n'est pas possible elle essaye de doubler par la gauche et si vraiment cela n'est pas possible elle essaye un dépassement illégal par la droite ! Si les trois cases situées à sa droite sont toutes occupées, la voiture ne peut pas avancer et elle reste donc à sa place.

Lorsqu'elle arrive en bout de piste, la voiture repart du début de piste (opérateur modulo).

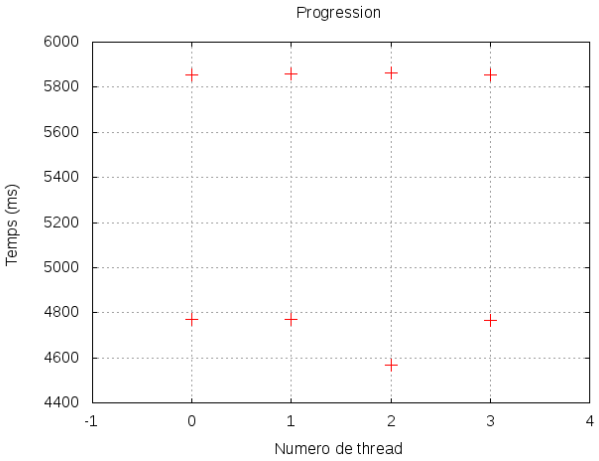
Chaque T_v doit inscrire dans un fichier commun à tous les T_v (il conviendra de bien gérer les accès concurrents à ce fichier) le numéro du *thread* ainsi que l'instant d'accès au fichier (donné en milliseconde à partir du démarrage du programme principal). Ce fichier de `log` sera ouvert en écriture par le T_o et son descripteur sera passé en paramètre aux T_v. Le T_o aura la charge de fermer le fichier une fois les T_v terminés.

On peut se servir de l'exemple donné dans l'annexe pour accéder aux structures de temps. On souhaite donc trouver dans ce fichier de `log` un ensemble de lignes constituées comme suit :

```
#temps en millisecondes et numero du thread
4567 2
4768 1
4769 3
4769 0
5856 3
5856 0
5861 1
5862 2
...
```

Nous remarquons que les interventions des différents T_v sont bien espacées d'au moins une seconde. Un programme comme **`gnuplot`** nous offre une possibilité d'affichage rapide à des fins de contrôle :

```
moi@ici:>gnuplot
set xlabel "Numero de thread"
set ylabel "Temps (ms)"
plot "thrv_log" title ""
set term png
set output "ma_courbe.png"
plot "thrv_log" title ""
exit
moi@ici:>
```



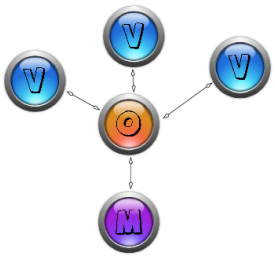
25.2 Cahier des charges

Contraintes structurelles

Il est fondamental de respecter les contraintes structurelles énoncées ci-après. Elles conditionnent en effet l'élaboration de votre programme et votre évaluation portera en grande part sur ce respect.

Le projet est réalisé en binôme ou en trinôme. Chaque élève d'un binôme doit appartenir au même groupe de petites classes.

Un T_v **ne peut pas** avancer de plus d'une case par seconde. Il est interdit d'utiliser une fonction de temporisation dans les T_v. Seul le T_o devra utiliser la temporisation. Une bonne organisation des blocages d'accès doit permettre de satisfaire cette contrainte. Il est aussi possible d'utiliser des tuyaux (fonction `pipe()`) et de faire en sorte que le T_o par le biais d'écritures à des instants stratégiques commande l'activité des T_v en attente de lecture sur ces tuyaux. Une préférence sera toutefois donnée à l'utilisation des fonctions de levée de condition telle que `pthread_cond_broadcast()`.



La fonction `main()` (**M**) crée le *thread* opérateur (**O**). Celui-ci crée à son tour les *threads* voiture (**V**) (il peut en créer au démarrage comme pendant la course) puis il attend la terminaison des différents T_v. Le *thread* principal attend quant à lui la terminaison du T_o.

Le programme principal devra créer un T_o. Le T_o devra créer un T_v par voiture. Le nombre de voitures et les dimensions de la piste seront donnés en ligne de commande. On peut convenir, par exemple, que pour lancer une course de 10 voitures sur une piste de longueur 80 et de largeur 5, on aboutisse à la ligne de commande suivante : `./course 10 80 5`. Le T_o pourra toutefois, sur intervention de l'utilisateur au travers de l'interface de commandes, créer un nouveau T_v pendant la course (avant ou après le démarrage).

La course ne doit pas démarrer tant que l'opérateur n'en donne pas l'ordre et ne doit consommer que très peu de CPU lors des différentes phases.

L'utilisateur interagit avec le T_o par une interface minimaliste proposant de donner des ordres. Cette interface doit de plus être capable d'afficher la piste et son occupation. Une voiture accidentée sera représentée par le symbole « $*$ », les voitures en fonctionnement par une lettre.

L'attente d'une commande dans l'interface ne doit en aucun cas être bloquante pour la régulation de course. L'utilisation de la fonction `select()` présentée dans les annexes est fortement conseillée.

Il est **strictement déconseillé** de terminer le programme sans attendre la fin des différents *threads* par l'utilisation de `pthread_join()`.

Programmation

Il est important de séparer dans plusieurs fichiers les différents éléments constituant le projet. On veillera à rédiger aussi des fichiers d'en-têtes. On veillera à apporter un grand soin dans la rédaction des programmes, notamment en utilisant des commentaires.

La compilation permettant d'obtenir le fichier exécutable de votre projet devra utiliser un `Makefile`. Vous trouverez dans l'aide mémoire du `C` du poly des exemples de `Makefile` dont vous pourrez vous inspirer.

Il est déconseillé de croire que tout appel système se déroule correctement ! On veillera donc à tester la valeur de retour de toutes fonctions systèmes. On pourra alors

utiliser la fonction `perror()` pour l'affichage de l'erreur [4].

Les différents fichiers sources devront compiler sans avertissement en utilisant les options suivantes du compilateur **`gcc`** :

```
-ansi -posix -pedantic -Wall
```

Il ne faudra pas faire usage de variables globales, source d'effets de bord souvent problématiques. Si vous ne pouvez pas faire autrement, il faudra le justifier de manière didactique dans votre rapport. En général, la recherche d'une justification suffit à comprendre qu'il existe un moyen de se passer de cette variable globale !

Remise du projet

Votre projet se composera de l'ensemble des fichiers sources (`*.h`,`*.c` et `Makefile`) ainsi que d'un rapport.

Le rapport doit contenir des explications concernant les différents choix que vous avez faits. Il doit contenir la description des différentes structures ainsi que l'organisation des fonctions que vous aurez programmées. Il doit aussi contenir un ensemble de graphiques permettant de démontrer, à partir de vos propres simulations, que vous respectez les contraintes de temps (un T_v n'obtient la main qu'une fois par seconde). Vous devrez naturellement donner la méthode permettant de compiler votre programme ainsi que quelques exemples de lancement et de sorties observées.

Le projet sera remis par courrier électronique à votre maître de petite classe, à l'adresse qu'il vous aura précisée. Ce courrier électronique sera composé de la manière suivante :

- Sujet : Projet IN201 nom1,nom2
- Pièce jointe : une archive tar contenant exclusivement les fichiers sources (`.c,.h`), le `Makefile` ainsi que le rapport au format `PDF`. Cette archive tar devra être réalisée de façon à intégrer un répertoire formé par les deux noms du binôme, soit :

```
$> mkdir jules_jim
$> cp *.c *.h Makefile rapport.pdf jules_jim/
$> tar cvf projet_201.tar jules_jim/
....
$>
```

La date de remise de votre projet n'est pas encore fixée. Toutefois, un système de pénalités de retard sera appliqué : 2 points pour 1 jour de retard, 4 points pour 2 jours de retard, 8 points pour 3 jours, 16 points pour 4 jours et plus.

Dans une carrière d'ingénieur ou de décideur, il est impératif, lorsque l'on est dépendant de contraintes extérieures, de prévoir des solutions de contournement. Le prétexte fallacieux d'une panne réseau / informatique / courrier électronique / RATP / EDF ou d'une crise financière ne sera pas pris en compte pour justifier un retard.

4. L'auteur de ce document n'a pas suivi cette recommandation dans les annexes présentées, il a tort, mais il n'a pas à faire le projet !

Améliorations éventuelles

Nous vous proposons certaines améliorations. Toutefois, **il est impératif d'avoir un projet fonctionnel avant d'envisager d'éventuelles améliorations**. En effet si vous ne parvenez pas à faire aboutir un projet « amélioré », vous n'aurez rien à présenter, ce qui a des conséquences sur votre note.

Vous pouvez proposer dans votre rapport comment se déroulerait une programmation dans laquelle chaque T_v serait remplacé par un programme. Le T_o réaliserait des appels à `fork()` pour créer autant de fils que de voitures. Quelle méthode de communication pourriez-vous utiliser ?

Vous pouvez générer des pistes avec obstacles et prévoir dans ce cas que les voitures puissent aller vers le bas ou le haut pour contourner les obstacles.

L'utilisateur peut gérer une des voitures et choisir comment cette dernière avance en fonction de sa vision de la course. Si l'attente d'une commande utilisateur est par exemple de deux secondes au début de la course, on peut aussi penser que ce temps diminue peu à peu, laissant à l'utilisateur très peu de temps pour réfléchir.

25.3 Annexes

La meilleure source de renseignements pour l'utilisation correcte d'une fonction de la bibliothèque `C` est de loin la commande **`man`** :

```
moi@ici:> man pthread_cond_wait
...
NAME
  pthread_cond_wait -- wait on a condition variable

SYNOPSIS
  #include <pthread.h>

  int
  pthread_cond_wait(pthread_cond_t *restrict cond,
                    pthread_mutex_t *restrict mutex);
...
moi@ici:>
```

Les appels système

Il serait très judicieux que vos programmes utilisent au moins une fois les fonctions suivantes :

- `pthread_create()` : création d'un *thread*,
- `pthread_join()` : attente du retour d'un *thread*,
- `pthread_mutex_lock()` : pose d'un verrou,
- `pthread_mutex_unlock()` : dépose d'un verrou,
- `pthread_cond_wait()` : attente bloquante d'une condition,
- `pthread_cond_signal()` : signal de levée de condition (1 *thread*),
- `pthread_cond_broadcast()` : signal de levée de condition, (plusieurs *threads*),

– `select()` : surveillance de descripteurs de fichier.

Piste 1D *vs* 2D

La piste, donnée pour être un tableau bidimensionnel est présentée dans une des structures comme un tableau monodimensionnel. Cela présente un avantage, celui de réduire le mécanisme d'allocation de la mémoire. Ensuite, une simple macro définition permet de circuler dans ce tableau 1D comme dans un tableau 2D :

```
struct piste {
  int w,h;
  int *p;
}
#define CASE(P,X,Y) (*(((P)->p) + (Y)*((P)->w) + (X)))
  ...
  piste->p = malloc(largeur*longueur*sizeof(int));
  CASE(piste,3,4) = 2;
  ...
```

Passage de paramètres à un *thread*

Il est tout à fait possible de délivrer un certain nombre de paramètre à un *thread* par l'utilisation d'une structure :

```
struct parametre {
  int numero; /* numero affecte*/
  pthread_mutex_t mtx; /* mutex protegeant la condition */
  pthread_cond_t cnd;  /* condition */
  int dummy; /* un entier pouvant servir */
}

/*
 Fonction multi-threadee
 Cette fonction fait...

 Parametre: *param sert a passer le numero du thread
 Retour: la fonction retourne un pointeur NULL (pas
 de pthread_join avec attribut utilisable).
*/
void *thread_func(void *p)
{
  struct parametre *lp;
  lp = (struct parametre *)p;

  if (lp->numero == 1) {
    ...
  }
  ...
  return NULL;
}

int main(int argc, char **argv)
{
  ...
  struct parametre prm;
  prm.numero = 1;
  prm.mtx = PTHREAD_MUTEX_INITIALIZER;
```

```
  prm.cnd = PTHREAD_COND_INITIALIZER;
  prm.dummy = 26;
  ...
  pthread_create(&mon_thread, NULL,
                 thread_func, (void*)(&prm));
  ...
}
```

Temporisation par sommeil

La fonction `nanosleep()` s'utilise assez simplement. Cependant, en raison de l'utilisation de différentes versions d'OS, il convient de faire précéder les éventuelles inclusions d'en-têtes (tels que `stdio.h`, `stdlib.h`, ...) par les lignes qui suivent :

```
#define _BSD_SOURCE
#define _POSIX_C_SOURCE 200112L
#include <features.h>
...

#include <time.h>
...
  /* Attente de 1,5 secondes */
  treq.tv_sec =          1;
  treq.tv_nsec = 500000000;
  if (nanosleep(&treq,NULL) != 0) {
    perror("Impossible de dormir!");
    ...
  }
```

Mesure du temps écoulé

L'exemple qui suit montre comment réclamer une attente de $1,51$ secondes et mesurer par l'intermédiaire de la fonction `gettimeofday()` le nombre de millisecondes écoulées entre l'appel à `nanosleep()` et la sortie de veille.

```
int main(int argc, char **argv)
{
  struct timeval tp_start;
  struct timeval tp_current;
  struct timespec request_sleep;

  request_sleep.tv_sec = 1; /* secondes */
  request_sleep.tv_nsec = 510000000; /* nano-secondes */

  gettimeofday(&tp_start,NULL);

  nanosleep(&request_sleep,NULL);

  gettimeofday(&tp_current,NULL);

  fprintf(stderr,"Temps ecoule: %d\n",
          (int)(((tp_current.tv_sec) - (tp_start.tv_sec)) * 1000 +
                ((tp_current.tv_usec) - (tp_start.tv_usec)) / 1000 ));
  exit(EXIT_SUCCESS);
}
```

Surveillance de descripteurs

La fonction `select()` permet de surveiller des descripteurs de fichiers sans pour autant être bloquante. Si un descripteur est prêt (en lecture ou en écriture), la fonction retourne immédiatement et l'on peut alors vérifier lequel. Son utilisation est assez simple comme le montre l'exemple ci après.

Listing 25.1 – *Utilisation de* `select()`

```
#define _BSD_SOURCE
#define _POSIX_C_SOURCE 200112L
#include <features.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>

int main(int argc, char **argv)
{
  fd_set surveille_input;
  struct timeval attente;
  char buffer[256];
  int ret;

  fprintf(stdout,">");
  while (1) {
    /*FD_ZERO vide l'ensemble des descripteurs surveilles*/
    FD_ZERO(&surveille_input);
    /*FD_SET ajout le descripteur de numero 0, soit stdin*/
    FD_SET(0, &surveille_input);
    attente.tv_sec = 0;
    attente.tv_usec = 500000;
    buffer[0] = '\0';
    ret = select(1, &surveille_input,NULL,NULL,&attente);
    if (ret < 0) {
      perror("Probleme avec select:");
      exit(EXIT_FAILURE);
    }
    if (ret == 0) {
      fprintf(stdout,"Soyez rapide!\n>");
    } else if (ret == 1) {
      /*Ici FD_ISSET est inutile car on ne surveille qu'un*/
      /* descripteur, donc si select() sort en etant > 0  */
      /*c'est que stdin contient quelque chose*/
      if (FD_ISSET(0,&surveille_input)) {
        fgets(buffer,256,stdin);
        buffer[strlen(buffer) - 1] = '\0';
        fprintf(stdout,"Vous etes bon (\"%s\")\n>",buffer);
        if (strncmp(buffer,"exit",strlen("exit")) == 0) {
          exit(EXIT_SUCCESS);
        }
      }
    }
  }
  exit(EXIT_FAILURE);
```

Temporisation par attente de condition

La fonction `pthread_cond_wait()` prend comme paramètres un pointeur sur une condition (de type `pthread_cond_t *`) ainsi qu'un *mutex* (de type `pthread_mutex *`). Cette fonction commence par déverrouiller le *mutex* passé en paramètre puis rentre en attente de la condition. Le *mutex* sera verrouillé de nouveau lorsque la fonction réceptionne le signal de condition, juste avant de sortir. **Il est très important que la fonction qui appelle `pthread_cond_wait` verrouille le *mutex* avant l'appel**.

Le signal de condition est envoyé par la fonction `pthread_cond_signal()` qui prend comme paramètre le pointeur sur la condition. Cette fonction ne réveille qu'un seul *thread* parmi ceux attendant la condition. Le signal peut aussi être envoyé à tous les *thread* par une une autre fonction (cf. ci-après).

Un exemple d'utilisation pourrait être le suivant :

Listing 25.2 – *Signal de réveil*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
/*
 Fonction de calcul multi-threadee
 Cette fonction attend 2 signaux de conditions puis
 se termine.

 Parametre: *param ne sert pas, on peut passer NULL
 Retour: la fonction retourne un pointeur NULL (pas
 de pthread_join avec attribut utilisable).
*/
void *thread_func(void *param)
{
  int k=2;
  fprintf(stderr,"Demarrage du thread\n");
  while(k>0) {
    pthread_mutex_lock(&mutex);
    fprintf(stderr,"Attente\n");
    pthread_cond_wait(&condition,&mutex);
    fprintf(stderr,"Travail\n");
    pthread_mutex_unlock(&mutex);
    k--;
  }
  fprintf(stderr,"Fin du thread\n");
  return NULL;
}

int main(int argc, char **argv)
{
   pthread_t mon_thread;
   int k;
   fprintf(stderr,"######Creation du thread\n");
   pthread_create(&mon_thread, NULL, thread_func, (void*)NULL);
   for(k=0;k<2;k++) {
     fprintf(stderr,"######Attente de 5s\n");
     sleep(5);
     fprintf(stderr,"######Reveil du thread\n");
```

```
    pthread_cond_signal(&condition);
  }
  pthread_join(mon_thread,NULL);
  fprintf(stderr,"######Fin\n");
  exit(EXIT_SUCCESS);
}
```

dont la sortie est :

```
moi@ici:> ./exemple
######Creation du thread
Demarrage du thread
Attente
######Attente de 5s
######Reveil du thread
######Attente de 5s
Travail
Attente
######Reveil du thread
Travail
Fin du thread
######Fin
moi@ici:>
```

Voici un autre exemple avec cette fois l'utilisation de la fonction

`pthread_cond_broadcast()`

ainsi qu'une mesure du temps d'attente.

Listing 25.3 – *Signal à diffusion large*

```
#define _BSD_SOURCE
#define _POSIX_C_SOURCE 200112L
#include <features.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;

pthread_mutex_t mutex_stderr = PTHREAD_MUTEX_INITIALIZER;

/*
 Fonction de calcul multi-threadee
 Cette fonction attend 2 signaux de conditions puis
 se termine. Elle affiche ses temps d'attente.

 Parametre: *param sert a passer le numero du thread
 comme parametre afin de differencier les affichages

 Retour: la fonction retourne un pointeur NULL (pas
 de pthread_join avec attribut utilisable).
*/
void *thread_func(void *param)
{

  struct timeval tp_start;
```

```
  struct timeval tp_current;
  int *lparam = (int *)param;
  int k=2;

  int i,j;

  fprintf(stderr,"Demarrage du thread %d\n",*lparam);
  while(k>0) {
    pthread_mutex_lock(&mutex);
    gettimeofday(&tp_start,NULL);
    fprintf(stderr,"%d bloque\n",*lparam);
    pthread_cond_wait(&condition,&mutex);
    pthread_mutex_unlock(&mutex);
    gettimeofday(&tp_current,NULL);
    pthread_mutex_lock(&mutex_stderr);
    fprintf(stderr,"%d travaille (attente de %d)\n",*lparam,
            (int)(((tp_current.tv_sec) -
                   (tp_start.tv_sec)) * 1000 +
                  ((tp_current.tv_usec) -
                   (tp_start.tv_usec)) / 1000 ));
    pthread_mutex_unlock(&mutex_stderr);
    k--;
    for(i=0;i<12000000;i++) j=random();
  }
  pthread_mutex_lock(&mutex_stderr);
  fprintf(stderr,"Fin du thread %d\n",*lparam);
  pthread_mutex_unlock(&mutex_stderr);
  return NULL;
}

int main(int argc, char **argv)
{
   pthread_t mon_thread[2];
   int k;
   int thident[2];
   thident[0] = 0;   thident[1] = 1;
   fprintf(stderr,"######Creation des threads\n");
   pthread_create(mon_thread+0, NULL,
                  thread_func, (void*)(thident+0));
   pthread_create(mon_thread+1, NULL,
                  thread_func, (void*)(thident+1));
   for(k=0;k<2;k++) {
     sleep(5);
     fprintf(stderr,"######Reveil du thread\n");
     pthread_cond_broadcast(&condition);
   }
   pthread_join(mon_thread[0],NULL);
   pthread_join(mon_thread[1],NULL);
   fprintf(stderr,"######Fin\n");
   exit(EXIT_SUCCESS);
}
```

La sortie de ce programme donne quelque chose comme suit :

```
moi@ici:> ./wait_cond
######Creation des threads
Demarrage du thread 0
Demarrage du thread 1
0 bloque
1 bloque
######Reveil du thread
0 travaille (attente de 5000)
1 travaille (attente de 5002)
```

```
0 bloque
1 bloque
######Reveil du thread
0 travaille (attente de 2278)
1 travaille (attente de 2285)
Fin du thread 0
Fin du thread 1
######Fin
moi@ici:>
```

Quatrième partie

L'aide-mémoire du langage C

26

Aide-mémoire de langage C

Introduction

Ce document n'est pas un manuel de programmation, ni un support de cours concernant le langage `C`. Il s'agit simplement d'un aide-mémoire qui devrait permettre à tout élève ayant suivi le cours de `C` d'écrire facilement des programmes. Il a été écrit à partir de l'excellent livre de référence de B. W. Kernighan et D. M. Ritchie intitulé *Le langage C, C ANSI*.

Le langage `C` dont il est question ici est celui défini par la norme ANSI ISO/IEC 9899:1990(E). Néanmoins, quelques ajouts seront faits, comme par exemple le type `long long`, car ils sont supportés par la plupart des compilateurs. De plus la majorité de ces ajouts ont été intégrés dans la nouvelle norme ISO/IEC 9899:1999.

Comme il est très difficile de présenter indépendamment les différents éléments d'un langage, il y aura parfois des références à des sections ultérieures dans le cours du texte. Cet aide-mémoire est un document de travail et il ne faut pas hésiter à aller et venir de page en page.

Pour une utilisation régulière et efficace du langage, il ne saurait être trop recommandé de compléter ce petit aide-mémoire par une lecture du livre de référence cité ci-dessus, en particulier la partie *Annexe A : manuel de référence* de la norme ANSI et *Annexe B : La bibliothèque standard*, dont un résumé succinct est présenté à la fin de ce document.

Le `C` a été développé au début par Dennis Ritchie, puis Brian Kernighan, chez AT&T Bell Labs à partir de 1972, pour une machine de l'époque, le PDP-11. Le but était d'en faire un langage adapté pour écrire un système d'exploitation portable. Le langage fut appelé `C` car il héritait de fonctionnalités d'un précédent langage nommé `B` et était influencé par un autre langage nommé `BCPL`. Le `C` a servi immédiatement (en 1972-1974 par Ken Thompson et Dennis Ritchie) à réimplanter le système Unix (dont le début du développement date de 1970). Les lois anti-trust empêcheront AT&T de commercialiser Unix, et c'est ainsi que le `C` et Unix se répandront via les universités dans les années 1980. Le `C` est un langage très utilisé du fait de sa relative facilité d'utilisation, de son efficacité et de son adéquation au développement des systèmes

d'exploitations (grâce en particulier à l'arithmétique sur les pointeurs et au typage faible). On dit parfois avec humour que c'est *un langage qui combine l'élégance et la puissance de l'assembleur avec la lisibilité et la facilité de maintenance de l'assembleur* !

Certaines des faiblesses du langage, en particulier des différences de plus en plus marquées entre ses nombreuses implantations et la nécessité d'une boîte à outils portable seront corrigées par la norme ANSI de 1985. Afin d'adapter le langage aux exigences de développement de grande envergure, son successeur, restant autant que possible compatible au niveau source, sera le `C++`, développé par Bjarne Stroustrup, avec une philosophie assez différente, mais en conservant la syntaxe.

26.1 Éléments de syntaxe

Structure générale d'un programme C

Pour respecter la tradition, voici le programme le plus simple écrit en C :

Listing 26.1 – *Le premier programme !*

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
      printf("Je vous salue, O maitre.");
      exit(EXIT_SUCCESS);
}
```

Un programme C est constitué de :

- **Directives du préprocesseur** qui commencent par un # et dont nous reparlerons par la suite.
  Ici `#include <stdio.h>` permet de disposer des fonctions d'entrée-sortie standard *STanDard Input/Output Header*, en particulier de la fonction `printf()` qui permet l'affichage à l'écran.
- **Déclarations globales** de types et de variables, ainsi que des `prototypes` de fonction, c'est-à-dire des déclarations de fonctions où ne figurent que le nom de la fonction, les paramètres et le type de valeur de retour, mais sans le corps de la fonction. Il n'y en a pas dans l'exemple ci-dessus.
- **Fonctions**. Tous les sous-programmes en C sont des fonctions. Ce que l'on appelle habituellement procédure (en Pascal par exemple) est ici une fonction qui ne retourne rien, ce qui se dit `void` en C. Le compilateur reconnaît une fonction à la présence des parenthèses qui suivent le nom de la fonction et à l'intérieur desquelles se trouve la liste des paramètres. Cette liste peut être vide comme c'est le cas ci-dessus pour `main()`.
- **Une fonction particulière** dont le nom est obligatoirement **main** qui est la fonction appelée par le système d'exploitation lors du lancement du programme. C'est la seule fonction dans cet exemple.

Un exemple complet

Voici un exemple plus complet montrant la structure générale, avec une variable globale `iglob` visible de partout, les prototypes des fonctions (indispensable [1] pour la fonction `somme()` dans le cas présent) et les fonctions. Ce programme calcule 5! d'une façon un peu détournée...

1. En fait, en l'absence de prototype, et si le texte de la fonction n'a pas été rencontré jusqu'alors, le compilateur considère qu'elle retourne un `int`, ce qui fonctionne ici... mais n'est pas recommandé.

Listing 26.2 – *Calcul de factorielle*

```
#include <stdio.h>

/* variables globales */
int iglob;

/* prototypes des fonctions */
int somme(int a, int b);
int produit(int a, int b);

/* fonctions */
int produit(int a, int b) {
     int sortie;

     if (a>1) {
       sortie = somme(produit(a-1, b) , b);
     } else {
       sortie = b;
     }
     return sortie;
}

int somme(int a, int b) {
     return a+b;
}

/* programme principal */
main() {
    int j;
    j=1;
    for (iglob=1;iglob<=5;iglob++) {
        /* appel de la fonction produit() */
        j = produit(iglob, j);
     }
     printf("%d\n",j);
}
```

On remarquera dès à présent qu'une fonction est formée d'*un type*, du *nom de la fonction* suivi de la *liste des paramètres entre parenthèses* puis d'un `bloc` formant le corps de la fonction. On appelle bloc en `C` tout ensemble délimité par des accolades {} et comportant des déclarations de variables (il peut ne pas y en avoir) suivies d'expressions. Il est toléré de ne pas faire figurer de type de retour pour une fonction (c'est le cas du `main()` ici), mais c'est une pratique fortement déconseillée.

Un *prototype* a la même écriture que la fonction, à cela près que le bloc du corps de la fonction est remplacé par un point-virgule. Lorsqu'il sera fait référence à des fonctions provenant d'une bibliothèque dans la suite de ce document, celles-ci seront présentées par leur prototype.

Ces prototypes permettent au compilateur de connaître les paramètres et la valeur de retour de la fonction avant même de connaître son implantation. Comme on le verra par la suite, il est généralement recommandé de regrouper les prototypes de toutes les fonctions dans un ou des fichiers séparés du code source. Ainsi, on sépare (au sens Ada) l'interface de son implémentation, ce qui évite de replonger dans le code source pour trouver les arguments d'une fonction et permet une vérification de cohérence par le compilateur.

26.2 La compilation sous Unix

Exemple élémentaire

Le compilateur `C` sous Unix fourni avec la machine s'appelle généralement `cc`. Néanmoins on utilise souvent `gcc` car il est présent partout, il supporte la norme ANSI ainsi que quelques extensions bien pratiques, il est libre et très efficace sur toutes les machines. C'est par ailleurs un des compilateurs `C` digne de ce nom disponibles sur les systèmes d'exploitation libres (Linux, FreeBSD, NetBSD). Il est important de mentionner deux autres compilateurs disponibles, `Clang` développé à partir de 2007 par Apple pour obtenir un compilateur supportant spécifiquement le `C`, le `C++` et l'`Objective C` (voir `http://clang.llvm.org/` et `PCC` (*Portable C Compiler*) disponible pour les `BSD` (voir `http://pcc.ludd.ltu.se/`). Si ces deux derniers compilateurs affichent des performances nettement meilleures que `gcc`, ils ne supportent pas autant de langages et ce n'est d'ailleurs pas leur but.

La compilation se fait dans le cas le plus simple par :

```
Idefix> gcc <source.c> -o <executable>
```

où il faut bien entendu remplacer les chaînes entre `<>` par les bons noms de fichiers.

Si des bibliothèques sont nécessaires, par exemple la bibliothèque mathématique, on ajoute à la ligne de commande précédente `-l<bibliotheque>` (`-lm` pour la bibliothèque mathématique).

Options du compilateur *gcc*

`gcc` accepte de très nombreuses options (voir la page de manuel `man gcc`). En voici quelques unes fort utiles :

-g Compilation avec les informations utiles pour le débogage (debug). Avec `gcc` on peut ainsi utiliser l'excellent débogueur `gdb` (ou la version avec interface graphique `xxgdb`).

-ansi Force le compilateur à être conforme à la norme ANSI. Les extensions comme les `long long` sont néanmoins acceptées.

-pedantic Le compilateur refuse tout ce qui n'est pas strictement conforme à la norme ANSI. Ceci, combiné avec `-ansi` permet de vérifier qu'un programme est strictement conforme à la norme et donc en théorie portable.

-Wall Force l'affichage de tous les messages d'alerte lors de la compilation.

-O Optimisation de la compilation. Il vaut mieux ne l'utiliser que lorsque le programme fonctionne déjà de manière correcte. Des optimisations encore plus poussées sont possibles avec `-On`, où *n* est un entier entre 1 et 3.
L'optimisation conduisant parfois à des déplacements d'instructions, il est déconseillé d'exécuter au débogueur des programmes optimisés.

-O0 Permet d'empêcher toute optimisation.

-Os Optimisation pour la taille. Le compilateur essaye de produire l'exécutable le plus petit possible, parfois aux dépens de la vitesse d'exécution.

-I<rep> Précise le(s) répertoire(s) où il faut chercher les fichiers d'en-têtes (`#include`).

-D<var=val> Permet de définir des variables du préprocesseur. `-DDEBUG=2` est équivalent à mettre au début du source la ligne `#define DEBUG 2`. On peut également écrire `-DMONOPTION` qui est équivalent à `#define MONOPTION` [2].

-S Le compilateur génère uniquement le listing assembleur (et non du code objet ou un exécutable), avec par défaut le suffixe `.s`. C'est utile pour optimiser efficacement de très petits morceaux de code critiques (ou par simple curiosité !).

-M Permet d'extraire des règles de dépendance (envoyées sur la sortie standard) pour le programme `make`, de façon similaire à `makedepend` (cf. page 498).

-p L'exécutable généré intègre des fonctions de mesure de temps passé, nombre d'appels (*profiling*), ... Lors du lancement de cet exécutable, un fichier `gmon.out` est généré, et les résultats sont consultables ultérieurement avec :

`gprof <nom_de_l'exécutable>`.

-static Demande au compilateur de produire un binaire qui s'autosuffit, c'est-à-dire qui ne nécessite aucune bibliothèque dynamique. Cette option n'agit en fait que pour l'édition de lien et nécessite de disposer de toutes les bibliothèques sous leur forme archive (extension .a).

De nombreuses autres options d'optimisation sont disponibles et peuvent influencer sensiblement la rapidité d'exécution dans le cas de programmes très calculatoires.

Compilation séparée

La programmation modulaire consiste à ne pas regrouper dans un seul fichier tout le code source d'un projet, mais au contraire à le répartir dans plusieurs fichiers et à compiler ces derniers séparément. Ceci permet une réutilisation simple du code car un ensemble de fonctions d'usage général peut être isolé dans un fichier et réutilisé

2. Il s'agit de directives du préprocesseur qui seront présentées en partie 26.6.

facilement dans d'autres programmes. De plus, pour les gros développements logiciels, la compilation est une étape longue à chaque modification ; il est donc souhaitable de séparer les sources en plusieurs petits fichiers, ainsi seuls les fichiers effectivement modifiés devront être recompilés à chaque fois.

Afin de rendre le source d'un projet le plus lisible possible, on adopte généralement quelques conventions simples. On distingue ainsi trois types de fichiers sources :

- Des fichiers contenant des fonctions qu'on nommera avec une extension `.c`, mais ne contenant en aucun cas une fonction `main()`.
- Des fichiers contenant les déclarations de type, les prototypes des fonctions (voir plus loin) et éventuellement des macros, correspondant aux sources précédentes. On les nommera comme les sources auxquelles ils correspondent mais avec un suffixe `.h`, et ils seront inclus par des directives `#include` dans les sources des fichiers `.c`. On les appelle des fichiers d'en-têtes (*headers*).
- Des fichiers avec une extension `.c`, contenant un `main()` et utilisant les fonctions des autres fichiers (qui sont déclarées par des inclusions des `.h`). Chaque fichier de ce type donnera un exécutable du projet final. Il est recommandé d'adopter une convention claire de nommage de ces fichiers afin de les distinguer des premiers (par exemple les nommer `m_<nom_exécutable>.c` ou `main.c` s'il n'y en a qu'un).

La compilation d'un fichier source `.c` en un fichier objet ayant le même nom de base mais le suffixe `.o` se fait par une commande du type :

```
menthe22> gcc -c <source.c>
```

Rappelons qu'un fichier objet est en fait un fichier contenant le code compilé du source correspondant, mais où figurent également (entre autre) une table des variables et des fonctions exportées définies dans le source, ainsi qu'une table des variables et des fonctions qui sont utilisées mais non définies.

Les fichiers objets (`.o`) sont alors liés ensemble, c'est la phase *d'édition de liens*, où sont également ajoutées les bibliothèques (ensemble de fonctions précompilées). Dans cette phase, les tables de tous les fichiers objets sont utilisées pour remplacer les références aux variables et aux fonctions inconnues par les vrais appels. Le résultat est alors un fichier exécutable. Ceci se fait par une commmande du type :

```
menthe22> gcc <obj1.o> <obj2.o> ... <objn.o> -o <executable> -lm
```

Il convient de bien faire la différence entre la ligne `#include <math.h>` et l'option `-lm` sur la ligne de commande. En effet, l'en-tête `math.h` contient tous les prototypes des fonctions mathématiques, mais ne contient pas leur code exécutable. Le code permettant de calculer le logarithme par exemple se trouve en fait dans une bibliothèque précompilée. C'est l'option d'édition de lien `-lm` qui fournit la fonction elle-même en ajoutant au programme le code de calcul des fonctions de la bibliothèque mathématique.

Les bibliothèques précompilées sont stockées sous Unix comme des fichiers *archive* portant l'extension `.a`. Lors de l'utilisation de `-l<nom>`, le fichier recherché s'appelle en fait `lib<nom>.a` et se trouve généralement dans `/usr/lib`. Il est donc possible mais fortement déconseillé de remplacer la ligne

```
menthe22> gcc <obj1.o> ... <objn.o> -o <executable> -lm
```

par

```
menthe22> gcc <obj1.o> ... <objn.o> /usr/lib/libm.a
```

En fait, sur la majorité des machines, `gcc` crée des exécutables qui font appel à des bibliothèques dynamiques (« shared object » extension .so). La commande `ldd <exécutable>` permet d'obtenir les bibliothèques dynamiques dont un exécutable aura besoin pour fonctionner. Dans ce mécanisme, l'édition de lien finale se fait au moment du lancement de l'exécutable, avec les bibliothèques dynamiques qui peuvent éventuellement être déjà présentes en mémoire.
Pour compiler en incluant la bibliothèque libm.a dans ce cas il faut ajouter l'option -static :

```
menthe22> gcc -static <obj1.o> <obj2.o> ... <objn.o> -o <executable> -lm
```

Makefile

Garder trace des dépendances [3] des fichiers entre eux et réaliser manuellement les phases de compilation serait fastidieux. Un utilitaire générique pour résoudre ce genre de problèmes de dépendances existe : `make`. Son utilisation dépasse largement la compilation de programmes en `C`. La version utilisée ici est GNU make, accessible sous le nom `make` sur certains Unix (notamment sous Linux) et sous le nom `gmake` sur d'autres.

`make` fonctionne en suivant des règles qui définissent les dépendances et les actions à effectuer. Il cherche par défaut ces règles dans un fichier nommé `Makefile` dans le répertoire courant. Voici un exemple simple qui permet de compiler les fichiers `fich.c` et `m_tst.c` en un exécutable `tst` :

Listing 26.3 – *Un `Makefile` simple*

```
## Makefile
# L'executable depend des deux fichiers objets
tst : fich.o m_tst.o
rightarrowfillgcc fich.o m_tst.o -o tst #ligne de compilation a executer

# Chaque fichier objet depend du source correspondant
```

3. On dit qu'un fichier dépend d'un autre, si le premier doit être reconstruit lorsque le second change. Ainsi un fichier objet dépend de son source et de tous les en-têtes inclus dans le source.

```
      # mais egalement de l'en-tete contenant les prototypes
      # des fonctions
      fich.o : fich.c fich.h
rightarrowfillgcc -c fich.c

      m_tst.o : m_tst.c fich.h
rightarrowfillgcc -c m_tst.c
```

La première ligne d'une règle indique le nom du fichier concerné (appelé fichier cible) suivi de deux-points (`:`) et de la liste des fichiers dont il dépend. Si l'un de ces fichiers est plus récent [4] que le fichier cible, alors les commandes situées sur les lignes suivant la règle sont exécutées. Ainsi, si `tst` doit être fabriqué, la première règle nous dit que `fich.o` et `m_tst.o` doivent d'abord être à jour. Pour cela, `make` tente alors de fabriquer ces deux fichiers. Une fois cette tache effectuée (ce sont les deux autres règles qui s'en chargent), si le fichier nommé `tst` est plus récent que `fich.o` et `m_tst.o`, c'est terminé, dans le cas contraire, `tst` est refabriqué par la commande qui suit la règle de dépendance : `gcc fich.o m_tst.o -o tst`.

> **Attention** le premier caractère d'une ligne de commande dans un `Makefile` est une tabulation (caractère TAB) et non des espaces.

Avec le `Makefile` ci-dessus, il suffit ainsi de taper `make tst` (ou même `make`, car `tst` est la première règle rencontrée) pour fabriquer l'exécutable `tst`. `make` n'exécute que les commandes de compilation nécessaires grâce au procédé de résolution des dépendances explicité ci-dessus.

`make` peut servir à la génération automatique de fichiers d'impression à partir de sources LaTeX... Il peut permettre d'automatiser à peu près n'importe quelle séquence d'actions systématiques sur des fichiers. L'envoi par courrier électronique du projet en `C` est un exemple (à aménager si vous souhaitez l'utiliser) :

Listing 26.4 – *Un `Makefile` pour le projet !*

```
      # Envoi automatique de mail pour le projet
      # A remplacer par le bon nom
      NOM=monnom
      PROF=bcollin

      # Les sources du projet
      SRCS= Makefile interface.c calcul.c \
rightarrowfillfichiers.c m_projet.c

      mail.txt: explication.txt $(NOM).tar.gz.uu
```

4. Les dépendances sont vérifiées à partir de la date de dernière modification de chaque fichier.

```
rightarrowfillcat explication.txt $(NOM).tar.gz.uu > mail.txt

      mail: mail.txt
rightarrowfillmail -s IN201 gueydan < mail.txt

      $(NOM).tar.gz.uu: $(SRCS)
rightarrowfilltar cvf $(NOM).tar $(SRCS)
rightarrowfillgzip -c $(NOM).tar > $(NOM).tar.gz
rightarrowfilluuencode $(NOM).tar.gz $(NOM).tar.gz > $(NOM).tar.gz.uu
```

Voici enfin un exemple générique complet pour compiler les sources `interface.c`, `calcul.c`, `fichiers.c` et `m_projet.c`. Les premières lignes de l'exemple sont des commentaires (commençant par #), et expliquent le rôle du projet.

Il est fait usage ici de l'utilitaire `makedepend` pour fabriquer automatiquement les dépendances dues aux directives `#include`. En effet, chaque fichier objet dépend d'un fichier source mais aussi de nombreux fichiers d'en-têtes dont il est fastidieux de garder la liste à jour manuellement. `makedepend` parcourt le source à la recherche de ces dépendances et ajoute automatiquement les règles correspondantes à la fin du fichier `Makefile` (les règles ainsi ajoutées à la fin ne sont pas reproduites ci-dessous). `makedepend` est un outil faisant partie de X11, `gcc -M` peut jouer le même rôle mais son utilisation est un peu plus complexe.

L'usage de variables (`CFLAGS`, `LDFLAGS`...) permet de regrouper les différentes options au début du fichier `Makefile`. La variable contenant la liste des fichiers objets est obtenue automatiquement par remplacement des extensions `.c` par `.o`.

Des caractères spéciaux dans les règles permettent d'écrire des règles génériques, comme ici la génération d'un fichier objet ayant l'extension `.o` à partir du source correspondant. Un certain nombre de ces règles sont connues par défaut par `make` si elles ne figurent pas dans le fichier `Makefile`.

Listing 26.5 – *Un `Makefile` très complet !*

```
############################################################
# ECOLE    : ENSTA
# PROJET   : Cours IN201
# FONCTION : Compilation du projet
# CIBLE    : UNIX / make ou GNU make
# AUTEUR   : MERCIER Damien
# CREATION : Thu Sep 23 10:02:31 MET DST 1997 sur Obiwan
# MAJ      : Tue Jan 12 20:18:28 MET 1999 sur Idefix
# Version  : 1.02
############################################################
# Le compilateur utilise
CC=gcc
# Options de compilation
```

```
      CFLAGS=-I. -ansi -g -Wall
      # Options d'edition de liens
      LDFLAGS=-lm

      # Le nom de l'executable
      PROG = projet
      # La liste des noms des sources (le caractere \ permet de
      # passer a la ligne sans terminer la liste)
      SRCS = interface.c calcul.c \
rightarrowfillfichiers.c m_projet.c

      # La liste des objets
      OBJS = $(SRCS:.c=.o)

      ############################################################
      # Regles de compilation
      ############################################################
      # Regle par defaut : d'abord les dependances, puis le projet
      all: dep $(PROG)

      # Ajout automatique des dependances entre les .c et les .h
      #  a la fin de ce Makefile
      dep:
rightarrowfillmakedepend -- $(CFLAGS) -- $(SRCS)

      # Comment compile t'on un .c en un .o ? (Regle generique)
      %.o : %.c
rightarrowfill$(CC) $(CFLAGS) -c $*.c

      # L'edition de liens
      $(PROG) : $(OBJS)
rightarrowfill$(CC) $(CFLAGS) $(OBJS)  -o $(PROG) $(LDFLAGS)

      # 'make clean' fait un nettoyage par le vide
      clean:
rightarrowfill\rm -f $(PROG) $(OBJS) Makefile.bak *~ #* core
```

26.3 Types, opérateurs, expressions

Les types simples

Les types de base en `C` sont les nombres entiers (`int`), les nombres réels (`float`) et les caractères (`char`). Certains de ces types existent en plusieurs tailles :

char représente un caractère codé généralement sur un octet (8 bits) sur les machines actuelles ;

int désigne un entier codé sur la taille d'un mot machine, généralement 32 bits sur les machines actuelles ;

short (on peut aussi écrire `short int`) désigne un entier sur 16 bits ;

long (ou `long int`) désigne un entier sur 32 bits ;

float est un nombre flottant (réel) simple précision, généralement codé sur 32 bits ;

double est un flottant double précision codé le plus souvent sur 64 bit ;

long double est un flottant quadruple précision. Il n'est pas garanti que sa gamme soit supérieure à un `double`. Cela dépend de la machine.

Le mot clé `unsigned` placé avant le type permet de déclarer des nombres non-signés. De la même façon, `signed` permet de préciser que le nombre est signé. En l'absence de spécification, tous les types sont signés, à l'exception de `char` qui est parfois non signé par défaut [5]. Il convient de noter que le type `char` peut servir à stocker des nombres entiers et peut servir comme tel dans des calculs. La valeur numérique correspondant à un caractère donné est en fait son code ASCII.

Les tailles citées ci-dessus sont valables sur une machine *standard* dont le mot élémentaire est 16 ou 32 bits et pour un compilateur *standard* : autant dire qu'il est préférable de vérifier avant...

Déclarations des variables

Une déclaration de variable se fait systématiquement **au début d'un bloc** avant toute autre expression [6].

La déclaration s'écrit `<type> <nom>;` où `<type>` est le type de la variable et `<nom>` est son nom. On peut regrouper plusieurs variables de même type en une seule déclaration en écrivant les variables séparées par des virgules : `int a, b, i, res;` par exemple. Remarquer que ceci est différent des paramètres d'une fonction, qui doivent tous avoir un type explicite (on écrit `int somme(int a, int b);`).

Toute variable déclarée dans un bloc est locale à ce bloc, c'est-à-dire qu'elle n'est pas visible en dehors de ce bloc. De plus, elle masque toute variable du même nom venant d'un bloc qui l'englobe. Ceci peut se voir sur un exemple :

5. Avec `gcc`, le type `char` est signé ou non selon les options de compilation utilisées sur la ligne de commande : `-fsigned-char` ou `-funsigned-char`.

6. Cette limitation ne fait plus partie de la norme ISO de 1999, mais la plupart des compilateurs l'impose encore.

Listing 26.6 – *Variables globales et locales*

```
#include <stdio.h>
int a, b;

dummy(int a) {
    /* la variable a globale est ici masquee
       par le parametre a, en revanche b est globale */
    /* on a a=3 a l'appel 1 (provient du c du main()), */
    /*  b=2 (globale) */
    /* et a=5 a l'appel 2 (provient du d du sous-bloc),
    /* b=2 (globale) */

    /* c et d ne sont pas visibles ici */
    printf("%d %d\n",a,b);
}

main() {
    int c, d;
    a=1; b=2; c=3; d=4;
    dummy(c); /* appel 1 */
    /* on a toujours a=1, b=2 globales
      et c=3, d=4 locales */
    {         /* sous-bloc */
        int d, b;
        d=5; /* d est locale a ce bloc et masque le d de main() */
        b=7; /* b est locale a ce bloc */
        dummy(d); /* appel 2 */
    }
    printf("%d\n",d); /* on revient a d=4 du bloc main() */
}
```

Ce programme affiche dans l'ordre :

```
3  2
5  2
4
```

Attention, la simple déclaration d'une variable ne provoque généralement aucune initialisation de sa valeur. Si cela est néanmoins nécessaire, on peut écrire l'initialisation avec une valeur initiale sous la forme `int a=3;`.

Les classes de stockage

On appelle de ce nom barbare les options qui peuvent figurer en tête de la déclaration d'une variable. Les valeurs possibles sont les suivantes :

auto Il est *facultatif* et désigne des variables locales au bloc, allouées automatiquement au début du bloc et libérées en fin de bloc. On ne l'écrit quasiment jamais.

register Cette déclaration est similaire à `auto`, mais pour un objet accédé très fréquemment. Cela demande au compilateur d'utiliser si possible un registre du processeur pour cette variable. Il est à noter que l'on ne peut demander l'adresse mémoire d'une variable (par l'opérateur & présenté en partie 26.5) ayant cette spécification, celle-ci n'existant pas.

static Les variables non globales ainsi déclarées sont allouées de façon statique en mémoire, et conservent donc leur valeur lorsqu'on sort des fonctions et des blocs et que l'on y entre à nouveau. Dans le cas des variables globales et des fonctions, l'effet est différent : les variables globales déclarées statiques ne peuvent pas être accédées depuis les autres fichiers source du programme.

extern Lorsqu'un source fait appel à une variable globale déclarée dans un autre source, il doit déclarer celle-ci **extern**, afin de préciser que la vraie déclaration est ailleurs. En fait une variable globale doit être déclarée sans rien dans un source, et déclarée **extern** dans tous les autres sources où elle est utilisée. Ceci peut se faire en utilisant le préprocesseur (cf. 26.6).

En règle générale, il est préférable d'éviter l'utilisation de la classe **register** et de laisser au compilateur le soin du choix des variables placées en registre.

Afin de simplifier la tâche de développement et pour limiter les risques d'erreurs, il est recommandé de déclarer avec le mot clé `static` en tête les variables et fonctions qui sont globales au sein de leur fichier, mais qui ne doivent pas être utilisées directement dans les autres sources.

En revanche, utiliser `static` pour des variables locales permet de conserver une information d'un appel d'une fonction à l'appel suivant de la même fonction sans nécessiter de variable globale. Ceci est à réserver à quelques cas très spéciaux : ne pas en abuser !

> La fonction `strtok()` de la bibliothèque standard utilise ainsi une variable statique pour retenir d'un appel à l'autre la position du pointeur de chaîne.
> D'autres qualificateurs existent, en particulier le terme `volatile` placé avant la déclaration permet de préciser au compilateur qu'il peut y avoir modification de la variable par d'autres biais que le programme en cours (périphérique matériel ou autre programme). Il évitera ainsi de réutiliser une copie de la variable sans aller relire l'original. Les cas d'utilisation en sont assez rares dans des programmes classiques. Seuls les développeurs du noyau ou de modules du noyau et ceux qui utilisent des mécanismes de mémoire partagée entre processus en ont régulièrement besoin.

Types complexes, déclarations de type

Le `C` est un langage dont la vocation est d'écrire du code de très bas niveau (manipulation de bits), comme cela est nécessaire par exemple pour développer un système d'exploitation [7], mais il dispose également de fonctionnalités de haut niveau comme les définitions de type complexe.

Trois familles de types viennent s'ajouter ainsi aux types de base, il s'agit des **types énumérés**, des **structures** et des **unions**. Ces trois types se déclarent de façon similaire. Notons aussi les tableaux dont la déclaration est réalisée par :

```
/* un tableau de 16 entiers */
int un_tableau[16];
```

7. `C` à été développé au départ comme le langage principal du système Unix.

```
/* un tableau de 32 pixels */
/* voir ci-apres           */
struct pixel mes_pixels[32];
```

Le type énuméré est en fait un type entier déguisé. Il permet en particulier de laisser des noms littéraux dans un programme pour en faciliter la lecture. Les valeurs possibles d'un type énuméré font partie d'un ensemble de constantes portant un nom, que l'on appelle des énumérateurs. L'exemple suivant définit un type énuméré correspondant à des booléens, noté **bool**, et déclare en même temps une variable de ce type nommée **bvar** :
`enum bool {FAUX, VRAI} bvar;`
Après cette déclaration il est possible de déclarer d'autres variables de ce même type avec la syntaxe `enum bool autrevar;`.

> Dans la première déclaration, il est permis d'omettre le nom du type énuméré (dans ce cas seule la variable **bvar** aura ce type) ou bien de ne pas déclarer de variable sur la ligne de déclaration d'énumération.

Dans une énumération, si aucune valeur entière n'est précisée (comme c'est le cas dans l'exemple précédent), la première constante vaut zéro, la deuxième 1 et ainsi de suite. La ligne ci-dessus est donc équivalente à (sans nommage du type)

```
enum {FAUX=0, VRAI=1} bvar;
```

Lorsque'elles sont précisées, les valeurs peuvent apparaître dans n'importe quel ordre :

```
enum {lun=1,ven=5,sam=6,mar=2,mer=3,jeu=4,dim=7} jour;
```

Une structure est un objet composé de plusieurs membres de types divers, portant des noms. Elle permet de regrouper dans une seule entité plusieurs objets utilisables indépendamment. Le but est de regrouper sous un seul nom plusieurs éléments de types différents afin de faciliter l'écriture et la compréhension du source.

Une structure `pixel` contenant les coordonnées d'un pixel et ses composantes de couleur pourra par exemple être :

```
struct pixel {
   unsigned char r,v,b;
   int x,y;
};
```

Par la suite, la déclaration d'une variable `p1` contenant une structure `pixel` se fera par `struct pixel p1;`.

> La remarque précédente concernant les possibilités de déclaration des énumérations s'applique également aux structures et aux unions.

Les accès aux champs d'une structure se font par l'opérateur `point` (`.`) . Rendre le point `p1` blanc se fera ainsi par `p1.r = 255 ; p1.v = 255 ; p1.b = 255 ;`

Une union est un objet qui contient, selon le moment, un de ses membres et un seul. Sa déclaration est similaire à celle des structures :

```
union conteneur {
   int vali;
   float valf;
   char valc;
};
union conteneur test;
```

En fait, il s'agit d'un objet pouvant avoir plusieurs types selon les circonstances.

On peut ainsi utiliser `test.vali` comme un `int`, `test.valf` comme un `float` ou `test.valc` comme un `char` après la déclaration précédente, mais un seul à la fois !

La variable `test` est juste un espace mémoire assez grand pour contenir le membre le plus large de l'union et tous les membres pointent sur la même adresse mémoire [8]. Il incombe donc au programmeur de savoir quel type est stocké dans l'union lorsqu'il l'utilise : c'est une des raisons qui rendent son usage peu fréquent.

> **Attention**, il est fortement déconseillé d'utiliser la valeur d'un membre d'une union après avoir écrit un autre membre de la même union (écrire un `int` et lire un `char` par exemple) car le résultat peut différer d'une machine à l'autre (et présente en général peu d'intérêt, sauf si l'on veut savoir l'ordre que la machine utilise pour stocker les octets dans un `long`...).

Déclaration de type

Celle-ci se fait au moyen du mot-clé `typedef`. Un nouveau type ainsi défini peut servir par la suite pour déclarer des variables exactement comme les types simples. On peut ainsi écrire

```
typedef unsigned char uchar;
```

afin d'utiliser `uchar` à la place de `unsigned char` dans les déclarations de variables...

La déclaration de type est très utile pour les énumérations, les structures et les unions. Dans notre exemple précédent, il est possible d'ajouter

```
typedef struct pixel pix;
```

pour ensuite déclarer `pix p1,p2;`. Une version abrégée existe également dans ce cas et permet en une seule fois de déclarer la sructure `pixel` et le type `pix` :

8. Si l'architecture de la machine utilisée le permet.

```
typedef struct pixel {
   unsigned char r,v,b;
   int x,y;
} pix;
```

Le comportement est alors exactement le même que ci-dessus : les déclarations `struct pixel p1;` et `pix p1;` sont équivalentes.

Il est possible aussi de supprimer complètement le nom de structure dans ce cas en écrivant :

```
typedef struct {
   unsigned char r,v,b;
   int x,y;
} pix;
```

Comme précédemment, les déclarations de variables ultérieures se feront par `pix p1;`.

> Mais **attention**, cette dernière écriture ne permet pas de réaliser les structures `autoréférentielles` qui sont présentées au 26.5.

Écriture des constantes

Les constantes entières sont écrites sous forme décimale, octale (base 8) ou hexadécimale (base 16). L'écriture décimale se fait de façon naturelle, l'écriture octale s'obtient en faisant précéder le nombre d'un `0` (par exemple 013 vaut en décimal 11), et l'écriture hexadécimale en faisant précéder le nombre du préfixe `0x`.

De plus, il est possible d'ajouter un suffixe `U` pour `unsigned` et/ou `L` pour `long` dans le cas où la valeur de la constante peut prêter à confusion. Ainsi `0x1AUL` est une constante de type `unsigned long` et valant 26.

Les constantes de type caractère peuvent être entrées sous forme numérique (par leur code ASCII, car du point de vue du compilateur ce sont des entiers sur un octet), il est aussi possible de les écrire entre apostrophes (e.g. `'A'`). Cette dernière écriture accepte également les combinaisons spéciales permettant d'entrer des caractères non-imprimables sans avoir à mémoriser leur code ASCII (voir au paragraphe 26.5). Le caractère `'\0'` vaut la **valeur** 0 (appelée NUL en ASCII) et sert de marqueur spécial [9].

Il faut bien remarquer que le type caractère correspond à un caractère unique (même si certains caractères sont représentés par des chaînes d'échappements comme `'\n'`) et non à une chaîne de caractères.

Attention à ne pas utiliser des guillemets " à la place des apostrophes, car "A" par exemple représente en `C` une chaîne de caractères constante contenant le caractere 'A'. C'est donc un pointeur vers une adresse mémoire contenant 'A' et non la valeur 'A' !

9. Le chiffre '0' vaut en fait en ASCII 48

Les constantes flottantes (qui correspondent à des nombres réels) peuvent utiliser une syntaxe décimale ou une notation scientifique avec la lettre `e` pour marquer l'exposant.

Il est possible d'affecter une valeur à une variable lors de sa déclaration. Ceci se fait le plus simplement possible par `int a=3;`. Dans le cas où cette variable est en fait une constante (c'est un comble !), c'est-à-dire qu'elle ne peut et ne doit pas être modifiée dans le corps du programme, on peut préciser `const` devant la déclaration :

```
const float e2 = 7.3890561;
const pi = 3.141592;
const nav = 6.02e23;
```

`const` permet selon sa position de déclarer des valeurs non modifiables ou des pointeurs non modifiables. L'exemple suivant explicite les différents cas :

```
char s,t;
const char c='a'; /* l'affectation initiale est possible */
const char *d;
char *const e=&t;

c='b'; /* erreur : c est const */
s=c;   /* affectation correcte de caractère */
d=&s;  /* affectation correcte de pointeur */
*d=c;  /* erreur : *d est un caractère constant */
*e=c;  /* correct : *e est un char */
e=&s;  /* erreur : e est un pointeur constant */
```

Cette même syntaxe d'affectation initiale peut se faire pour les structures, les énumérations et les tableaux. Pour les structures, les valeurs initiales doivent être écrites entre accolades, séparées par des virgules et dans l'ordre de la structure. Il est possible d'omettre les derniers champs de la structure, qui prennent alors par défaut la valeur zéro :

```
typedef struct {
   unsigned char r,v,b;
   int x,y;
} pix;

pix p1={2, 3, 4, 5, 6}; /* r=2 v=3 b=4 x=5 y=6 */
pix p2={1, 2};          /* r=1 v=2 b=0 x=0 y=0 */
```

`const` sert également à préciser que certains arguments de fonction (dans la déclaration, et s'il s'agit de pointeurs, cf. 26.5) désignent des objets que la fonction ne modifiera pas [10].

10. L'usage de `const` est particulièrement important pour les systèmes informatiques embarqués, où *le code exécutable et les constantes* peuvent être stockés en mémoire morte (ROM).

Le transtypage (*cast*)

Lors de calculs d'expressions numériques, les types flottants sont dits `absorbants` car la présence d'un unique nombre flottant dans une expression entraîne une évaluation complète sous forme de flottant.

Il est fréquent qu'une variable ait besoin d'être utilisée avec un autre type que son type naturel. Cela arrive par exemple si l'on veut calculer le quotient de deux entiers et stocker le résultat dans un réel (voir le paragraphe suivant).

Ce transtypage s'obtient en faisant précéder l'expression que l'on souhaite utiliser avec un autre type par ce type entre parenthèses. C'est par exemple le seul moyen de convertir un nombre flottant en entier, car une fonction `floor()` est bien définie dans la bibliothèque mathématique, mais elle retourne un nombre flottant ! On écrira ainsi `i = (int)floor(f);`.

> Le transtypage d'entier en flottant et réciproquement comme présenté ci-dessus est un cas très particulier, qui pourrait laisser croire que toutes les conversions sont possibles par ce biais. Attention donc, **dans la grande majorité des cas**, le transtypage conduit juste à interpréter le contenu de la mémoire d'une autre façon : c'est le cas par exemple si l'on essaie de transformer une chaîne de caractère `char s[10]` en un entier par `(int)s`, et le résultat n'a pas d'intérêt (en fait cela revient à lire l'adresse mémoire à laquelle est stockée `s` !). Pour effectuer les conversions, il convient d'utiliser les fonctions adéquates de la bibliothèque standard.

Les opérateurs

Les opérations arithmétiques en `C` sont d'écriture classique : `+, -, *, /` pour l'addition, la soustration, la multiplication et la division. Il est à noter que la division appliquée à des entiers donne le quotient de la division euclidienne. Pour forcer un résultat en nombre flottant (et donc une division réelle), il faut convertir l'un des opérandes en nombre flottant. Par exemple `3/2` vaut `1` et `3/2.0` ou `3/(float)2` vaut [11] `1.5`.

Les affectations s'écrivent avec le signe `=`. Une chose importante est à noter : une affectation peut elle-même être utilisée comme un opérande dans une autre opération, la valeur de cette affectation est alors le résultat du calcul du membre de droite. Ainsi on pourra par exemple écrire `a=(b=c+1)+2;` dont l'action est de calculer `c+1`, d'affecter le résultat à `b`, d'y ajouter 2 puis d'affecter le nouveau résultat à `a`. Il est à noter que ce type d'écriture, sauf cas particuliers, rend la lecture du programme plus difficile et doit donc être si possible évité.

Les opérations booléennes permettent de faire des tests. Le `C` considère le `0` comme la valeur `fausse` et **toutes** les autres valeurs comme vraies. Pour combiner des valeurs en tant qu'expression booléenne, on utilise le OU logique noté `||` et le ET logique noté `&&`. La négation d'une valeur booléenne s'obtient par `!` placé avant

11. Il s'agit ici d'un transtypage (*cast*).

la valeur. Il est important de retenir également que *les opérateurs logiques* && *et* || *arrêtent l'évaluation de leurs arguments dès que cela est possible* [12]. Cette propriété peut être intéressante, par exemple dans des conditions de fin de boucle du type SI *mon indice n'a pas dépassé les bornes* ET *quelque chose qui dépend de mon indice* ALORS *instructions*. En effet dans ce cas le calcul qui dépend de l'indice n'EST PAS effectué si l'indice a dépassé les bornes, ce qui est préférable !

Il reste à engendrer des grandeurs booléennes qui représentent quelque chose ! Ceci ce fait avec **les opérateurs de comparaison**. Le test d'égalité se fait par == (deux signes égal), le test d'inégalité (différent de) se fait par `!=`, la relation d'ordre par >, . >=, < et <=.

> **Attention** une erreur classique est d'écrire un seul signe égal dans un test. Comme le résultat d'une affectation est la valeur affectée et comme un nombre non nul est un booléen vrai, écrire :
>
> ```
> if (a == 2) {
>       printf("a vaut 2\n");
> }
> ```
>
> ou écrire :
>
> ```
> if (a = 2) {
>       printf("a vaut 2\n");
> }
> ```
>
> est très différent ! (la deuxième version est un test toujours vrai, et a vaut 2 après l'exécution).

Les opérations d'incrémentation et de décrémentation étant fréquentes sur les entiers, il est possible (et même recommandé) d'écrire `i++` plutôt que `i=i+1`. Comme dans le cas des affectations, l'opération d'incrémentation à une valeur, qui est la valeur avant incrémentation si le ++ est à droite de la variable, et est la valeur incrémentée si c'est `++i`. Un opérateur de décrémentation noté `--` existe également. En résumé :

```
           /* valeurs apres la ligne */
i = 2;     /* i=2 */
a = ++i;   /* i=3, a=3 */
b = a--;   /* b=3, a=2 */
b++;       /* b=4 */
```

On remarquera que, bien que l'opérateur ait une valeur de retour, celle-ci peut être ignorée. Ceci est vrai également pour les fonctions. Par exemple `printf()` que nous avons utilisé tout au début retourne normalement un `int`, mais le plus souvent on ignore cette valeur de retour.

12. Ceci s'appelle de *l'évaluation paresseuse* (lazy evaluation).

Dans la lignée des opérateurs d'incrémentation, il existe toute une famille **d'opérateurs de calcul-affectation** qui modifient une variable en fonction d'une expression. Leur syntaxe est de la forme :
`<variable> R= <expression>`
où R est une opération parmi +, -, *, /, %, >>, <<, &, | et ^. L'écriture précédente est alors équivalente à :
`<variable> = <variable> R <expression>`.

Quelques opérateurs spéciaux viennent s'ajouter à cela : nous avons déjà parlé des fonctions, **l'appel de fonction** est un opérateur (c'est même l'opérateur de priorité maximale). Il y a également l'opérateur **conditionnel** dont la valeur résultat provient d'une des deux expressions qu'il contient selon une condition :
`<Condition> ? <Expression si vraie> : <Expression si fausse>`
La condition est une expression booléenne. Elle est systématiquement évaluée (avec un arrêt dès que le résultat est certain comme nous l'avons vu précédemment) et selon sa valeur, une des deux expressions et une seulement est évaluée. Un exemple d'usage est la construction d'une fonction MIN :

```
int min(int a, int b) {
  return ( (a>b) ? b : a );
}
```

Un dernier opérateur souvent passé sous silence est **l'opérateur virgule**. Il permet de séparer une suite d'expressions qui seront nécessairement évaluées l'une après l'autre de la gauche vers la droite, le résultat final de cette séquence d'expression étant le résultat de la dernière évaluation (expression la plus à droite).

Les opérateurs concernant les pointeurs et les tableaux seront présentés en partie 26.5.

Les opérateurs sur les bits d'un mot sont extrêmement utiles en programmation bas-niveau et permettent généralement d'éviter la programmation en langage machine. Les opérations agissent bit à bit sur la représentation binaire de leur(s) opérande(s). On dispose ainsi de la complémentation (~) de tous les bits d'un mot, du ET bit à bit, du OU et du XOR (OU eXclusif) notés respectivement &, | et ^. Les opérateurs de décalage à gauche (équivalent à une multiplication par deux sur des entiers non signés) et à droite (division par 2) se notent << et >>. Quelques exemples :

```
unsigned char c, d;
c=0x5f;      /* en binaire 0101 1111 */
d = ~c;      /* d = 0xA0   1010 0000 */
d = d | 0x13; /*    0x13 = 0001 0011   -> d = 0xB3  1011 0011 */
c = d ^ (c << 2);  /* c decale a gauche 2 fois :    0111 1100 */
                                  resultat c= 0xCF 1100 1111 */
```

> **Attention** une autre erreur classique consiste à oublier un signe et à écrire `cond1 & cond2` au lieu de `cond1 && cond2` dans un test. Le compilateur ne peut bien sûr rien remarquer et l'erreur est généralement très difficile à trouver.

L'appel de fonction

En C, l'appel de fonction se fait par le nom de la fonction suivi de la liste des paramètres entre parenthèses. Une particularité très importante du langage est que les paramètres passés en argument de la fonction sont recopiés dans des variables locales à la fonction avant le début de son exécution. On appelle cela un **passage par valeur**. Ainsi, sans utiliser les pointeurs [13], le seul résultat qui puisse sortir d'un appel de fonction est sa valeur de retour.

La fonction se termine lorsque l'exécution atteint l'accolade fermante du bloc, et dans ce cas il n'y a pas de valeur de retour (la fonction doit être déclarée de type `void`). Si la fonction doit retourner une valeur, celle-ci doit être précisée par l'instruction `return valeur`. L'exécution de cette instruction termine la fonction. Il est possible d'avoir plusieurs instructions `return` dans une même fonction, par exemple dans le cas de tests.

> Dans le cas d'une fonction retournant `void` (sans valeur de retour), `return ;` permet de terminer la fonction.

Voici un exemple :

Listing 26.7 – *Passage par valeur*

```
#include <stdio.h>
int test(int a, int b) {  /* lors de l'appel, a=2 et b=7 */
   if (a==0) return b;
   /* la suite n'est execute que si a != 0 */
   a=b/a;
   b=a;
   return a;
}
int main(int argc, char *argv[]) {
   int x, y, z;
   x=2; y=7;
   z=test(x,y);  /* z=3 mais x et y ne sont pas modifies */
                 /* seule leur valeur a ete copiee dans  */
                 /* a et b de test() */
   printf("%d %d %d %d\n",x,y,z,test(0,z));
   /* affiche 2 7 3 3 */
}
```

Les fonctions peuvent bien sûr être récursives, comme le montre l'exemple du 26.1.

13. Voir 26.5. Ceux-ci permettent quelque chose de similaire au passage par variable (ou par référence).

Les fonctions *inline*

L'écriture de programmes lisibles passe par le découpage des algorithmes en de nombreuses fonctions, chacune formant une unité logique. Certaines de ces fonctions ne servent qu'une fois, mais on préfère ne pas intégrer leur code à l'endroit où elles sont appelées. À l'inverse, d'autres fonctions sont très simples (quelques lignes de programme, comme par exemple le calcul du minimum de deux valeurs) et sont utilisées un grand nombre de fois.

Dans les deux cas, chaque appel de fonction est une perte de temps par rapport à écrire le code directement à l'endroit de l'appel (le compilateur sauvegarde le contexte au début de l'appel de fonction, place éventuellement les arguments sur la pile, puis appelle la fonction) le retour de l'appel est également assez coûteux.

Pour éviter ce gâchis de puissance de calcul, la plupart des compilateurs essayent d'optimiser en plaçant le code de certaines fonctions directement à la place de leur appel. Ceci a de nombreuses limitations : il faut compiler avec des optimisations fortes et les fonctions concernées doivent être dans le même fichier source [14].

Pour aider le compilateur dans le choix des fonctions à optimiser ainsi, il est possible de préciser le mot-clef `inline` au début de la déclaration de la fonction. Ceci prend alors effet même si les optimisations ne sont pas demandées (ce n'est qu'une indication, comme la classe *register* pour les variables).

Ce mot-clef `inline` ne fait pas partie de la norme ANSI de 1989 mais a été, à notre connaissance, intégré dans la norme de 1999. Il est supporté par `gcc` et de nombreux autres compilateurs.

> Pour permettre l'appel de fonction depuis d'autres fichiers objet[a], le compilateur conserve tout de même une version sous forme de fonction classique, sauf si la fonction est de plus déclarée `static inline`.
>
> *a*. C'est également nécessaire si la fonction est appelée quelque part par pointeur de fonction, cf. 26.5.

On pourra avec avantage déclarer ainsi une fonction minimum de deux entiers sous la forme de fonction inline (plutôt que de macro, ce qui conduit à de fréquentes erreurs) :

```
static inline int min(int a, int b) {
  return ( (a>b) ? b : a );
}
```

Ces fonctions déclarées `static inline` sont les seules fonctions que l'on pourra écrire dans les fichiers `.h` afin de pouvoir les utiliser comme on le fait avec des macros.

14. Ceci n'empêche pas la compilation séparée, mais tout appel de fonction entre des fichiers objets différents est obligatoirement effectué sous la forme d'un vrai appel de fonction. À cette fin, le compilateur conserve toujours une version de la fonction dont l'appel soit réalisable *normalement*.

Les priorités

Voici un tableau récapitulatif donnant tous les opérateurs et leur priorité (la première ligne du tableau est la plus prioritaire, la dernière est la moins prioritaire), les opérateurs ayant des priorités égales figurent dans une même ligne et sont évalués dans le sens indiqué dans la colonne associativité :

Opérateurs	Fonction	Associativité	priorité
`()`	appel de fonction	→	maximale
`[]`	élément de tableau	→	
`.`	champ de structure ou d'union	→	
`->`	champ désigné par pointeur	→	

Opérateurs	Fonction	Associativité	priorité
`!`	négation booléen	←	
`~`	complémentation binaire	←	
`-`	opposé	←	
`++`	incrémentation	←	
`--`	décrémentation	←	
`&`	adresse	←	
`*`	déréférenciation de pointeur	←	
`(type)`	transtypage	←	

Opérateurs	Fonction	Associativité	priorité
`*`	multiplication	→	
`/`	division	→	
`%`	reste euclidien	→	

Opérateurs	Fonction	Associativité	priorité
`+`	addition	→	
`-`	soustraction	→	

Opérateurs	Fonction	Associativité	priorité
`<<`	décalage à gauche	→	
`>>`	décalage à droite	→	

Opérateurs	Fonction	Associativité	priorité
`<`	strictement inférieur	→	
`<=`	inférieur ou égal	→	
`>`	strictement supérieur	→	
`>=`	supérieur ou égal	→	

Opérateurs	Fonction	Associativité	priorité
==	égal	→	
!=	différent	→	

Opérateurs	Fonction	Associativité	priorité
&	ET bit à bit	→	
^	OU eXclusif (XOR) bit à bit	→	
|	OU bit à bit	→	
&&	ET booléen	→	
||	OU booléen	→	

Opérateurs	Fonction	Associativité	priorité
? :	conditionnelle	←	
=	affectation	←	
*=	affectations avec calcul	←	
/=		←	
%=		←	
-=		←	
+=		←	
<<=		←	
>>=		←	
&=		←	
|=		←	
^=		←	
,	séquence d'expressions	→	minimale

Les opérateurs concernant les pointeurs sont présentés au 26.5.

Ce tableau est très important, et son ignorance peut conduire parfois à des erreurs difficiles à détecter. Citons par example `a=*p++` : les opérateurs `++` et $*$ sont de même priorité, donc l'expression est évaluée dans l'ordre d'associativité (de droite à gauche), et donc p est incrémenté, et l'ancienne valeur de p est déréférencée, le résultat étant stocké dans a. Si l'on veut incrémenter la valeur pointée par p (et non p lui-même), il faudra écrire `a=(*p)++`. Le résultat stocké dans a est le même dans les deux cas.

En cas de doute, il ne faut pas hésiter à mettre quelques parenthèses (sans abuser toutefois) car cela peut aider à la compréhension. On évitera ainsi d'écrire : `a=++*p>b|c+3` !
On évitera aussi : `a=((++(*p)>b)|(c+3))` et on préfèrera quelque chose comme : `a = ++(*p)>b | (c+3)`.

26.4 Tests et branchements

L'instruction `if`

Nous avons vu lors de l'examen des opérateurs l'expression conditionnelle. Il existe également bien sûr une structure de contrôle qui réalise l'exécution conditionnelle. Sa syntaxe est la suivante :

```
if (<condition>) {
   <instructions ...>
}
```

La condition est testée comme un booléen, c'est-à-dire qu'elle est considérée comme vraie si son évaluation n'est pas nulle. Dans le cas où une seule instruction est à exécuter quand la condition est vraie, on trouvera souvent cette instruction toute seule à la place du bloc entre accolades.
La version avec alternative :

```
if (<condition>) {
   <instructions ...>
} else {
   <instructions ...>
}
```

Là encore il est possible de remplacer l'un ou l'autre (ou les deux) blocs entre accolades par une unique instruction. Attention dans ce cas, il faut terminer cette instruction par un point-virgule.
Avec de multiples alternatives :

```
if (<cond1>) {
   <instructions ...> /* Si cond1 est vraie */
} else if (<cond2>) {
      <instructions ...> /* si cond1 fausse et cond2 vraie */
   } else {
      <instructions ...> /* si cond1 et cond2 fausses */
   }
}
```

De nombreux programmes `C` utilisent des tests sans instruction de comparaison explicite, voire des tests qui sont le résultat d'opérations d'affectation, par exemple :

```
if (a = b - i++) i--; else a=b=i++;
```

Ceci utilise le fait qu'une valeur non nulle est considérée comme vraie. C'est assez difficile à lire [15], aussi il sera préférable d'écrire cette version équivalente :

15. L'un des exemples les plus réussis dans l'illisibilité se trouve ici : `http://www.cwi.nl/~tromp/tetris.html`

```
i++;
a= b-i;
if (a != 0) {
     i--;
} else {
     i++;
     b = i;
     a = b;
}
```

L'instruction `switch()`

```
switch (<expression>) {
   case <val 1>:  <instructions ...>
   case <val 2>:  <instructions ...>
   ...
   case <val n>:  <instructions ...>

   default:  <instructions ...>
}
```

L'instruction `switch` permet de diriger le programme vers une séquence d'instructions choisie parmi plusieurs, suivant la valeur d'une expression qui doit être de type entier.

Le branchement se fait à l'étiquette `case` dont la valeur est égale à l'expression de tête (il ne doit y avoir au plus qu'une étiquette qui correspond). Si aucune étiquette ne correspond, le saut se fait à l'étiquette `default` si elle existe.

Attention, une fois le branchement effectué, le programme continue de s'exécuter dans l'ordre, y compris les `case` qui peuvent suivre. Pour empêcher ce comportement, on utilise souvent `break` à la fin des instructions qui suivent le `case` : son effet est de sauter directement après le bloc du `switch` (Voir 26.4).

Un exemple complet d'utilisation standard :

Listing 26.8 – *Branchements*

```
int val,c;
c=1;
val = mafonctiondecalcul(); /* une fonction definie ailleurs */
switch (val) {
    case 0:
    case 1: printf("Resultat 0 ou 1\n");
              c=2;
              break;
    case 2: printf("Resultat 2\n");
              break;
```

```
    case 4: printf("Resultat 4\n");
              c=0;
              break;
    default: printf("Cas non prevu\n");
}
```

Les boucles

La boucle `for`

C'est une boucle beaucoup plus générale que dans d'autres langages. Son rôle ne se limite pas à la simple répétition en incrémentant un entier. Son utilisation est la suivante :

```
for ( <instruction initiale> ;
         <condition de poursuite> ;
             <instruction d'avancement> ) {
    <instructions ...>
}
```

L'*instruction initiale* est exécutée avant le début de la boucle, la *condition de poursuite* est testée à chaque début de boucle et l'*instruction d'avancement* est exécutée en dernière instruction à chaque tour de boucle.

L'équivalent de la boucle d'incrémentation d'une variable entière des autres langages s'écrira par exemple :

```
for (i=0; i<100; i++) {
    <instructions...> /* exécutées pour i=0, i=1 ... i=99 */
}
```

On peut omettre n'importe laquelle des trois expressions de la boucle `for`. Si c'est la condition de poursuite qui est omise, la boucle se comporte comme si cette condition était toujours vraie. Ainsi une boucle infinie [16] peut s'écrire :

```
for (;;) {
    <instructions ...>
}
```

16. Qui ne se termine jamais si l'on utilise pas les instructions de branchement qui seront présentées dans le paragraphe traitant des branchements.

La boucle `while`

```
while (<condition de poursuite>) {
    <instruction ...>
}
```

qui est complètement équivalente à (remarquer que les expressions de la boucle `for` sont facultatives) :

```
for (;<condition de poursuite>;) {
    <instructions ...>
}
```

De façon similaire, il est possible de réécrire une boucle `for` générique sous la forme d'une boucle `while`. L'écriture générique précédente est équivalente à :

```
<inst. initiale>;
while (<cond. de poursuite>) {
    <instructions...>
    <inst. d'avancement>;
}
```

La boucle `do...while`

Ici le test est effectué en fin de boucle. Cette boucle est donc, contrairement aux deux boucles précédentes, exécutée au moins une fois, même si la condition est fausse dès le début de la boucle. Voici la syntaxe :

```
do {
    <instructions ...>
} while (<condition de poursuite>);
```

Le style d'écriture habituel du `C` utilise extrêmement rarement ce type de boucle.

Les branchements

Ce sont des instructions qui modifient le déroulement d'un programme sans condition.

L'instruction **continue** ne peut être présente qu'à l'intérieur d'une boucle. Elle conduit à un branchement vers la fin de la plus petite boucle qui l'entoure, mais ne sort pas de la boucle (ainsi l'*instruction d'avancement* si elle existe est effectuée, puis la boucle reprend).

L'instruction **break** peut apparaître dans une boucle ou dans une instruction `switch`. Elle effectue un saut immédiatement après la fin de la boucle ou du `switch`. Son usage est indispensable dans le cas des boucles infinies.

L'instruction **goto** <**label**> quant à elle peut apparaître n'importe où. Elle effectue un saut à l'instruction qui suit l'étiquette `<label>:` (cette étiquette se note par le label suivi de deux-points).

L'usage du `goto` est fortement déconseillé car il conduit fréquemment à un source illisible. Il peut dans quelques rares cas rendre service en évitant d'immenses conditions, ou de nombreuses répétitions d'un même petit morceau de code.

L'exemple suivant montre l'utilisation de **break** et de **continue** au sein d'une boucle `for`. On trouvera plus fréquemment l'instruction **break** dans des boucles infinies.

Listing 26.9 – *Sortie de boucle prématurée*

```
for (i=0; i<100; i++) {
/* si i<5 la fin de la boucle n'est pas executee */
    if (i<5) continue;
    if (i==10) break; /* si i vaut 10 la boucle se termine */
    printf("%d\n",i);
}
```

Le résultat de cette boucle est l'affichage des nombres **5, 6, 7, 8, 9**. La condition de poursuite de la boucle `for` est en fait totalement inutile ici.

26.5 Tableaux et pointeurs

Déclaration et utilisation des tableaux

En C, un tableau à une dimension de 10 entiers sera déclaré par la syntaxe `int x[10]`. Il s'agit d'une zone de mémoire de longueur dix entiers qui est allouée automatiquement, `x` étant comme expliqué ci-dessous un pointeur constant vers le premier entier de cette zone.

Les dix cellules du tableau sont alors accessibles par `x[0]` à `x[9]`. Il convient de noter qu'**aucune vérification n'est faite sur la validité du numéro de la case demandée** du tableau. L'accès à une case au delà du dernier élément du tableau (ou avant le premier, par exemple `x[-1]`) peut conduire à une erreur, mais il se peut aussi que cela accède à d'autres variables du programme et dans ce cas, l'erreur est très difficile à détecter...

Les tableaux de plusieurs dimensions existent également, par exemple :

```
char pix[128][256]
```

L'accès aux cases de ce tableau se fait alors par `pix[i][j]`, `i` et `j` étant des entiers. Il faut voir ce type d'écriture comme l'accès à un tableau à une seule dimension de 128 éléments qui sont chacuns des tableaux de 256 caractères.

L'affectation de valeurs initiales dans un tableau à une dimension se fait au moyen d'une liste de valeurs entre accolades :

```
int a[10]={1,2,3,4};
```

Dans cet exemple, `a[0]`, `a[1]`, `a[2]` et `a[3]` sont initialisés, `a[4]` à `a[9]` ne le sont pas. On peut omettre d'écrire la taille du tableau, le compilateur la calcule alors en comptant le nombre d'éléments fournis pour l'initialisation : `int a[]={1,2,3,4}` conduit alors à un tableau de quatre éléments seulement. Pour les tableaux de plusieurs dimensions, le dernier indice correspond au nombre de colonnes du tableau :

```
int b[2][4]={{1,2,3,4}, {5,6,5,6}}; /* 2 lignes, 4 colonnes */
```

Dans ce cas, seul le premier indice peut être omis, le compilateur comptant qu'il y a deux lignes, mais ne comptant pas les colonnes sur chaque ligne. Il est même possible dans ce cas de ne pas assigner tous les éléments de chaque ligne ! Ainsi

```
int b[][4]={{1,2}, {5,6,5}};
/* 2 lignes (comptees par le compilateur), 4 colonnes */
```

conduit à un tableau de deux lignes et quatre colonnes dont seuls les deux premiers éléments de la première ligne sont affectés, ainsi que les trois premiers éléments de la deuxième ligne [17].

Les pointeurs

Un pointeur est en fait une variable spéciale dans laquelle on ne stocke pas la valeur que l'on désire conserver, mais seulement son adresse en mémoire. En `C` un *pointeur* sur un objet de type `typ` est une variable de type noté `typ *`. Ainsi la déclaration `int *x;` définit `x` comme un *pointeur* sur une valeur de type `int`, c'est-à-dire comme l'adresse mémoire d'un entier.

L'accès au contenu d'un pointeur se fait par déréférenciation avec l'opérateur étoile. La valeur pointée par `x` peut donc être accédée par l'expression `*x`.

Un pointeur est une variable qui a sa propre adresse, et qui contient l'adresse de la variable sur laquelle il pointe. Ainsi, la déclaration `int *x;` alloue la case mémoire correspondant au pointeur `x`, mais n'initialise pas sa valeur. Or cette valeur doit être l'adresse de l'entier pointé par `*x`. En conséquence, la déréférenciation de `x` provoquera probablement une erreur (ou un résultat stupide...) tant que l'on n'a pas écrit explicitement dans le pointeur une adresse valide.

Pour initialiser la valeur du pointeur `x`, il faut ainsi utiliser une adresse mémoire valide, par exemple, celle d'une autre variable. À cet effet, l'adresse d'une variable s'obtient par l'opérateur `&`. Ainsi si l'on a déclaré

```
int y,*x;
```

on peut écrire

17. Les variables tableaux globales sont par défaut initialisées à zéro, ainsi que les variables locales déclarées `static`, en revanche les variables tableaux locales *normales* (non statiques) NE sont PAS initialisées.

```
x = &y;
```

Le pointeur `x` pointe alors sur la case mémoire correspondant à la valeur de la variable `y`. Tant que le pointeur `x` n'est pas modifié, `*x` et `y` ont toujours la même valeur (c'est la même case mémoire). On peut voir `*` comme l'opérateur inverse de `&`. On a ainsi `*(&y)` qui vaut `y`, et de même `&(*x)` qui vaut `x`.

> Un pointeur est en fait une variable presque comme les autres. Il s'agit d'un entier de la taille d'une adresse mémoire de la machine (le plus souvent 32 bits). La variable pointeur `x` de notre exemple précédent est un entier de signification spéciale. Il est donc possible de prendre l'adresse d'un pointeur ! Le pointeur `&x` est alors un pointeur vers un pointeur sur un entier, il contient l'adresse de l'adresse de l'entier. Son type est `int **`.

Ces possibilités sont tout particulièrement intéressantes pour les appels de fonction. En effet, si on passe en argument d'une fonction l'adresse d'une variable, la fonction va pouvoir modifier la variable locale par l'intermédiaire de son adresse. L'adresse est recopiée dans un pointeur local, on ne peut donc pas la modifier, mais en revanche la case mémoire pointée est la même et peut donc être modifiée. Voici un exemple :

Listing 26.10 – *Utilisation des pointeurs*

```
int exemple(int x, int *y)
{
   x += 3;
   *y += x;
   return(x);
}

main()
{
   int a,b,c;

   a = 2;
   b = 1;
   c = exemple(a,&b);
   printf("a=%d b=%d c=%d\n");
   exit(0);
}
```

Dans cet exemple, le programme affiche `a=2 b=6 c=5`. La variable `b` a été modifiée par l'appel de fonction : on parle alors d'*effet de bord*.

> Pour écrire une fonction qui modifie la valeur d'un pointeur (et pas seulement l'élément pointé comme ci-dessus), il faudra passer en paramètre à cette fonction l'adresse du pointeur à modifier, c'est-à-dire un pointeur sur le pointeur.

Lorsque l'on définit des pointeurs sur des structures, il existe un raccourci particulièrement utile (`st` est ici un pointeur sur une structure contenant un champ nommé `champ`) :
`(*st).champ` est équivalent à `st->champ`, l'opérateur flêche s'obtenant par la succession des signes `-` et `>`.

> Les pointeurs sont également utiles dans les appels de fonction même si la fonction n'a pas à modifier la valeur pointée : il est en effet beaucoup plus efficace de passer une adresse à une fonction que de recopier un gros bloc de mémoire (on évite ainsi généralement de passer des structures entières en paramètre, on préfère un pointeur sur la structure). L'accès aux champs d'une structure référencée par un pointeur est ainsi le cas d'écriture le plus général, d'où la syntaxe spéciale.
>
> Lorsque la fonction ne modifie pas l'objet pointé, on utilisera avantageursement le mot clé `const` qui permet au compilateur de vérifier qu'aucune écriture n'est faite directement dans l'objet pointé (ce n'est qu'une vérification rudimentaire).
>
> Une dernière utilisation commune des pointeurs est le cas de fonctions pouvant prendre en paramètre des zones mémoires dont le type n'est pas imposé. On fait cela au moyen du pointeur générique de type `void *`, qui peut être affecté avec n'importe quel autre type de pointeur. Un exemple typique de cette utilisation est la fonction `memset()` (définie dans `<string.h>`) qui permet d'initialiser un tableau de données de type quelconque.

Allocation dynamique de mémoire

Pour pouvoir stocker en mémoire une valeur entière, il faut donc explicitement demander une adresse disponible. Il existe heureusement un autre moyen d'obtenir des adresses mémoire valables, c'est le rôle de la fonction `malloc()`. Son prototype est `void *malloc(size_t size);`. Le type `size_t` est un type spécial, on pourra retenir qu'il s'agit en fait d'un entier codant un nombre d'octets mémoire. La taille d'un objet de type donné s'obtient par l'opérateur `sizeof()`, appliqué à la variable à stocker ou à son type.

Ainsi on pourra écrire

```
int *x;
   /* on peut aussi écrire x=(int *)malloc(sizeof(*x)); */
x = (int *)malloc(sizeof(int));
*x = 2324;
```

ce qui :

1. Déclare la variable `x` comme un *pointeur* sur une valeur de type `int`.
2. Affecte à `x` une adresse dans la mémoire qui est réservée pour le stockage d'une donnée de taille `sizeof(int)`, donc de type `int`.
3. Place la valeur 2324 à l'adresse `x`.

L'appel `malloc(s)` réserve en mémoire un bloc de taille `s`. On remarque que la fonction `malloc()` retourne un pointeur de type `void *` : on appelle cela un pointeur générique (c'est le type de pointeur que manipulent les fonctions qui doivent travailler indépendamment de l'objet pointé). Il faut bien entendu le convertir dans le type du pointeur que l'on désire modifier et ceci se fait par transtypage.

La mémoire ainsi allouée le reste jusqu'à la fin du programme. C'est pour cela que lorsqu'on n'a plus besoin de cette mémoire, il convient de préciser qu'elle peut à nouveau être utilisée. Ceci se fait par la fonction `free()` dont l'argument doit être un pointeur générique. La taille n'a pas besoin d'être précisée car elle est mémorisée lors de l'appel à `malloc()`. Dans le cas présent, la libération de la mémoire se ferait par `free((void *)x);`.

Du fait des méthodes utilisées pour gérer cette allocation dynamique, il est préférable d'éviter de faire de très nombreuses allocations de très petits blocs de mémoire. Si cela est nécessaire, on écrira sa propre fonction de gestion mémoire, en faisant les allocations et libérations avec `malloc()` et `free()` par gros blocs [18].

Arithmétique sur les pointeurs

Un pointeur est en fait un entier un peu spécial puisqu'il contient une adresse mémoire. Certaines opérations arithmétiques sont disponibles afin de permettre la manipulation aisée de blocs de mémoire contenant plusieurs éléments consécutifs du même type.

On dispose ainsi de l'addition d'un pointeur `p` et d'un entier `n` qui donne un pointeur de même type que `p`, pointant sur le n-ième élément du bloc commençant à l'adresse pointée par `p`. Par exemple `p+0` vaut `p` et pointe donc sur l'élément numéro zéro, `p+1` pointe sur l'élément suivant... Ceci fonctionne quelle que soit la taille de l'élément pointé par `p`, `p+n` provoquant ainsi un décalage d'adresses correspondant en octets à `n` fois la taille de l'objet pointé par `p`.

Il est possible aussi d'effectuer la soustraction de deux pointeurs de même type, et cela donne un résultat entier qui correspond au nombre d'éléments qui peuvent être placés entre ces deux pointeurs. L'exemple suivant montre une façon d'initialiser un bloc mémoire (ce n'est pas la méthode la plus simple pour y arriver, mais elle est efficace) :

Listing 26.11 – *Manipulation d'adresses*

```
#define TAILLE 1000
typedef struct pixel {
   unsigned char r,v,b;
   int x,y;
```

18. L'inefficacité de la gestion par petits blocs vient entre autre de l'ensemble des paramètres stockés pour chaque `malloc()` effectué, qui peut atteindre 16 octets ! D'où, pour allouer un bloc de 1024*1024 éléments de type char, environ 1 Mo si cela se fait par un appel `malloc(1024*1024)` et environ 17 Mo si cela est fait en 1024*1024 appels à `malloc(1)`...

```
} pix;

main() {
   pix *p;
   pix *q;

   p=(pix *)malloc(sizeof(pix)*TAILLE);

   for (q=p;q-p<TAILLE;q++) {
       q->r=q->v=q->b=0;
       q->x=q->y=0;
   }
    ...
}
```

Il existe des similarités entre les tableaux et les pointeurs. En particulier, un tableau à une dimension est en fait un pointeur constant (il n'est pas possible de modifier l'adresse sur laquelle il pointe) vers la première case du tableau. Ainsi il y a une équivalence totale entre les écritures `p[i]` et `*(p+i)` lorsque `p` est un pointeur ou un tableau monodimensionnel et `i` un entier (`char`, `int` ou `long`) [19].

> **Attention**, il n'y a pas équivalence simple entre une déréférenciation de pointeur et un accès à un tableau à plusieurs dimensions. Il faut en fait voir un tableau à deux dimensions `int tab[3][5]` comme un tableau de trois éléments qui sont des tableaux de cinq entiers (et non l'inverse). Ces trois éléments sont stockés les uns à la suite des autres en mémoire.
>
> Les écritures `tab[i][j]`, `*(tab[i]+j)`, `*(*(tab+i) +j)` et `*((int *)tab+5*i+j)` sont alors équivalentes, mais il reste fortement conseillé pour la lisibilité de se limiter à la première.
>
> De façon similaire, le passage d'un tableau en paramètre à une fonction peut se faire de façon équivalente sous la forme d'un pointeur (`void test(int *a)`) et sous la forme d'un tableau sans taille (`void test(int a[])`). Dans le cas de tableaux à plusieurs dimensions... la deuxième solution est très fortement déconseillé, on vous aura prévenu ! Si le type est un simple pointeur vers le début du tableau à deux dimensions, il faudra donc fournir également à la fonction la taille du tableau (tout ceci peut se mettre dans une jolie structure, comme on le fait par exemple pour des images).

Les chaînes de caractères

Les chaînes de caractères sont représentées en `C` sous la forme d'un pointeur sur une zone contenant des caractères. La fin de la chaîne est marquée par le caractère spécial `'\0'`. Il est possible d'utiliser la déclaration sous forme de tableau : `char`

19. On a même équivalence avec `i[p]` bien que cette dernière écriture soit à éviter !

`ch[256];` définit une chaîne dont la longueur maximale est ici de 256 caractères, en comptant le caractère de terminaison. En fait, par convention, toute chaîne de caractère se termine par le caractère **nul** (pas le chiffre `'0'`, mais l'entier 0 noté souvent pour cette utilisation `'\0'`). La chaîne de caractères peut aussi se déclarer sous la forme d'un pointeur `char *`, vers une zone qu'il faut allouer avec `malloc()`.

On peut écrire des chaînes constantes délimitées par des guillemets. L'expression `"chaine\n"` est donc une chaîne comportant le mot `chaine` suivi d'un retour chariot, et comme toujours terminée par un caractère `'\0'`.

La fonction `strlen()` (déclarée dans <string.h> dans la bibliothèque standard) retourne la longueur de la chaîne `s` jusqu'au premier code `'\0'` [20]. La fonction `strcmp(s1,s2)` permet de comparer deux chaînes. Elle retourne -1 si lexicographiquement `s1`<`s2`, 0 si les deux chaînes sont identiques, et +1 si `s1`>`s2`.

Pour l'anecdote, cette fonction est équivalente à `strlen()` (!) :

```
int mystrlen(char *s) {
       int longueur;
       for (longueur=0; *s++;longueur++) ;
       return longueur;
}
```

Les structures autoréférentielles

Les structures autoréférentielles [21] sont des structures dont un des membres est un pointeur vers une structure du même type. On les utilise couramment pour programmer des listes chaînées et pour les arbres.

Voici un exemple simple mais complet, dans lequel la liste initiale est allouée en un bloc consécutif et chaînée, puis ensuite utilisée comme une liste chaînée quelconque :

Listing 26.12 – *Structures autoréférentielles*

```
#include <stdio.h>
#include <stdlib.h>

typedef struct liste {
   struct liste *suivant;
   int valeur;
} tliste;


main() {
```

20. Il s'agit du nombre de caractères effectivement utiles dans la chaîne et non de la taille du bloc mémoire alloué comme le retournerait `sizeof()`.

21. On dit aussi parfois `structures récursives` par abus de langage.

```
   tliste *maliste;
   tliste *temp;
   int i,som;

   maliste=(liste *)malloc(100*sizeof(liste));
   /* chainage des elements de la liste */
   for(i=0;i<99;i++) {
       (maliste+i)->suivant=(maliste+i+1);
       (maliste+i)->valeur=i;
   };
   (maliste+99)->valeur=99;
   (maliste+99)->suivant=NULL;

   som=0;
   /* somme des elements de la liste */
   /* fonctionne par le chainage et  */
   /* non par les positions en memoire */
   for (temp=maliste; temp; temp=temp->suivant) {
      som += temp->valeur;
   }
   printf("somme: %d\n",som);

   exit(0);
}
```

Dans la déclaration d'une structure autoréférentielle, il n'est pas possible d'utiliser le type `tliste` défini par le `typedef` comme on le fait pour les déclarations de variables du `main()`, il est possible en revanche d'utiliser le type structuré `struct liste` en cours de déclaration.

Les pointeurs de fonction

Il est parfois souhaitable qu'une fonction générique ait besoin de pouvoir effectuer le même traitement à peu de choses près. C'est par exemple le cas d'une fonction de tri générique, qui aura besoin d'une fonction de comparaison, différente selon l'application [22].

Pour passer une fonction en paramètre à une autre fonction, il existe des pointeurs de fonctions, qui sont en fait des pointeurs contenant l'adresse mémoire de début de la fonction [23]. Afin que le compilateur connaisse les arguments de la fonction pointée, ceux-ci font partie de la déclaration du pointeur. On aura par exemple :

22. C'est comme cela que fonctionne la fonction de tri `qsort()`.
23. Dans le cas où la fonction a été déclarée `inline`, c'est la version sous forme de fonction standard qui est utilisée.

Listing 26.13 – *Pointeur de fonctions*

```
#include <stdio.h>
#include <stdlib.h>

/* fonction qui effectue la derivation numerique partielle */
/* par difference finie d'une fonction de x et y selon x */
/* fonc est le nom local de la fonction a deriver */
/* x,y est le point de derivation et h est l'intervalle */
float numderiv(float (*fonc) (float, float),
               float x, float y, float h) {
   return (fonc(x+h,y)-fonc(x-h,y))/2/h;
   /* on pourrait aussi ecrire : */
   /*   return ((*fonc)(x+h,y)-(*fonc)(x-h,y))/2/h; */
   /*   mais attention aux parentheses ! */
}

/* une fonction de deux variables */
float f1(float x, float y) {
   return x*y*y;
}
/* La meme fonction inversee */
float f2(float x, float y) {
   return f1(y,x);
}
/* une autre fonction */
float f3(float x, float y) {
   return 3*x*x;
}

main() {
   float x,y;
   x=3.0; y=2.0;
   printf("x:%f y:%f  f(x,y):%f    df/dx(x,y):%f\n",x,y,f1(x,y),
                                       numderiv(f1, x, y, 0.1));
   /* on pourrait aussi ecrire numderiv( &f1 , x, y, 0.1)); */
   printf("x:%f y:%f  f(x,y):%f    df/dy(x,y):%f\n",x,y,f2(y,x),
                                       numderiv(f2, y, x, 0.1));
   printf("x:%f y:%f f3(x,y):%f   df3/dx(x,y):%f\n",x,y,f3(x,y),
                                       numderiv(f3, x, y, 0.1));
   exit(0);
}
```

Qui affiche :

```
x:3.000000 y:2.000000  f(x,y):12.000000    df/dx(x,y):3.999996
x:3.000000 y:2.000000  f(x,y):12.000000    df/dy(x,y):11.999994
x:3.000000 y:2.000000 f3(x,y):27.000000   df3/dx(x,y):17.999982
```

Il est possible de passer une fonction en argument (donc un pointeur de fonction) à une autre fonction indifféremment par `f1` ou `&f1`. Ceci car le compilateur sait qu'il s'agit nécessairement d'un pointeur, la fonction n'étant pas une variable. De la même façon, lors de son utilisation, les syntaxes `fonc()` et `(*fonc) ()` sont équivalentes (noter les parenthèses dans la deuxième écriture, nécessaires du fait des priorités des opérateurs).

> Pour les amateurs de programmes illisibles, il est possible d'écrire les appels de fonctions normaux de plusieurs façon aussi (en utilisant `gcc`, ceci n'est pas valable sur tous les compilateurs) : `f1(y,x)` est équivalent à `(*f1)(y,x)`, et même à `(* (* (&f1))) (y,x)`.

26.6 Le préprocesseur

Lors de la compilation d'un fichier source en `C`, il se déroule en fait plusieurs opérations successives. La première phase est une passe de remplacement textuel uniquement et est effectuée par le préprocesseur. Il est possible de voir le code source en sortie du préprocesseur avec la commande `gcc -E`.

Les lignes que le préprocesseur interprète commencent par un dièse (#). Les commmandes du préprocesseur s'étendent chacune sur une ligne entière et une seule ; la seule façon d'écrire une commande du préprocesseur sur plusieurs lignes est d'empêcher l'interprétation du caractère de fin de ligne en le faisant précéder d'un caractère \ (tout à fait à la fin de la ligne).

Trois types d'opérations sont effectuées par le préprocesseur : **l'inclusion de fichiers**, **la substitution des macros** et **la compilation conditionnelle**.

L'inclusion de fichiers

La directive `#include <nom de fichier>` permet d'inclure le texte du fichier référencé à la place de la commande. Deux syntaxes existent pour cette commande : `#include <toto.h>` et `#include "toto.h"` qui diffèrent par les répertoires dans lesquels le préprocesseur va chercher `toto.h`.
On retiendra en général `#include <stdio.h>` pour les en-têtes du système (le compilateur ne va pas chercher dans le répertoire courant, mais uniquement dans les répertoires standards du système tels que `/usr/include`), et `#include "toto.h"` pour les en-têtes du projet, cette dernière syntaxe forçant le compilateur à chercher le fichier d'abord dans le répertoire courant.

Les macros (*#define*)

Une macro permet un remplacement d'un mot particulier dans tout le texte qui suit. Par exemple, la définition suivante remplace dans toute la suite le mot `test` par `printf("Ceci est un test\n")` :

```
#define test printf("Ceci est un test\n");
```

> Il n'est pas possible dans la suite du code de redéfinir de la même façon `test`, il faut commencer pour cela par supprimer la définition courante par `#undef test`. Appliquer `#undef` à un identificateur inconnu n'est pas une erreur, c'est simplement sans effet.

Les macros permettent également de définir des remplacements plus intelligents, avec des arguments variables. Ainsi la fonction MIN décrite précédemment (paragraphe sur les opérateurs) peut s'écrire sous forme de macro :
`#define MIN(a,b) ((a)>(b)?(b):(a))`

Tout écriture dans la suite du texte de `MIN(expr1,expr2)` fera le remplacement textuel en écrivant l'expression de remplacement après avoir substitué à `a` le texte de `expr1` et à `b` le texte de `expr2`. La présence de nombreuses parenthèses est chaudement recommandée car dans certains cas la priorité des opérations effectuées dans la macro peut être supérieure à celle des opérations faites dans ses arguments. Par exemple, supposons que l'on définisse une macro `PRODUIT` ainsi :
`#define PRODUIT(a,b) a*b`
Alors une instance notée `PRODUIT(2+3, 7);` donnera $2+3*7$ c'est à dire 23, ce qui n'est peut-être pas l'effet recherché...

D'autres fonctionnalités de remplacement plus fines sont possibles, mais sortent du cadre de cet aide-mémoire.

Le préprocesseur tient également à jour des macros spéciales qui contiennent la date, l'heure, le numéro de la ligne en cours et le nom de fichier en cours : `__DATE__`, `__TIME__`, `__LINE__` et `__FILE__`, qui sont utiles dans le cas d'affichage de messages d'erreur. Notons que les macros `__DATE__` et `__TIME__` sont remplacées par les valeurs correspondant à la date et l'heure de compilation et non celles d'une éventuelle exécution du programme !

Compilation conditionnelle

On souhaite parfois compiler ou non certaines parties d'un programme, par exemple s'il doit fonctionner sur des machines utilisant des systèmes d'expoitation différents, mais aussi pour inclure ou non du code de débogage par exemple. Pour cela le préprocesseur permet d'inclure des parties de code en fonction de conditions. La syntaxe est :

```
#if <condition>
       <code C a compiler si la condition est vraie>
#endif
```

Lorsqu'une alternative est possible, on peut écrire :

```
#if <condition>
       <code C a compiler si la condition est vraie>
#else
       <code C a compiler si la condition est fausse>
#endif
```

Dans le cas d'alternatives multiples, on a même :

```
#if <condition 1>
       <code C a compiler si la condition 1 est vraie>
#elif  <condition 2>
       <code C a compiler si la condition 1 est fausse et 2 vraie>
#else
       <code C a compiler si les deux conditions sont fausses>
#endif
```

Les conditions sont des expressions C booléennes ne faisant intervenir que des identificateurs du préprocesseur. Les expressions de la forme `defined(<identificateur>)` sont remplacées par 1 si l'identificateur est connu du préprocesseur et par 0 sinon. Comme il est fréquent d'écrire `#if defined(<id>)` ou `#if !defined(<id>)`, des formes abrégées existent et se notent respectivement `#ifdef <id>` et `#ifndef <id>`.

Citons deux autres utilisations importantes des remplacements par des macros, qui sont la compilation avec du code de débogage (*debug*) optionnel, et la gestion des variables globales. Le petit morceau de code suivant effectue un appel à `printf()` pour afficher un message, mais uniquement si le programme a été compilé en définissant DEBUG. Ceci peut avoir été fait dans un des fichiers d'en-têtes (.h) inclus systématiquement, ou bien par l'ajout de l'option `-DDEBUG` pour `gcc` sur la ligne de compilation.

Listing 26.14 – *Compilation conditionnelle*

```
#include <stdio.h>

#ifdef DEBUG
#define DEB(_x_) _x_
#else
#define DEB(_x_)
#endif

main() {
  DEB(printf("Ceci est un message de debug\n"));
}
```

La macro `DEB` est ici remplacée, soit par ce qui est entre parenthèses si DEBUG est défini, soit par rien dans le cas contraire. C'est plus court et plus lisible que :

```
main() {
#ifdef DEBUG
  printf("Ceci est un message de debug\n");
#endif
}
```

L'autre utilisation importante concerne les variables globales. En effet, celles-ci doivent être déclarées dans UN SEUL fichier source (si possible celui qui contient la fonction `main()`), et doivent avoir une déclaration précédée de `extern` dans tous les autres sources. Ceci se fait simplement en déclarant cette variable avec une macro EXT dans un fichier d'en-têtes (.h) inclus dans tous les sources. La macro est ensuite définie comme valant rien du tout si l'inclusion est faite dans le fichier contenant le `main()`, et comme `extern` partout ailleurs. Voyons un petit exemple :

Ici un fichier *general.h*, inclus dans TOUS les sources :

```
/* EXT sera remplace par extern partout sauf si MAIN est defini */
#ifdef MAIN
#  define EXT
#else
#  define EXT extern
#endif

EXT int monglob1;
EXT char *glob2;
```

Voici le début de *calcul.c* :

```
#include "general.h"

int trescomplique(int a) {
  return a+monglob1;
}
```

Et le début de *main.c* :

```
#define MAIN
#include "general.h"
#include <stdio.h >
```

```
main() {
  monglob1=3;
  printf("Resultat: %d\n",trescomplique(2));
}
```

26.7 Fonctions d'entrée/sortie

De nombreuses fonctions d'entrée/sortie existent dans la bibliothèque `stdio.h`. Seules certaines sont décrites dans cette partie, le reste est présenté dans la section 26.8. Une description plus détaillée de ces fonctions se trouve à la fin du manuel de référence de Kernighan et Ritchie, ou dans le `man` en ligne sous Unix.

Le passage d'arguments en ligne de commande

La fonction `main()` d'un programme peut en fait prendre des arguments. La déclaration complète est (les types des arguments sont imposés) :

```
int main(int argc, char **argv)
```

On écrit aussi fréquemment : `int main(int argc, char *argv[])` qui a le mérite de rappeler que `argv` est un tableau. Ces deux paramètres correspondent au nombre d'arguments de la ligne de commande *en comptant le nom de l'exécutable* (`argc`) et à la liste de ces arguments sous forme de chaînes de caractères (`argv[1]`, `argv[2]`, ..., `argv[argc-1]`). La première chaîne `argv[0]` contient le nom d'appel du programme.

Il est ainsi possible de passer des arguments en ligne de commande à un programme. Ces arguments peuvent être convertis avec les fonctions de conversion de la bibliothèque standard (`#include <stdlib.h>` : cf. Annexe) :

```
double atof(char *);
int atoi(char *);
long atol(char *);
```

Ou grâce aux fonctions d'entrée mise en forme que nous allons voir maintenant.

Entrée/sortie simples

Tout ce qui suit est subordonné à l'inclusion du fichier d'en-tête `stdio.h` qui déclare les fonctions d'entrée-sortie ainsi que la structure `FILE`. Les deux sections qui suivent donnent un aperçu des fonctions d'entrée-sortie sans les détailler. On se reportera à l'annexe qui suit pour de plus amples précisions.

Les fonctions d'entrée-sortie opèrent sur des flots (de type `FILE`), qui doivent avoir été déclarés au préalable. L'entrée standard, la sortie standard et la sortie d'erreur sont

automatiquement ouverts et se nomment respectivement `stdin`, `stdout` et `stderr`[24]. Par défaut l'entrée standard est le clavier, et les sorties sont à l'écran. Les opérateurs de redirection du *shell* permettent néanmoins d'altérer ceci en redirigeant ces accès vers/depuis un fichier/autre programme.

Pour des flots associés à des fichiers, il faut au préalable les ouvrir avec la fonction `fopen()`. Une bonne règle à respecter est que tout flot ouvert par un programme doit être fermé par `fclose()` dès qu'il ne sert plus.

Les flots sont des abstractions qui permettent de ne pas avoir à repréciser à chaque utilisation d'un fichier tous les paramètres : une fois le fichier ouvert, toutes les opérations ultérieures font référence à ce fichier par le flot, ce qui est beaucoup plus rapide car la bibliothèque standard (et aussi le système d'exploitation) retient l'état courant du fichier (position dans le fichier, emplacement sur le disque...). Cette abstraction permet également de travailler sur d'autres objets séquentiels en utilisant les mêmes fonctions, c'est le cas par exemple pour les `tuyaux` entre processus, pour les *sockets* réseau...

Les fonctions `fread()` et `fwrite()` permettent de lire des blocs de données dans un flot sans aucune interprétation (cf. section 26.8). Si l'on souhaite plutôt lire un fichier ligne par ligne, il est préférable d'utiliser `fgets()` qui offre l'avantage de s'arrêter aux fins de ligne (caractère `'\n'`), et garantit tout de même que la longueur de la chaîne de stockage ne peut pas être dépassée.

Dans les cas où le but est de traiter les caractères un à un sans attendre, il faut utiliser `getc()` et `putc()` (ou leurs équivalents `getchar()` et `putchar()` pour les flots d'entrée-sortie standards). La fonction `ungetc()` qui permet d'implanter simplement un *look-ahead*[25] de façon simple peut également rendre service.

24. En fait, cette propriété n'est assurée que pour un système d'exploitation Unix, même si la plupart des compilateurs `C` fonctionnant dans d'autres environnements tentent de simuler ce comportement.

25. Littéralement *regarder en avant*, c'est une méthode de décision du type de lexème qui suit à partir de la lecture du premier caractère uniquement.

Les fonctions `fopen()`, `fread()`, `fprintf()`, `fflush()`... sont des fonctions présentes dans la bibliothèque standard, et sont donc normalement disponibles sur tous les systèmes d'exploitation. Ces fonctions sont écrites en `C` et précompilées dans un fichier d'archive qui forme la bibliothèque standard, et qui est ajouté automatiquement lors de la compilation. Écrire `fopen()` est en fait un simple appel de fonction vers cette bibliothèque, dont le code exécutable fait partie du programme. À son tour, ces fonctions (`fopen()` par exemple) utilisent des services fournis par le système d'exploitation pour effectuer leur tâche. Ces services sont sollicités via un ou plusieurs appels système qui demandent au noyau du système d'exploitation d'ouvrir le fichier voulu. Sous Unix, l'appel système (dont l'utilisation est similaire à un appel de fonction) utilisé ainsi est `open()`, qui retourne un entier appelé descripteur de fichier Unix. Cette valeur de retour est stockée dans l'objet `FILE` afin d'en disposer pour les manipulations ultérieures. Quelques autres appels système concernant les fichiers sous Unix sont `read()`, `write()`, `close()`...

Il est fortement recommandé de ne pas mélanger pour un même flot des appels aux fonctions de la bibliothèque standard et des appels système Unix, car il n'y aurait dans ce cas aucune garantie que l'ordre des opérations soit respecté (à cause des tampons en mémoire maintenus par la bibliothèque standard).

Entrées/sorties avec format : `fprintf(), fscanf()`

Pour obtenir des résultats joliment mis en forme, la commande à utiliser est `fprintf( flot, format, variables...)`. Lorsque `flot` est la sortie standard, un raccourci existe : `printf( format, variables...)`. Le `format` est une chaîne de caractères qui décrit la façon d'afficher les variables qui suivent ; par exemple `printf("x=%d y=%f\n",x,y);` affiche `x=` suivi de la valeur de la variable entière `x`, puis une espace puis `y=` suivi de la valeur de la variable flottante `y` et un retour chariot. La fonction `printf` est une fonction particulière qui prend un nombre variable d'arguments. Les codes à utiliser dans le format indiquent le type de la variable et sont présentés en annexe. Ainsi pour afficher un nombre complexe, on écrira par exemple `printf("z=%f+i%f\n",Re(z),Im(z));`.

La valeur de retour de `fprintf()` est **le nombre de caractères** effectivement écrits, ou une valeur négative en cas d'erreur.

La lecture de valeur se fait par la fonction `fscanf(flot, format, adresses des variables...)`. Le principe est similaire à `fprintf()`, excepté que les adresses des variables doivent être données afin que `fscanf()` puisse aller modifier leur valeur. Lorsque `flot` est l'entrée standard, on peut utiliser `scanf(format, adresses des variables...)`. Ainsi pour lire en entrée une variable flottante `float x;` suivie d'une chaîne de caractères `char cha[1024];`, on écrira `scanf("%f%s",&x, cha);` La variable `x` est passée par adresse, et la chaîne aussi puisqu'il s'agit d'un pointeur, donc d'une adresse mémoire.

La fonction `fscanf()` retourne **le nombre de champs** qu'elle a pu lire et convertir (éventuellement zéro), ou la constant speciale `EOF` (qui vaut généralement -1) si une

erreur de lecture ou la fin du fichier se produit avant la premiere conversion.

> Les caractères autres que des caractères de formatage et les espaces présents dans le format de `scanf()` doivent correspondre **exactement** aux caractères entrés au clavier. Il faut en particulier éviter de mettre un espace ou un `\n` dans ce format. Les caractères d'espacement sont considérés par `fscanf()` comme des séparateurs de champs. Il n'est donc pas possible de récupérer d'un coup une chaîne de caractères comportant des blancs avec cette fonction. C'est pourquoi on utilisera plutôt `char *fgets(char *s, int n, FILE *f)` qui lit une ligne jusqu'au retour chariot ou jusqu'à ce qu'il y ait `n` caractères et stocke le résultat dans `s` (elle retourne `s` également s'il n'y a pas eu d'erreur).

Il ne faut pas perdre de vue que le système de fichiers d'Unix est bufferisé. Ainsi pour l'entrée standard, celle-ci n'est prise en compte que ligne par ligne après un retour chariot [26]. De même pour la sortie standard [27]. Lorsqu'on écrit dans un fichier, ou si la sortie standard est redirigée, la bufferisation est encore plus importante puisque la taille du tampon peut atteindre plusieurs kilooctets. Pour forcer l'écriture, on utilise `fflush()`. Cette fonction est utile, car si un programme n'arrive pas à son terme (arrêt intempestif de la machine, ou erreur à l'exécution par exemple) les buffers sont irrémédiablement perdus.

Enfin la fonction `fclose()` permet de fermer un fichier, en vidant auparavant son buffer.

26.8 Compléments : la bibliothèque standard et quelques fonctions annexes

La bibliothèque standard ne fait pas partie à proprement parler du langage `C`, mais elle a été standardisée par la norme ANSI et est devenue une norme ISO. Elle offre donc un ensemble de déclarations de fonctions, de déclarations de types et de macros qui sont communes à tous les compilateurs qui respectent la norme, quel que soit le système d'exploitation. Quelques exemples présentés ici sont néanmoins des extensions de cette bibliothèque standard spécifiques au monde Unix. De nombreuses fonctions de la bibliothèque standard reposent sur des appels systèmes Unix qui leur ressemblent, comme `open()`, `close()`, `read()` ou `lseek()` par exemple. Si la portabilité vers des environnements non-Unix n'est pas de mise, il pourra être intéressant dans certains cas d'utiliser plutôt ces dernières fonctions.

Toutes les déclarations de la bibliothèque standard figurent dans des fichiers d'entête :

26. Il est possible, mais ceci est spécifique Unix, de reparamétrer le terminal afin de recevoir les caractères un à un sans attendre un retour chariot.

27. Là un `fflush()` permet de forcer l'écriture n'importe quand. Il reste néanmoins préférable d'utiliser `stderr` pour les sorties d'erreurs non-bufferisées.

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

Ces fichiers doivent être inclus avant toute utilisation d'une fonction de la bibliothèque par une directive du type `#include <fichier_entete.h>`. Ils peuvent être inclus dans un ordre quelconque, voire plusieurs fois dans un même source. Ils contiennent des déclarations de fonctions que le compilateur connaît par défaut pour la plupart, excepté pour <math.h> qui déclare des fonctions de la bibliothèque mathématique qu'il faut indiquer à l'édition de lien de façon explicite (voir 26.2).

Les identificateurs masqués à l'utilisateur à l'intérieur de cette bibliothèque ont des noms commençant par un caractère de soulignement _. Il est donc recommandé d'éviter d'utiliser ce type de dénomination pour éviter d'éventuels conflits.

Seules les fonctions principales sont présentées ici.

Pour des informations plus détaillées, on se reportera aux pages de manuel en ligne sous Unix, accessibles par `man fgets` par exemple pour la fonction `fgets()`. Mais une fonction comme `open()` est également le nom d'une commande Unix, aussi `man open` ne retournera pas la page voulue. Dans ce cas, on précisera la section du manuel correspondante, ici la section 2 (pour les appels système) ou 3 (pour les fonctions de la bibliothèque). Ceci s'écrit `man 2 open` ou `man 3 fprintf`.

<stdio.h> : entrées/sorties

Il s'agit de la partie la plus importante de la bibliothèque. Sans cela un programme ne peut afficher ou lire des données de façon portable. La source ou la destination des données est décrite par un objet de type FILE, appelé *flot*. Ce type structuré, dont il n'est pas utile de connaître les différents champs (ils peuvent d'ailleurs différer entre les versions de la bibliothèque) est associé à un fichier sur disque ou à un autre périphérique comme le clavier ou l'écran. Il existe des flots binaires et des flots de texte, mais sous Unix ceux-ci sont identiques [28].

Un *flot* est associé à un fichier par ouverture, ceci retourne un pointeur sur un type FILE. À la fin des opérations sur le flot, il faut le fermer.

Au début de l'exécution du programme, trois flots spéciaux, nommés `stdin` pour l'entrée standard, `stdout` pour la sortie standard et `stderr` pour la sortie d'erreur sont déjà ouverts. En l'absence d'opérations spéciales, `stdin` correspond au clavier, `stdout` et `stderr` correspondent à l'écran.

Ouverture/fermeture de fichiers

`FILE *fopen(const char *nomfich, const char *mode)` Ouvre le fichier indiqué (le nom est absolu ou relatif au répertoire courant, mais sans remplacement des carac-

28. Sous DOS, les flots texte convertissent les séquences CR-LF en LF seul (codes \r\n en \n seul), sous Mac OS X, ils convertissent le CR en LF (code \r en \n), sous Unix, ils convertissent les LF en LF !

tères spéciaux comme ~, ? ou ∗) et retourne un flot, ou `NULL` en cas d'erreur. L'argument *mode* est une chaîne de caractères parmi :

`r`	lecture seule, la lecture commence au début du fichier.
`w`	écriture seule. Si le fichier n'existait pas, il est créé ; s'il existait, il est écrasé.
`a`	(*append*) similaire à `w` mais écrit à la fin du fichier sans l'écraser s'il existait.
`r+`	lecture et écriture. Le flot est positionné au début du fichier.
`w+`	le fichier est crée (ou écrasé s'il existait), et ouvert en lecture et écriture ; le flot est positionné au début.
`a+`	similaire à `r+`, mais le flot est initialement positionné à la fin du fichier.

Dans le cas des modes lecture/écriture, il faut appeler `fflush()` ou une fonction de déplacement entre les opérations de lecture et d'écriture. L'ajout de `b` à la fin du mode indique un fichier binaire, c'est-à-dire sans conversion automatique des fins de ligne (cela n'a aucun effet sous Unix, voir la note 28).

`FILE *freopen(const char *nomfich, const char *mode, FILE *flot)` est une fonction qui sert à associer aux flots prédéfinis `stdin`, `stdout` ou `stderr` un fichier.

`int fflush(FILE *flot)` Cette fonction n'est à appeler que sur un flot de sortie (ou d'entrée-sortie). Elle provoque l'écriture des données mises en mémoire tampon. Elle retourne zéro en l'absence d'erreur, `EOF` sinon.

`int fclose(FILE *flot)` Provoque l'écriture des données en attente, puis ferme le flot. La valeur de retour est zéro en l'absence d'erreur, `EOF` sinon.

`FILE *fdopen(int fd, const char *mode)` Ouvre un flot correspondant au descripteur de fichier Unix `fd`. Nécessite un descripteur de fichier valide (par exemple issu de `open()`). Le descripteur n'est pas dupliqué, aussi il faut au choix soit terminer par `close(fd)` [29], soit par `fclose(flot)`, mais surtout pas les deux. Ne fait pas partie de la norme ANSI.

`int fileno( FILE *flot)` Retourne l'entier descripteur de fichier Unix associé au flot. Ne fait pas partie de la norme ANSI.

Positionnement et erreurs

`int fseek(FILE *flot, long decalage, int origine)` Positionne la *tête de lecture* dans le flot `flot`. L'entier `origine` peut valoir `SEEK_SET`, `SEEK_CUR` ou `SEEK_END` selon que la position est donnée respectivement à partir du début du fichier, de la position courante, ou de la fin. `decalage` représente le décalage demandé en nombre d'octets (attention au signe, une valeur négative faisant revenir en arrière).

`fseek()` retourne une valeur non nulle en cas d'erreur, zéro en cas de succès.

29. Dans ce cas, les tampons de mémoire utilisés par les fonctions de la bibliothèque d'entrées / sorties standard ne seront pas vidés et il risque d'y avoir perte de données.

`long ftell(FILE *flot)` Donne la valeur de la position courante dans le fichier (en nombre d'octets depuis le début du fichier), ou la valeur `-1L` (-1 sous forme de `long`) en cas d'erreur.

`void rewind(FILE *flot)` Retourne au début du fichier. On pourrait réaliser cette action en utilisant `(void)fseek(flot, 0L, SEEK_SET)`.

`int feof(FILE *flot)` Retourne une valeur non nulle si l'indicateur de fin de fichier est actif, c'est-à-dire si la dernière opération sur le flot a atteint la fin du fichier.

`int ferror(FILE *flot)` Retourne une valeur non nulle si l'indicateur d'erreur du flot est actif.

Toutes les fonctions qui prennent en argument un flot (de type `FILE *`) nécessitent que ce flot soit valide. En particulier, il ne faut pas faire `f=fopen(...)` puis appeler immédiatement `ferror(f)`, car `f` peut valoir `NULL` si l'ouverture du fichier a échoué. la solution est alors :

```
FILE *f;

f=fopen("mon_fichier","r");
if (f==NULL) {
   perror("Impossible d'ouvrir le fichier\n");
   exit(EXIT_FAILURE);
}
```

`void clearerr(FILE *flot)` Remet à zéro les indicateurs de fin de fichier et d'erreur. C'est une fonction qui ne sert quasiment jamais.

`void perror(const char *s)` Affiche sur `stderr` la chaîne `s` et un message d'erreur correspondant à l'entier contenu dans la variable globale `errno`, donc issu de la dernière erreur rencontrée.

`perror(s)` est équivalent à :

```
(void)fprintf(stderr, "%s : %s\n", strerror(errno) , s)
```

`int fgetpos(FILE flot, fpos_t *pos)`

`int fsetpos(FILE *flot, fpos_t *pos)` Ce sont d'autres fonctions permettant de relire la position dans le fichier (comme `ftell(flot)`) ou de l'imposer (comme `fseek(flot,SEEK_SET)`). Cette position est stockée dans un objet de type `fpos_t` (sous Unix, c'est un `long`, mais ce peut être une structure complexe sur d'autres systèmes d'exploitation).

Ces deux fonctions ne peuvent guère servir qu'à lire la position dans un fichier pour y revenir plus tard.

Il arrive parfois que l'on souhaite raccourcir un fichier déjà ouvert. Cela n'est pas possible avec les fonctions de la bibliothèque standard mais peut se faire avec l'appel `ftruncate()` (cf manuel en ligne).

Entrée/sortie directe

`size_t` est un type (correspondant à un entier) qui exprime une taille mémoire (en octet, issue par exemple de `sizeof()`, comme pour `malloc()`) ou un nombre d'éléments.

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *flot)`
Lit sur le flot `nobj` objets de taille `size` et les écrit à l'emplacement mémoire pointé par `ptr`. La valeur de retour est le nombre d'objets lus avec succès, qui peut être inférieur à `nobj`. Pour déterminer s'il y a eu erreur ou si la fin du fichier a été atteinte, il faut utiliser `feof()` ou `ferror()`.

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *flot)`
Écrit sur le flot `nobj` objets de taille `size` pris à l'emplacement pointé par `ptr`. Retourne le nombre d'objets écrits.

Entrée/sortie de caractères

`int fgetc(FILE *flot)` Lit un caractère sur le flot (`unsigned char` converti en `int`) et le retourne. Retourne `EOF` en cas de fin de fichier ou d'erreur (`EOF` est une constante entière définie par `stdio.h`, valant généralement -1).

`int fputc(int c, FILE *flot)` Écrit le caractère `c` (converti en `unsigned char`) sur le flot. Retourne ce caractère ou `EOF` en cas d'erreur.

`int getc(FILE *flot)` Équivalent à `fgetc()`, mais éventuellement implanté sous forme de macro. Plus efficace que `fgetc` en général. Attention, l'évaluation de `flot` ne doit provoquer d'effet de bord (utiliser une variable pour `flot` et non le retour d'une fonction).

`int putc(int c, FILE *flot)` Équivalent à `fputc()`. Même remarque que pour `getc()`.

`int getchar(void)` Équivalent à `getc(stdin)`.

`int putchar(int c)` Équivalent à `putc(c, stdout)`.

`int ungetc(int c, FILE *flot)` Remet le caractère `c` (converti en `unsigned char`) sur le flot, où il sera retrouvé à la prochaine lecture. Un seul caractère peut être ainsi remis, et cela ne peut pas être `EOF`. Retourne le caractère en question ou `EOF` en cas d'erreur.

`char *fgets(char *s, int n, FILE *flot)` Lit une ligne complète de caractères (jusqu'au caractère de fin de ligne compris) d'au plus $n-1$ caractères et place cette chaîne à partir de la case mémoire pointée par `s`. Si un caractère de fin de ligne est lu, il est stocké dans la chaîne. Le caractère `'\0'` est alors ajouté à la chaîne précédemment lue. Retourne `s` si tout s'est bien passé, ou bien `NULL` en cas d'erreur ou si la fin du fichier est atteinte sans aucun caractère disponible le précédant.

`int fputs(const char *s, FILE *flot)` Écrit la chaîne de caractêres `s` sur le flot (n'écrit pas le caractère `'\0'` de fin de chaîne). Retourne une valeur positive ou nulle, ou bien `EOF` en cas d'erreur.

Il existe une fonction `gets()` similaire à `fgets()`, mais son usage est fortement déconseillé. En effet cette fonction est équivalente à `fgets()` avec `n` infini ce qui ouvre grand les portes du débordement de mémoire.

Entrée/sortie mises en forme

`int fprintf(FILE *flot, const char *format, ...)` Est la commande de sortie mise en forme. Il s'agit d'une fonction à nombre variable d'arguments (Voir 26.8). La chaîne `format` décrit le texte à écrire, dans lequel des suites de caractères spéciaux commençant par `%` sont remplacées par les valeurs des arguments qui suivent dans l'ordre. La valeur entière de retour est *le nombre de caractères écrits*, ou en cas d'erreur un nombre négatif.

La chaîne de caractères `format` contient des caractères ordinaires (sans `%`) qui sont copiés tels quels sur le flot de sortie, et des directives de conversion dont chacune peut provoquer la conversion dans l'ordre d'un argument de `fprintf()` suivant la chaîne de format.

Chaque directive de conversion commence par le caractère `%` et se termine par un caractère de conversion parmi ceux présentés dans le tableau qui suit. Entre le `%` et le caractère de conversion peuvent apparaître dans l'ordre :

- Des drapeaux, parmi :
  - `#` format de sortie alternatif.
  - `–` cadrage à gauche du champ.
  - `+` ajout systématique du signe aux nombres.
  - `' '` (espace) ajout d'un caractère pour le signe des nombres, soit l'espace pour les valeurs positives ou le signe `'-'` pour les valeurs négatives.
  - `0` (zéro) complète le début du champ par des zéros au lieu d'espaces pour les nombres.
- Un nombre précisant la largeur minimale du champ d'impression en nombre de caractères.
- Une précision sous la forme d'un point suivi d'un nombre (la valeur de cette précision est considérée comme zéro en l'absence de ce champ). Il s'agit du nombre maximal de caractères à imprimer pour une chaîne de caractères, ou du nombre minimal de chiffres pour les conversions `d`, `i`, `o`, `u`, `x` et `X`. Pour les conversions `e`, `E` et `f` cela donne le nombre de chiffres après le point décimal, et sinon cela correspond au nombre maximal de chiffres significatifs pour les conversions `g` et `G`.
- Enfin une lettre `h`, `l` ou `L` qui modifie le type de l'argument converti. `h` spécifie un argument `short`, `l` spécifie un argument `long`, et `L` un argument `long double` (ou `long long` pour les conversions entières, bien que ceci ne fasse pas partie de la norme ANSI).

Voici enfin les caractères de conversion :

`d, i, o, u, x, X`	Variante d'`int`. `d` : décimal. `u` : décimal non signé. `o` : octal. `x` : hexadécimal non signé utilisant `abcdef`. `X` : hexadécimal non signé utilisant `ABCDEF`.
`e, E`	`double` écrit en décimal sous la forme `[-]m.dddddd`Edd. La précision donne le nombre de chiffre après le point de la mantisse, la valeur par défaut est 6. `e` utilise la lettre `e` et `E` utilise `E`.
`f`	`double` écrit sous la forme `[-]ddd.ddd`. La précision donne le nombre de chiffre après le point décimal, sa valeur par défaut est 6. Si la précision vaut 0, le point décimal est supprimé.
`g, G`	`double` écrit en décimal comme par une conversion par `f` ou `e` selon la valeur de l'argument.
`c`	`int` converti en `unsigned char` et affiché comme un caractère ASCII.
`s`	`char *`. Chaîne de caractères.
`p`	`void *`. Pointeur écrit en hexadécimal (comme avec `%#lx`).
`n`	`int *`. Ne convertit pas d'argument, mais écrit dans l'argument le nombre de caractères affichés jusqu'à présent.
`%`	Affiche simplement un `%` sans aucune conversion.

Chaque directive de conversion doit correspondre exactement à un argument de type correct dans le même ordre.

> Les erreurs sur les types des arguments ou leur nombre ne sont généralement pas détectés par le compilateur et peuvent provoquer des résultats surprenants à l'exécution.

`int fscanf(FILE *flot, const char *format, ...)` Est la commande d'entrée mise en forme. Il s'agit aussi d'une fonction à nombre variable d'arguments. La chaîne `format` décrit les objets à lire et leur type, et ceux-ci sont stockés dans les arguments qui suivent *qui doivent être obligatoirement des pointeurs*. La valeur entière de retour est le nombre d'éléments d'entrée correctement assignés (compris entre 0 et le nombre de directives de conversion). Si la fin de fichier est atteinte avant toute lecture, la valeur EOF est renvoyée. Si la fin de fichier est atteinte en cours de lecture, la fonction retourne le nombre d'éléments lus sans erreur.

La chaîne de caractères `format` contient des caractères ordinaires (sans `%`) qui doivent correspondre exactement aux caractères lus sur le flot d'entrée (exceptés les espaces et les tabulations qui sont ignorés). Elle contient également des directives de conversion formées dans l'ordre par :

– `%`,

– un caractère facultatif `'*'`, qui, lorsqu'il est présent, empêche l'affectation de la conversion à un argument (le champ est lu et sa valeur ignorée, aucun pointeur n'est utilisé dans les arguments de la fonction),
– un caractère facultatif `h`, `l` ou `L` qui spécifie une variante de taille de type (voir ci-dessous),
– un caractère de conversion parmi ceux présentés dans le tableau qui suit :

`d`	variante d'`int *`, lu sous forme décimale
`i`	variante d'`int *`, la lecture est en octal si l'entier commence par 0, en hexadécimal s'il commence par 0x ou 0X, en décimal sinon
`o`	variante d'`int *`, lu sous forme octale, qu'il commence ou non par un 0
`u`	Variante d'`unsigned int *`, lu sous forme décimale
`x`	variante d'`unsigned int *`, lu sous forme hexadécimale.
`f, e, g`	`float *` (ou `double *`). Les trois caractères sont équivalents
`c`	`char *`, lit les caractères suivants et les écrit à partir de l'adresse indiquée jusqu'à la largeur du champ si celle-ci est précisée. En l'absence de largeur de champ, lit 1 caractère. N'ajoute pas de `'\0'`. Les caractères d'espacement sont lus comme des caractères normaux et ne sont pas sautés
`s`	`char *`, chaîne de caractères lue jusqu'au prochain caractère d'espacement, en s'arrêtant éventuellement à la largeur du champ si elle est précisée. Ajoute un `'\0'` à la fin
`p`	`void *`, pointeur écrit en hexadécimal
`n`	`int *`, ne convertit pas de caractères, mais écrit dans l'argument le nombre de caractères lus jusqu'à présent. Ne compte pas dans le nombre d'objets convertis retourné par `fscanf()`
`[...]`	`char *`, la plus longue chaîne de caractère non vide composée exclusivement de caractères figurant entre les crochets. Un `'\0'` est ajouté à la fin. Pour inclure] dans cet ensemble, il faut le mettre en premier `[]...]`
`[^...]`	`char *`, la plus longue chaîne de caractère non vide composée exclusivement de caractères NE figurant PAS entre les crochets. Un `'\0'` est ajouté à la fin. Pour inclure] dans cet ensemble, il faut le mettre en premier `[^]...]`
`%`	lit le caractère `%`. Ne réalise aucune affectation

Les variantes de taille de type sont plus complètes que celles pour `fprintf()` : `h` précise un `short` et `l` un `long` pour les conversions `d, i, n, o, u, x`. `l` précise un `double` et `L` un `long double` pour les conversions `e, f, g`. Ainsi pour lire un `double` on utilise `scanf("%lg",&d)` mais pour l'imprimer on utilise `printf("%g",d)`.

Un champ en entrée est défini comme une chaîne de caractères différents des caractères d'espacement ; il s'étend soit jusqu'au caractère d'espacement suivant, soit

jusqu'à ce que la largeur du champ soit atteinte si elle est précisée. En particulier, `scanf()` peut lire au-delà des limites de lignes si nécessaire, le caractère de fin de ligne étant un caractère comme les autres.

Chaque directive de conversion doit correspondre exactement à un pointeur vers un argument de type correct dans le même ordre.

> Même remarque que pour `fprintf()`. De plus, si les pointeurs de destination n'ont pas le bon type (ou pire encore s'il ne sont même pas des pointeurs), le résultat à l'exécution risque d'être déroutant...

`int printf(const char *format, ...)` est équivalent à :

```
fprintf(stdout, format, ...)
```

`int scanf(const char *format, ...)` est équivalent à :

```
fscanf(stdin, format, ...)
```

`int sprintf(char *s, const char *format, ...)` est similaire à `fprintf()` excepté que l'écriture se fait dans la chaîne de caractères `s` plutôt que dans un flot. Attention, la taille de la chaîne `s` doit être suffisante pour stocker le résultat (y compris le dernier caractère `'\0'` invisible de fin de chaîne). Dans le cas contraire, on risque d'avoir des erreurs surprenantes à l'exécution.

`int sscanf(char *s, const char *format, ...)` Est similaire à `fscanf()` excepté que la lecture se fait dans la chaîne de caractères `s` plutôt que dans un flot.

<ctype.h> : tests de caractères

Ces fonctions retournent une valeur non nulle si l'argument remplit la condition, et zéro sinon. Toutes ces fonctions ont comme prototype `int isalnum(int c)`, et `c`, bien qu'étant un entier doit contenir une valeur représentable comme un `unsigned char`.

`isalnum()`	isalpha() ou isdigit() est vrai.
`isalpha()`	isupper() ou islower() est vrai.
`iscntrl()`	caractère de contrôle.
`isgraph()`	caractère imprimable sauf l'espace.
`islower()`	lettre minuscule.
`isprint()`	caractère imprimable y compris l'espace.
`ispunct()`	caractère imprimable différent de l'espace, des lettres et chiffres.
`isspace()`	espace, saut de page, de ligne, retour chariot, tabulation.
`isupper()`	lettre majuscule.
`isxdigit()`	chiffre hexadécimal.

<string.h> : chaînes de caractères et blocs mémoire

Chaînes de caractères

`char *strcpy(char *s, const char *ct)` Copie la chaîne `ct`, y compris le caractère `'\0'` dans le chaîne `s`, retourne `s`. À éviter si les tailles de chaînes ne sont pas connues et qu'il risque d'y avoir débordement.

`char *strncpy(char *s, const char *ct, size_t n)` Copie au plus `n` caractères de `ct` dans `s`, en complétant par des `'\0'` si `ct` comporte moins de `n` caractères.

`char *strcat(char *s, const char *ct)` Concatène `ct` à la suite de `s`, retourne `s`. À éviter si les tailles de chaînes ne sont pas connues et qu'il risque d'y avoir débordement.

`char *strncat(char *s, const char *ct, size_t n)` Concatène au plus `n` caractères de `ct` à la suite de `s`, termine la chaîne résultat par `'\0'`, retourne `s`.

`char *strcmp(const char *cs, const char *ct)` Compare lexicographiquement `cs` et `ct`. Retourne une valeur négative si `cs<ct`, nulle si `cs==ct`, positive sinon.

`char *strncmp(char *s, const char *ct, size_t n)` Comme `strcmp()` mais limite la comparaison aux `n` premiers caractères.

`char *strchr(char *s, int c)` Retourne un pointeur sur la première occurrence de `c` (converti en `char`) dans `cs`, ou `NULL` si il n'y en a aucune.

`char *strrchr(char *s, int c)` Comme `strchr` mais pour la dernière occurrence.

`char *strstr(const char *cs,const char *ct)` Retourne un pointeur sur la première occurrence de la chaîne `ct` dans `cs`, ou `NULL` si il n'y en a aucune.

`size_t strlen(const char *cs)` Retourne la longueur de `cs`, sans compter le caractère de terminaison `'\0'`.

`char *strerror(int n)` Retourne un pointeur sur la chaîne correspondant à l'erreur `n`.

`char *strtok(char *s, const char *ct)` Décomposition de `s` en lexèmes séparés par des caractères de `ct`.

Le premier appel à la fonction se fait en précisant dans `s` la chaîne de caractères à décomposer. Il retourne le premier lexème de `s` composé de caractères n'appartenant pas à `ct`, et remplace par `'\0'` le caractère qui suit ce lexème.

Chaque appel ultérieur à `strtok()` avec `NULL` comme premier argument retourne alors le lexème suivant de `s` selon le même principe (le pointeur sur la chaîne est mémorisé d'un appel au suivant). La fonction retourne `NULL` lorsqu'elle ne trouve plus de lexème. Cette fonction conservant un état dans une variable locale statique n'est pas réentrante. Une version `strtok_r()` stockant l'état dans une variable fournie par le programmeur existe a cet effet.

> Cette fonction, d'usage peu orthodoxe (utilisant des variables locales statiques...) est extrêmement efficace car elle est généralement écrite directement en assembleur. Ne pas croire la page de manuel Linux qui dit « never use this function » ! Il est quand même recommandé d'utiliser `strsep()` ou `strtok_r()`...

Blocs de mémoire

`void *memcpy(void *s, const void *ct, size_t n)` copie `n` octets de `ct` dans `s` et retourne `s`. Attention, les zones manipulées ne doivent pas se chevaucher.

`void *memmove(void *s, const void *ct, size_t n)` même rôle que la fonction `memcpy()`, mais fonctionne également pour des zones qui se chevauchent. `memmove()` est plus lente que `memcpy()` lorsque les zones ne se chevauchent pas.

`int memcmp(const void *cs, const void *ct, size_t n)` compare les zones de la mémoire pointées par `cs` et `ct` à concurrence de `n` octets. Les valeurs de retour suivent la même convention que pour `strcmp()`.

`void *memset(void *s, int c, size_t n)` écrit l'octet `c` (un entier convertit en `char`) dans les `n` premiers octets de la zone pointée par `s`. Retourne `s`.

<math.h> : fonctions mathématiques

La bibliothèque correspondant à ces fonctions n'est pas ajoutée automatiquement par le compilateur et doit donc être précisée lors de l'édition de liens par l'option `-lm`.

Lorsqu'une fonction devrait retourner une valeur supérieure au maximum représentable sous la forme d'un double, la valeur retournée est `HUGE_VAL` et la variable globale d'erreur `errno` reçoit la valeur `ERANGE`. Si les arguments sont en dehors du domaine de définition de la fonction, `errno` reçoit `EDOM` et la valeur de retour est non significative. Afin de pouvoir tester ces erreurs, il faut aussi inclure <errno.h>.

Toutes les fonctions trigonométriques prennent leur argument ou retournent leur résultat en radians. Les valeurs de retour sont toutes des `double`, ainsi que les arguments des fonctions du tableau suivant :

`sin()`	sinus.
`cos()`	cosinus.
`tan()`	tangente.
`asin()`	arc sinus.
`acos()`	arc cosinus.
`atan()`	arc tangente (dans $[-\pi/2, \pi/2]$).
`atan2(y,x)`	pseudo arc tangente de y/x, tenant compte des signes, dans $[-\pi, \pi]$.
`sinh()`	sinus hyperbolique.
`cosh()`	cosinus hyperbolique.
`tanh()`	tangente hyperbolique.
`exp()`	exponentielle.
`log()`	logarithme népérien.
`log10()`	logarithme à base 10.
`pow(x,y)`	x à la puissance y. Attention au domaine : ($x > 0$) ou ($x = 0$, $y > 0$) ou ($x < 0$ et y entier).
`sqrt()`	racine carrée.
`floor()`	Partie entière exprimée en `double`.
`ceil()`	Partie entière plus 1 exprimée en `double`.
`fabs()`	Valeur absolue.

Pour obtenir la partie entière d'un `float f` sous la forme d'un entier, on utilise le transtypage en écrivant par exemple : `(int)floor( (double)f )`.

<stdlib.h> : fonctions utilitaires diverses

`double atof(const char *s)` Convertit `s` en un `double`.

`int atoi(const char *s)` Convertit `s` en un `int`.

`long atol(const char *s)` Convertit `s` en un `long`.

`void srand(unsigned int graine)` Donne `graine` comme nouvelle amorce du générateur pseudo-aléatoire. L'amorce par défaut vaut 1.

`int rand(void)` Retourne un entier pseudo-aléatoire compris entre 0 et `RAND_MAX`, qui vaut au minimum 32767.

`void *calloc(size_t nobj, size_t taille)` Allocation d'une zone mémoire correspondant à un tableau de `nobj` de taille `taille`. Retourne un pointeur sur la zone ou `NULL` en cas d'erreur. La mémoire allouée est initialisée par des zéros.

`void *malloc(size_t taille)` Allocation d'une zone mémoire de taille `taille` (issu de `sizeof()` par exemple). Retourne un pointeur sur la zone ou `NULL` en cas d'erreur. La mémoire allouée n'est pas initialisée.

`void *realloc(void *p, size_t taille)` Change en `taille` la taille de l'objet pointé par `p`. Si la nouvelle taille est plus petite que l'ancienne, seul le début de la zone est conservé. Si la nouvelle taille est plus grande, le contenu de l'objet est conservé

et la zone supplémentaire n'est pas initialisée. En cas d'échec, `realloc()` retourne un pointeur sur le nouvel espace mémoire, qui peut être différent de l'ancien, ou bien `NULL` en cas d'erreur et dans ce cas le contenu pointé par `p` n'est pas modifié.

Si `p` vaut `NULL` lors de l'appel, le fonctionnement est équivalent à `malloc(taille)`.

`void free(void *p)` Libère l'espace mémoire pointé par `p`. Ne fait rien si `p` vaut `NULL` . `p` doit avoir été alloué par `calloc()`, `malloc()` ou `realloc()`.

`void exit(int status)` Provoque l'arrêt du programme. Les fonctions `atexit()` sont appelées dans l'ordre inverse de leur enregistrement, les flot restant ouverts sont fermés. La valeur de `status` est retourné à l'appelant [30]. Une valeur nulle de `status` indique que le programme s'est terminé avec succès. On peut utiliser les valeurs `EXIT_SUCCESS` et `EXIT_FAILURE` pour indiquer la réussite ou l'échec.

`void abort(void)` Provoque un arrêt anormal du programme.

`int atexit(void (*fcn)(void))` Enregistre que la fonction `fcn()` devra être appelée lors de l'arrêt du programme. Retourne une valeur non nulle en cas d'erreur.

`int system(const char *s)` Sous Unix provoque l'évaluation de la chaîne `s` dans un Bourne-shell (`sh`). La valeur de retour est la valeur de retour de la commande, ou bien 127 ou -1 en cas d'erreur. L'appel `system(NULL)` retourne zéro si la commande n'est pas utilisable.

Cette commande, parfois utile en phase de prototype, est à proscrire dans un projet finalisé : elle peut bloquer les interruptions du programme et, passant des variables d'environnement à `/bin/sh`, conduit fréquemment à des erreurs ou des problèmes de sécurité. On lui préfèrera une solution moins intégrée fondée sur `fork()` et `exec()`, plus efficace et plus sure.

Comme pour `wait()` la valeur de retour est encodée et il faudra donc utiliser `WEXITSTATUS()` pour obtenir la vraie valeur status.

`char *getenv(const char *nom)` Récupère la déclaration de la variable d'environnement dont le nom est `nom`. La valeur de retour est un pointeur vers une chaîne du type `nom = valeur`.

`void qsort(void *base, size_t n, size_t taille, \`
`int (*comp) (const void *, const void *))` Il s'agit d'une fonction de tri (par l'algorithme Quicksort) du tableau $base[0]..base[n-1]$ dont les objets sont de taille `taille`. La fonction de comparaison `comp(x,y)` utilisée doit retourner un entier plus grand que zéro si $x > y$, nul si $x == y$ et négatif sinon.

`void *bsearch(const void *key, const void *base, size_t n,\`
`size_t taille, int (*comp) (const void *, const void *))` Cette fonction permet la recherche parmi $base[0]..base[n-1]$ (objets de taille `taille` triés par ordre croissant pour la fonction de comparaison fournie), d'un objet s'identifiant à la clé `*key`. La valeur de retour est un pointeur sur l'élément identique à `*key`, ou `NULL` s'il n'en existe aucun. La fonction de comparaison `comp()` utilisée doit obéir aux mêmes règles que pour `qsort()`.

30. Sous Unix si le programme a été lancé depuis un *shell* csh ou tcsh, la valeur de `status` est stockée dans la variable du *shell* de même nom ou dans la variable `$?` quand on utilise bash.

`int abs(int n)` Valeur absolue d'un élément de type `int`.

`long labs(long l)` Valeur absolue d'un élément de type `long`. Pour les flottants, voir `fabs()` dans <math.h>.

<assert.h> : vérifications à l'exécution

Il s'agit de la déclaration d'une macro `assert` permettant de vérifier des conditions lors de l'exécution. On écrit pour cela dans le corps du programme :

```
assert( <expression> );
```

Lors de l'exécution, si l'évaluation de `<expression>` donne zéro (expression fausse) la macro envoie sur `stderr` un message donnant l'expression, le fichier source et la ligne du problème et arrête le programme en appelant `abort()`.

Si `NDEBUG` est défini avant l'inclusion de <assert.h>, la macro `assert` n'a aucun effet.

<limits.h> , <float.h> : constantes limites

<limits.h>

CHAR_BIT	nombre de bits par caractère
CHAR_MIN, CHAR_MAX	valeurs min et max d'un `char`
SCHAR_MIN, SCHAR_MAX	valeurs min et max d'un `signed char`
UCHAR_MIN, UCHAR_MAX	valeurs min et max d'un `unsigned char`
INT_MIN, INT_MAX	valeurs min et max d'un `int`
UINT_MIN, UINT_MAX	valeurs min et max d'un `unsigned int`
SHRT_MIN, SHRT_MAX	valeurs min et max d'un `short`
USHRT_MIN, USHRT_MAX	valeurs min et max d'un `unsigned short`
LONG_MIN, LONG_MAX	valeurs min et max d'un `long`
ULONG_MIN, ULONG_MAX	valeurs min et max d'un `unsigned long`

<float.h>

FLT_DIG	nombre de chiffres significatifs pour un float
FLT_EPSILON	plus petit float f tel que $f + 1.0 \neq 1.0$
FLT_MIN	plus petit nombre représentable
FLT_MAX	plus grand nombre représentable

Les mêmes constantes existent pour les `double`, en écrivant `DBL_` au lieu de `FLT_`.

<stdarg.h> : fonctions à nombre variable d'arguments

Les déclarations de <stdarg.h> permettent de construire des fonctions ayant une liste d'arguments de longueur inconnue et ayant des types inconnus à la compilation.

La déclaration d'une telle fonction se fait comme une fonction normale, mis à part que l'on écrit des points de suspension après le dernier argument. La fonction doit avoir au moins un argument nommé. On déclare ensuite à l'intérieur de la fonction une variable de type `va_list`, et on l'initialise en appelant `void va_start(va_list vl, last)` avec comme arguments la `va_list` et le dernier argument nommé de la fonction.

Pour accéder ensuite à chaque élément de la liste d'arguments, on appelle la macro `va_arg(va_list vl, type)` qui retourne la valeur du paramètre. `type` est le nom du type de l'objet à lire. Enfin il ne faut pas oublier d'appeler `void va_end(va_list vl)` à la fin de la fonction.

Comme petit exemple, voici une fonction qui ajoute tous ses arguments, chaque argument étant précédé de son type `int` ou `double` :

Listing 26.15 – *Somme d'arguments en nombre variable*

```
#include <stdio.h>
#include <stdarg.h>

/* un type enumere decrivant les types ! */
typedef enum {INT, DOUBLE} montype;

/* chaque element est ecrit */
/* sous la forme "type" puis "valeur" */
/* le premier argument est le nombre d'elements */
double somme (int nb, ...) {
  va_list vl;
  float f=0.0;
  int i;
  int vali;
  float valf;

  va_start(vl, nb);
  for (i=0;i<nb;i++) {
    switch (va_arg(vl,montype)) {
      case INT:
       vali=va_arg(vl,int);
       printf("Ajout de l'entier %d\n",vali);
       f += (float)vali;
       break;
      case DOUBLE:
```

```
       valf=va_arg(vl,double);
       printf("Ajout du flottant %f\n",valf);
       f += valf;
       break;
      default:
       printf("Type inconnu\n");
     }
  }
  va_end(vl);
  return f;
}

main() {
  int a,b,c,d;
  double e,f,g;
  double result;

  a=2; b=3; c=-5; d=10;
  e=0.1; f=-32.1; g=3.14e-2;
  result=somme(7, INT,a,DOUBLE,e,DOUBLE,f,
                  INT,b,INT,c,INT,d,DOUBLE,g);
  printf("Somme: %f\n",result);
  exit(0);
}
```

Qui affiche à l'exécution :

```
Ajout de l'entier 2
Ajout du flottant 0.100000
Ajout du flottant -32.099998
Ajout de l'entier 3
Ajout de l'entier -5
Ajout de l'entier 10
Ajout du flottant 0.031400
Somme: -21.968597
```

Un dernier exemple classique, une fonction similaire à `printf()` :

Listing 26.16 – *Un équivalent à `printf()`*

```
#include <stdio.h>
#include <stdarg.h>

/* mini fonction printf */
/* elle utilise printf ... mais */
```

```
/* l'idee est tout de meme la */
int monprintf(char *format, ...) {
  va_list vl;
  int nb=0;

  va_start(vl, format);

  do {
    if (*format != '%') {
           nb++;
           putchar(*format);
           continue;
    }
    format++;
    switch (*format) {
        case 'd':
          nb+=printf("%d",(int)va_arg(vl,int));
          break;
        case 's':
          nb+=printf("%s",(char *)va_arg(vl,char *));
          break;
        case 'f':
          nb+=printf("%f",(double)va_arg(vl,double));
          break;
        default:
          fprintf(stderr,"Probleme de format");
    }
  } while (*(++format) !='\0');

  va_end(vl);
  return nb;
}

main() {
  int a;
  int t;
  char s[30];
  double b,c;

  a=1;
  b=2.0;
  c=a+b;
  sprintf(s,"CQFD");
```

```
  monprintf("Voila : %d + %f = %f ; %s\n",a,b,c,s);
  exit(0);
}
```

L'exécution du programme donne alors :

```
Voila : 1 + 2.000000 = 3.000000 ; CQFD
```

Index – Général

Cet index regroupe les différents termes, expressions, acronymes, évoqués dans la première partie sur les systèmes d'exploitation.

Index – Programmation

Cet index regroupe les entrées (commandes, fonctions) de la quatrième partie de ce poly concernant l'aide mémoire du langage C.

Bibliographie

[1] Maurice J. BACH.
The Design of the UNIX Operating System.
Prentice Hall, mai 1986.
URL : http://www.informit.com/store/product.aspx?isbn=0132017997.

[2] David R. BUTENHOF.
Programming with POSIX® Threads.
Addison-Wesley Professional, mai 1997.
URL : http://www.informit.com/store/product.aspx?isbn=9780201633924.

[3] Donald LEWINE.
POSIX Programmer's Guide.
O'Reilly Media., avril 1991.
URL : http://oreilly.com/catalog/9780937175736/.

[4] Marshall Kirk MCKUSICK, Keith BOSTIC, Michael J. KARELS et John S. QUARTERMAN.
The Design and Implementation of the 4.4BSD Operating System.
Addison-Wesley Professional, avril 1996.
URL : http://www.informit.com/store/product.aspx?isbn=0201549794.

[5] Marshall Kirk MCKUSICK et George V. NEVILLE-NEIL.
The Design and Implementation of the FreeBSD Operating System.
Addison-Wesley Professional, août 2004.
URL : http://www.informit.com/store/product.aspx?isbn=0201702452.

[6] Bradford NICHOLS, Dick BUTTLAR et Jacqueline Proulx FARRELL.
Pthreads Programming.
O'Reilly Media., septembre 1996.
URL : http://oreilly.com/catalog/9781565921153/.

[7] W. Richard STEVENS.
UNIX Network Programming. Networking APIs : Sockets and XTI.
2e édition. Tome 1.
Prentice Hall, janvier 1998.
URL : `http://www.kohala.com/start/unpv12e.html`.

[8] W. Richard STEVENS.
UNIX Network Programming. Interprocess Communications.
2e édition. Tome 2.
Prentice Hall, septembre 1999.
URL : `http://www.kohala.com/start/unpv12e.html`.

[9] W. Richard STEVENS et Stephen A. RAGO.
Advanced Programming in the UNIX® Environment.
2e édition.
Addison-Wesley Professional, juin 2005.
URL : `http://www.apuebook.com/`.

[10] Andrew S. TANENBAUM.
Modern Operating Systems.
3e édition.
Prentice Hall, décembre 2007.
URL : `http://www.pearsonhighered.com/educator/product/Modern-Operating-Systems/9780136006633.page`.

[11] Andrew S. TANENBAUM et Albert S. WOODHULL.
Operating Systems. Design and Implementation.
3e édition.
Prentice Hall, janvier 2006.
URL : `http://www.pearsonhighered.com/educator/product/Operating-Systems-Design-and-Implementation/9780131429383.page`.