Cours de Système : Les processus

Bertrand Le cun et Emmanuel Hyon
bertrand.le_cun{at}u-paris10.fr et Emmanuel.Hyon{at}u-paris10.fr

Université Paris Ouest Nanterre

29 novembre 2011

Abstraction d'une exécution

- C'est l'unité d'exécution visible par le S.E.
- C'est l'unité ordonnancée par le S.E. (et rien d'autre).
- Il contient le contexte d'exécution d'un programme
  - Espace d'adressage mémoire,
  - Pointeur d'instruction,
  - Pointeur de pile
  - Autre ressources système : les fichiers ouverts, les connections réseaux ouvertes

Appelé Job, Tâche, processus

Différence entre programme et processus

Programme

- Entité statique décrivant un traitement ;
- Code situé sur disque (en langage source, en langage machine)
- Un programme peut donner lieu à plusieurs processus possibles par exemple : un même programme exécuté avec des données différentes

Processus

- Entité dynamique réalisant un traitement ;
- Code situé en mémoire centrale (en langage machine) ;
- Un processus peut mettre en jeu plusieurs programmes par exemple : un programme se terminant avec le lancement d'un autre programme (recouvrement).

Image

Ensemble des composants d'une image

Un programme en cours d'exécution manipule (met en jeu)

- Code

Image

Ensemble des composants d'une image

Un programme en cours d'exécution manipule (met en jeu)

- Code
- Données
  - Statiques
  - Tas
  - Pile

Image

Ensemble des composants d'une image

Un programme en cours d'exécution manipule (met en jeu)

- Code
- Données
  - Statiques
  - Tas
  - Pile
- Contexte d'exécution
  - Pointeur d'instruction
  - Registres mémoire
  - Fichiers ouverts
  - Répertoire courant
  - Priorité

Formellement

Définition

Image : Un ensemble d'objets qui peuvent donner lieu à une exécution (un code exécutable).

Définition

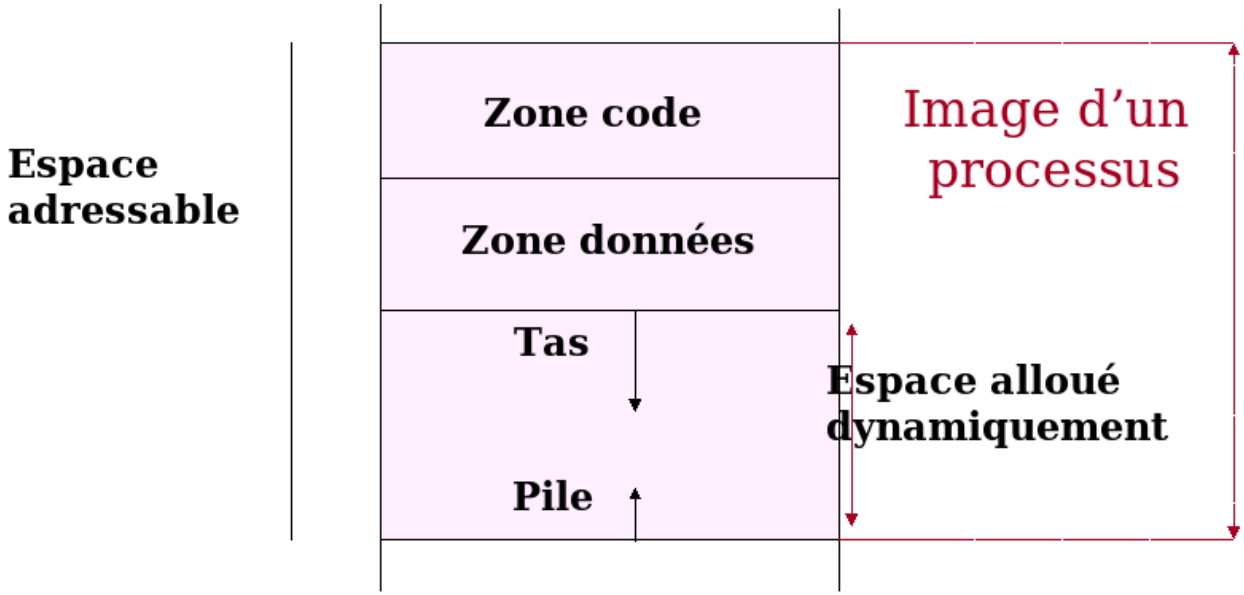
Processus : Exécution d'une image.

Illustration

Un programme (code) qui ajoute +1 à des entiers (données) sauvegardées dans un fichier (contexte). L'instance en train de s'exécuter (suivi des instructions dans l'environnement) est le processus.

Représentation de l'image dans un processus

zone utilisateur



Les processus

Identification : le PID

- Identification de processus : le PID (process id)
- `pid_t getpid(void);` Appel système retournant le PID du processus
- visibles par les commandes top, ps
- Nom de l'entrée dans le dossier /proc

Les processus

Identification : le PID

- Identification de processus : le PID (process id)
- `pid_t getpid(void);` Appel système retournant le PID du processus
- visibles par les commandes top, ps
- Nom de l'entrée dans le dossier /proc

Affiliation

- tout processus à un père : processus qui l'a lancé
- `pid_t getppid(void);` Appel système retournant le PID du processus père
- L'ancêtre $pid = 1$, le processus `init`, lancé au boot
- les orphelins (processus dont le père est mort) sont le plus souvent récupérés par init

Propriétaire d'un processus

Tout processus à un propriétaire (l'utilisateur qui a lancé le processus)

- le processus possède les même droit sur les fichiers que le propriétaire.
- Seul le propriétaire d'un processus peut le tuer.

Réel-effectif

- propriétaire réel : celui qui exécute la commande (idem pour groupe), le plus souvent celui qui a lancé la commande.
  Donnés par UID et GID obtenus avec `getuid()`, `getgid()`.
- le propriétaire effectif (respectivement groupe effectif) celui à qui appartient l'exécutable.
  Donnés par EUID et EGID obtenus avec `geteuid()`, `getegid()`.

Exemple classique : l'exécutable `passwd` exécuté par n'importe qui mais possédé par root.

États d'un processus

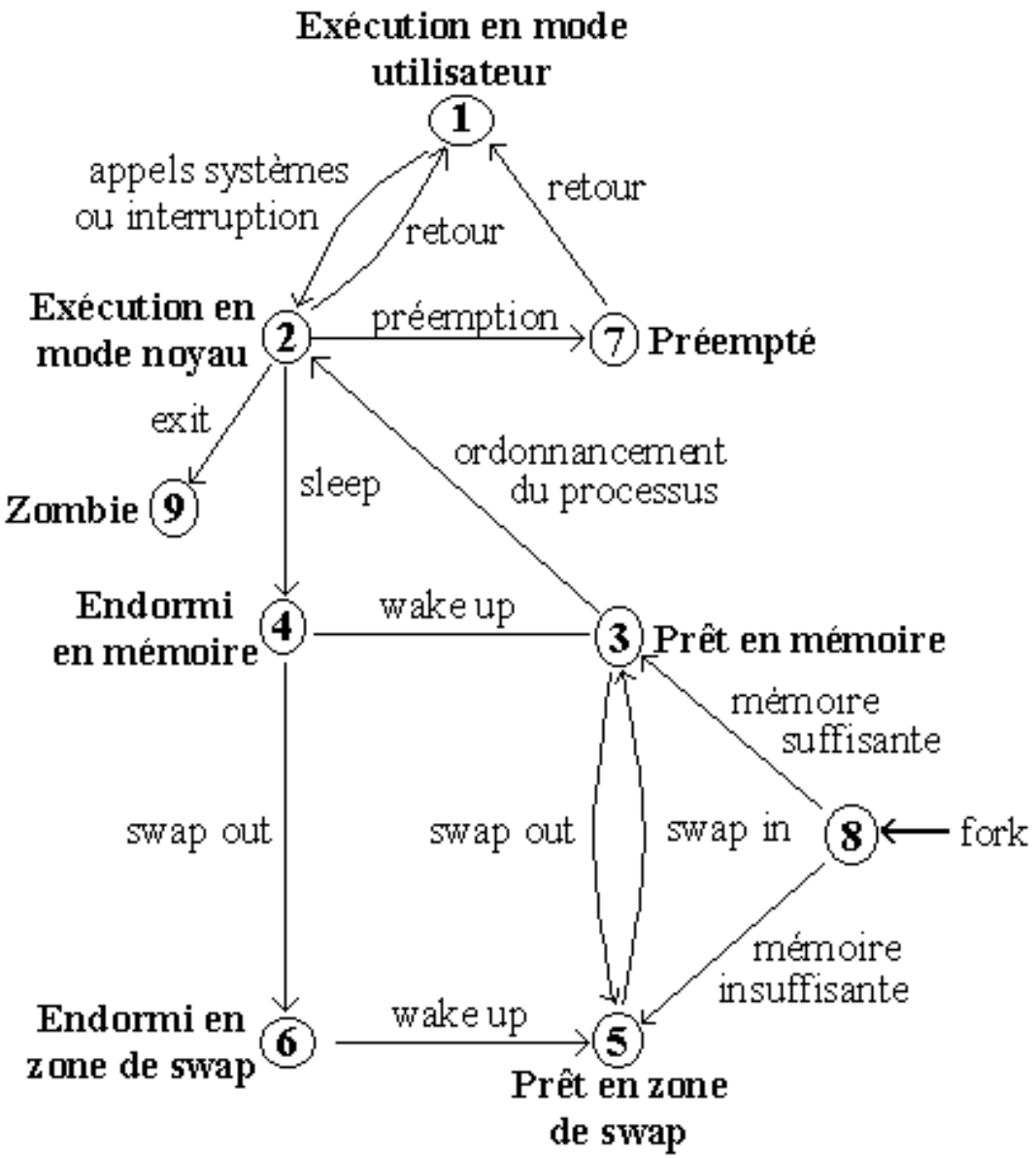
Sortie d'une commande `ps`

D	sommeil ininterruptible
R	Actif ou prêt (dans la file)
S	Sommeil interruptible (attente d'un événement)
T	Stoppé (par un signal)
X	Mort
Z	Deficient ("zombie")
	processus, terminé mais données non recup par parent

Caractères additionnels

$<$	Processus ayant une très haute priorité sur les autres
N	Basse priorité
s	session leader
l	multi-threadé
$+$	En avant plan

Graphe de transition d'un état à un autre



Création de processus

Primitive `fork()`, clônage (Sous Unix)

- Primitive système : `fork()`
- Recopie totale (données, attributs) du processus père vers son processus fils (nouveau pid).
- Le fils continue son exécution à partir de cette primitive

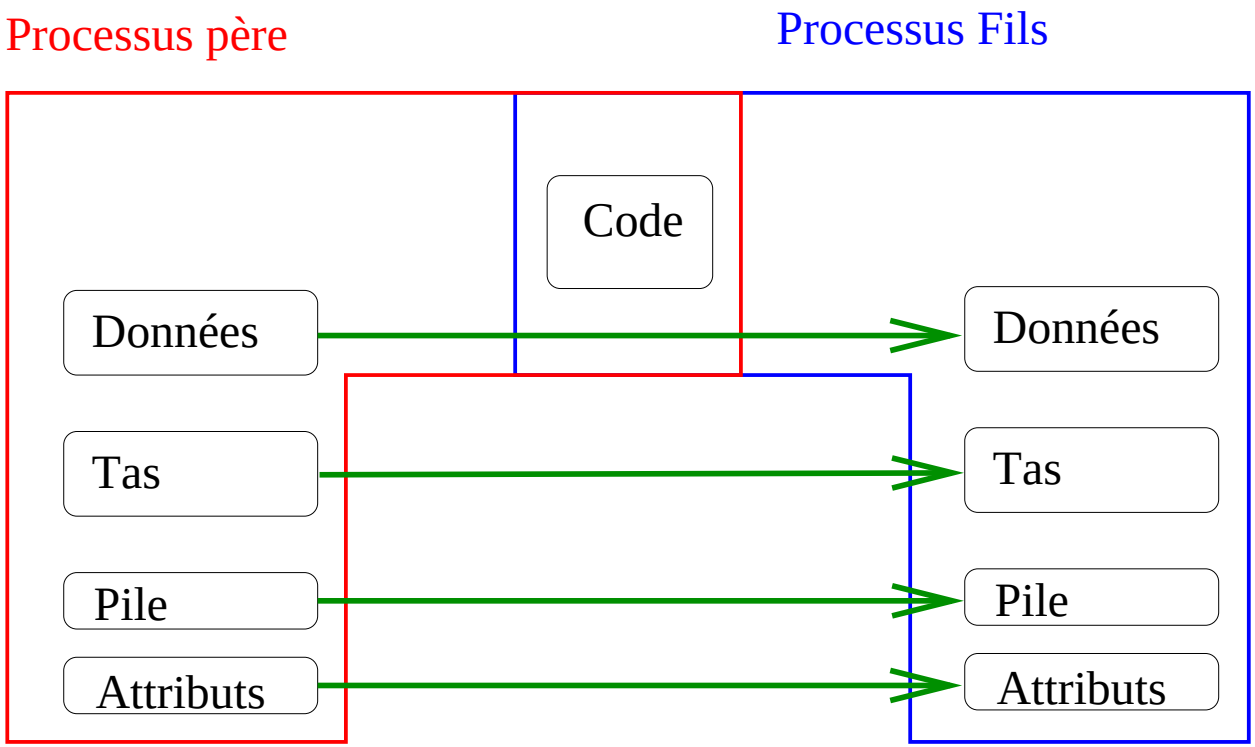
Code C : p1.c

```
#include<stdlib.h>
int main() {
 printf("Coucou %d\n",getpid());
 fork();
 printf("Hé %d\n",getpid());
}
```

Résultat

```
Machine:~/Moi> ./p1
Coucou 6874
Hé 6875
Hé 6874
Machine:~/Moi>
```

Création de processus
Clônage



Création de processus

Pour créer un processus, le système doit :

- Nommer le processus
- Créer un bloc de contrôle BCP
- Déterminer la priorité du processus
- Allouer des ressources au processus

Ensuite

- Recopie des données
- Recopie des attributs sauf
  - pid, ppid
  - signaux pendants
  - priorité
- Partage du code (si réentrance) sinon recopie du code

Création de processus *Déroulement de l'exécution*

Code C : p2.c

```
#include<unistd.h>
int main() {
    printf("Coucou %d\n",getpid());
    fork();
    fork();
    printf("Hé %d:%d\n",getpid(),
                        getppid());
}
```

Création de processus *Déroulement de l'exécution*

Code C : p2.c

```
#include<unistd.h>
int main() {
    printf("Coucou %d\n",getpid());
    fork();
    fork();
    printf("Hé %d:%d\n",getpid(),
                        getppid());
}
```

Résultat

```
Machine:~/Moi> ./p2
Coucou 7011
Hé 7013:7012
Hé 7012:7011
Hé 7014:7011
Hé 7011:5668
Machine:~/Moi>
```

Création de processus *Déroulement de l'exécution*

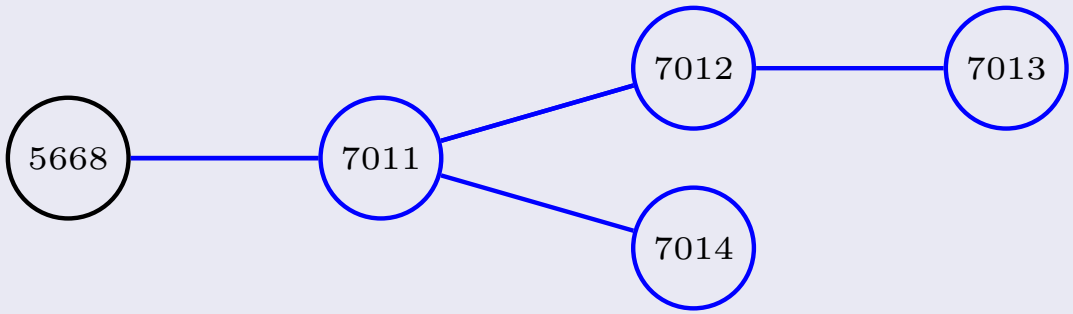
Code C : p2.c

```
#include<unistd.h>
int main() {
    printf("Coucou %d\n",getpid());
    fork();
    fork();
    printf("Hé %d:%d\n",getpid(),
                        getppid());
}
```

Résultat

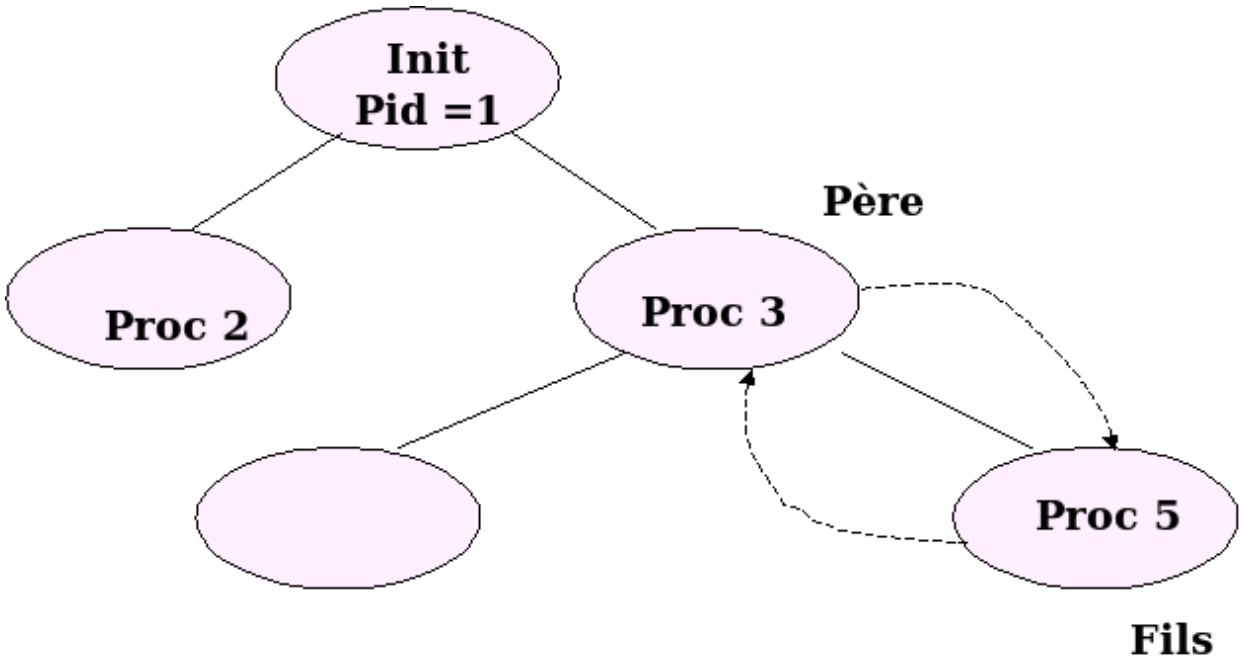
```
Machine:~/Moi> ./p2
Coucou 7011
Hé 7013:7012
Hé 7012:7011
Hé 7014:7011
Hé 7011:5668
Machine:~/Moi>
```

Hiérarchie



Création de processus
Arborescence

L'itération de `fork()` conduit à une arborescence à partir du processus init (pid =1)



Création de processus
Différence fils-père

Différencier le père du fils ?

- Code de retour du `fork()`
- Dans le père : le `fork` retourne le pid du processus fils
- Dans le fils : le `fork` retourne 0

Création de processus
Différence fils-père

Différencier le père du fils ?

- Code de retour du `fork()`
- Dans le père : le `fork` retourne le pid du processus fils
- Dans le fils : le `fork` retourne 0

Pourquoi ?

- Le fils peut connaître le pid de son père avec `getppid()`.
- Le code de retour du fork : seul moyen pour le père de connaître le pid du processus fils.

Création de processus
Exemple

Code C : p3.c

```
#include<unistd.h>
int main() { int r;
    printf("Coucou %d\n",getpid());
    if ( (r=fork())==0 )
        printf("Fils %d\n",getpid());
    else
        printf("Père %d\n",getpid());
    printf("Tous %d\n",getpid());
}
```

Création de processus
Exemple

Code C : p3.c

```
#include<unistd.h>
int main() { int r;
    printf("Coucou %d\n",getpid());
    if ( (r=fork())==0 )
        printf("Fils %d\n",getpid());
    else
        printf("Père %d\n",getpid());
    printf("Tous %d\n",getpid());
}
```

Résultat

```
Machine:~/Moi> ./p3
Coucou 7209
Père 7209
Tous 7209
Fils 7210
Tous 7210
Machine:~/Moi>
```

Portée du code

Portée du code : p4.c

```
int main(){
  int k ;
  printf(''Je suis seul au monde'\n');
  k=fork();
  if (k==-1) { printf("Erreur fork()"); exit(1);}
  if (k== 0) { /* code du fils */
      printf("Je suis le processus fils\n");
      exit(0);
  } else
      printf("Je suis le processus pere\n");
  printf("Et la qui suis-je %d\n",getpid());
}
```

Portée du code II

Portée du code : Résultat

```
Machine:~/Moi> ./p4
Je suis seul au monde
Je suis le processus fils 6681
Je suis le processus pere 6680
Et la qui suis-je 6680
Machine:~/Moi>
```

Portée des variables

Portée des variables : p5.c

```
int main(){
  int i,j,k ;
  i=5; j=2;
  if ((k=fork()) == -1) {printf("Erreur fork()");exit(1);}
  if (k== 0) { /* code du fils */
      printf("fils %d \n",getpid());
      j--;
  } else {
      printf("pere %d\n",getpid());
      i++;
  }
  printf("Pid:%d i:%d j:%d\n", getpid(),i,j);
}
```

Exemple

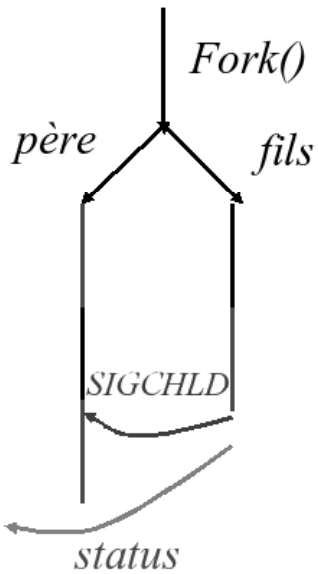
Portée des variables : Résultat de p6

```
Machine:~/Moi> ./p5
fils 6774
Pid:6774 i:5 j:1
pere 6773
Pid:6773 i:6 j:2
Machine:~/Moi>
```

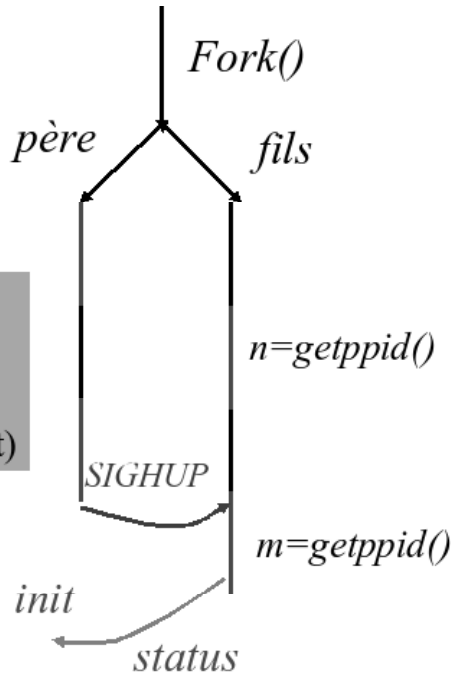
Indéterminisme du déroulement

Exécutions concurrentes

- Parallélisme d'exécution du père et du fils
- Concurrence pour l'accès aux ressources
- Ordre du déroulement (et des fins) est impossible à contrôler



Père se termine
avant le fils
n= pid du père
m=1 (pid de l 'init)



Synchronisation

- Père toujours prévenu de la fin d'un fils
- Fils toujours prévenu fin du père
- mais il faut que le père soit en attente
- Si la fin d'un fils n'est pas traitée par le père ce processus devient un processus zombie.

Synchronisation pour les fins d'exécution

- Possibilités d'échanges d'informations permettant une synchronisation sur les fins d'exécutions.
- Éviter les processus *zombies* (fin propre).
- Nécessite que le père soit en attente.
- Information de la fin du processus fils.

Appels système `wait` et `exit`

`exit` et code de retour

`exit (int)`

- valeur du `int` est “transmise” au père : code de retour
- Fin du processus fils après `exit`
- Par convention (défaut) une fin correcte donne un code de retour nul.

Appels système `wait` et `exit`

`exit` et code de retour

`exit (int)`

- valeur du `int` est “transmise” au père : code de retour
- Fin du processus fils après `exit`
- Par convention (défaut) une fin correcte donne un code de retour nul.

`wait`

`int wait(int *)`

- Entier retourné : pid du fils qui s’est terminé depuis l’invocation du wait
- Si aucun fils susceptible de se terminer alors renvoi de -1
- L’entier pointé enregistre l’état du fils lorsqu’il se finit (valeur en paramètre dans exit)

Création de processus et synchronisation

Code C : p6.c

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
int main() { int r,s,w;
  if ( (r=fork())==0 ) {
    printf("Fils %d\n",getpid());
    // Traitement long
    exit(14);
  } else {
    // Père doit attendre la
    // mort de son fils.
    w=wait(&s);
    printf("w:%d s:%d\n",w,s);}
}
```

Création de processus et synchronisation

Code C : p6.c

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
int main() { int r,s,w;
  if ( (r=fork())==0 ) {
    printf("Fils %d\n",getpid());
    // Traitement long
    exit(14);
  } else {
    // Père doit attendre la
    // mort de son fils.
    w=wait(&s);
    printf("w:%d s:%d\n",w,s);}
}
```

Résultat

```
Machine:~/Moi> ./p4
Fils 7344
w:7344 s:2560
Machine:~/Moi>
```

Recouvrement d'un processus

Nouveau code à la place d'un autre

Un processus peut changer de code par un appel système à `exec`.

- code et données remplacés
- pointeur d'instruction réinitialisé

Recouvrement d'un processus

Nouveau code à la place d'un autre

Un processus peut changer de code par un appel système à `exec`.

- code et données remplacés
- pointeur d'instruction réinitialisé

Appel Système : `exec()`

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ...,
                  const char *argn, char * /*NULL*/);
```

path = nom de l'exécutable recherché dans $PATH

arg0 = nom exécutable affiché par ps

argn = $n-2$ième argument de l'exécutable

NULL = argument de fin de la ligne de commande

Recouvrement d'un processus

Exemple : r1.c

```
#include<unistd.h>
int main() {
  printf("Je suis un programme qui va exécuter /bin/ls -l\n");
  execl("/bin/ls","ls","-l",NULL);
}
```

Recouvrement d'un processus : Résultats

```
Machine:~/Moi/Test> ls
d1  f1  f2
Machine:~/Moi/Test> ../r1
Je suis un programme qui va exécuter /bin/ls -l
total 4
drwxr-xr-x 2 blec sys 4096 2006-09-29 14:00 d1
-rw-r--r-- 1 blec sys    0 2006-09-29 14:00 f1
-rw-r--r-- 1 blec sys    0 2006-09-29 14:00 f2
Machine:~/Moi/Test>
```

Recouvrement d'un processus : déroulement

1. Le programme `r1.c` exécute le printf
2. lorsqu'il exécute le execl, il est totalement remplacé par l'exécution du "ls -l"
3. le processus tel qui l'était avec le programme, disparaît

Recouvrement d'un processus : remplacement du code

Exemple : r2.c

```
#include<unistd.h>
int main() {
   printf("Je suis un programme qui va exécuter /bin/ls -l\n");
   execl("/bin/ls","ls","-l",NULL);
   printf("Je suis un programme qui a exécuté /bin/ls -l\n");
   /// Code qui ne sera pas exécuté !!!!!
}
```

Résultats

```
Machine:~/Moi/Test> ls
d1  f1
Machine:~/Moi/Test> ../r2
Je suis un programme qui va exécuter /bin/ls -l
total 4
drwxr-xr-x 2 blec aoc 4096 2006-09-29 14:00 d1
-rw-r--r-- 1 blec aoc    0 2006-09-29 14:00 f1
Machine:~/Moi/Test>
```

Création puis Recouvrement et synchronisation

Exemple : pr1.c

```
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
int main() { int r,status,w;
  r=fork();
  if ( r==0 ) {
    execl("/bin/ls","ls","-a",NULL);
  }
  else {
    printf("Attente du fils\n");
    w=wait(&status);
    printf("w:%d s:%d\n",w,status);
} }
```

Résultats

```
Machine:~/Moi/Test>../pr1
Attente du fils
..  d1 f1 f2 d2
w:7508 s:0
Machine:~/Moi/Test>
```

Lancement d'un binaire par le shell ??

Shell lance un processus fils puis recouvrement par la commande

Accès aux fichiers et processus

Tâches incombants au système

- Le SGF doit permettre l'utilisation simultanée du même fichier par plusieurs processus (plusieurs utilisateurs).
- Nécessité de connaître les fichiers ouverts (et les fichiers ouverts simultanément par différents processus).
- Nécessité d'avoir un espace de stockage : Tampons d'un processus.

Gestion système et utilisation

- Le système pour accéder au fichier utilise les i-node.
- L'utilisateur accède au fichiers au moyen de descripteurs.

Les tables font le lien entre descripteur et i-node

Différents niveaux de tables I

Tables des descripteurs

Le système tient à jour **pour chaque processus** une table des descripteurs :

- C'est un tableau de "pointeurs"
- Une entrée dans la tables des descripteurs ne pointe pas directement sur le i-node en mémoire.
- Une entrée dans la tables des descripteurs pointe sur une entrée dans la table des fichiers ouverts.
- Un descripteur est un entier : c'est l'indice de l'entrée dans la table ($\approx$ *le numéro de la case du tableau*).

Il y a une table des descripteurs par processus.

Cette table est utilisée par les fonctions d'accès, qui transfèrent des flux d'octet (`read()`, `write()`).

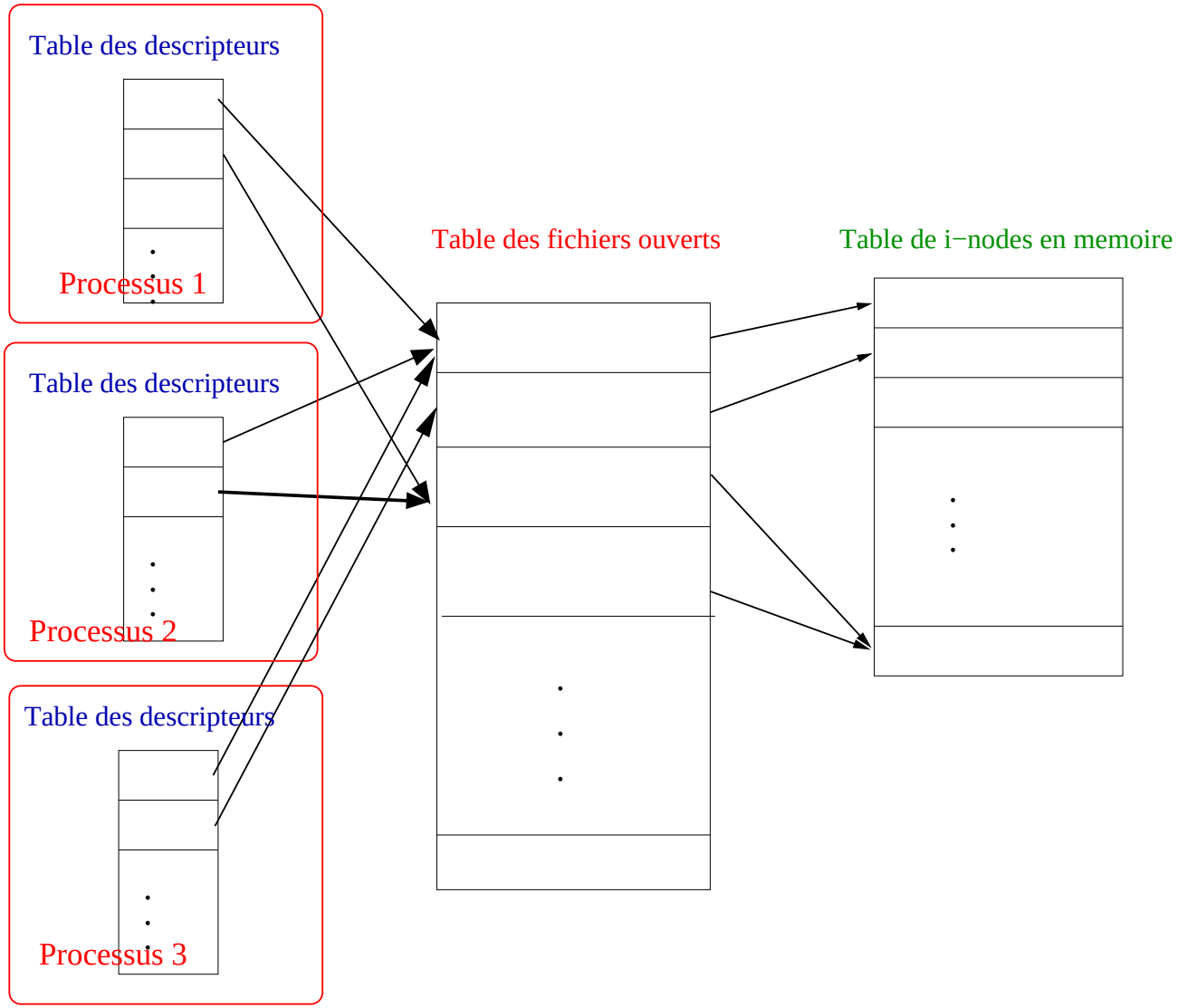
Différents niveaux de tables II

Tables des fichiers ouverts

Le système gère UNE table des fichiers ouverts, globale à tout le système.

- L'ouverture d'un fichier consiste à installer une entrée relative au fichier dans cette table
- Chaque fois qu'un processus ouvre un fichier :
  - Si aucune entrée relative au fichier : alors il création de l'entrée relative au fichier dans la table.
  - Si il existe déjà une entrée pour ce fichier dans la table (fichier déjà ouvert par un processus) : alors utilisation de cette entrée.
- Plusieurs descripteurs peuvent pointer vers la même entrée dans cette table.
- Dans cette table sont gérés
  - le pointeur de lecture courante
  - les droits d'accès par utilisateur.

Tables en mémoire



Manipulation de descripteur

Fonctions élémentaires de gestion

Un descripteur est un entier qui correspond à un numéro de la ligne dans la table qui fait référence au fichier concerné par la manipulation.

- Les fonctions `open()`, `close()` permettent d'ajouter ou de retirer des descripteurs de la table.
- Quand deux descripteurs sont ouverts sur le même fichier par `open()`, ils ont deux pointeurs courants différents.
  C'est à dire qu'il y a deux entrées différentes dans la table qui pointent sur la même entrée dans la table des fichiers ouverts.
- Un appel à `open()` prend la première entrée libre.
  D'un point de vue descripteur, il sera retourné l'indice le plus petit correspondant à la première entrée possible même s'il y a d'autres entrées d'indice supérieurs dans la table.

Duplications

Dupliquer un descripteur

Réserver une nouvelle entrée équivalente dans la table des descripteurs.

- Un nouveau descripteur pour le fichier ouvert est créé.
- Aucun nouveau pointeur n'est créé (pas ouverture nouveau fichier).
- Ces deux descripteurs partagent le même pointeur (déplacement dans fichier accédé par un, effectif pour tous).

Deux fonctions de duplication :

`int dup(desc)` et `int dup2(int desc1, int desc2)`.

```
int desc1 = open("Resultat", O_RDONLY);
int desc2, desc3;
dup2(desc1,desc2);
if (desc2 != -1)     desc3=dup(desc2);
```

Redirection

Les E/S standards

Tout processus unix a

- Sortie standard : là où s'affiche le printf
- Entrée standard : là où le scanf prend sa source
- Sortie erreur standard : là où les messages d'erreurs devraient être affichés

Nommage

nom	descripteur	FILE *
Entrée standard	0	stdin
Sortie standard	1	stdout
Sortie erreur standard	2	stderr

Redirection exemples

Exemple d1.c

```
#include<stdio.h>
#include<string.h>
int main() {
    char s[]="coucou\n";
    printf(s);
    fprintf(stdout,s);
    write(1,s,strlen(s));
}
```

Résultats

```
Machine:~/Moi/Test> ./d1
coucou
coucou
coucou
Machine:~/Moi/Test>
```

Redirection exemples

Exemple d2.c

```
#include<stdio.h>
#include<string.h>
int main() {
   char t[125],s[]="coucou";int i=4;
   printf("%s %d\n",s,i);
   fprintf(stdout,"%s %d\n",s,i);
   sprintf(t,"%s %d\n",s,i);
   write(1,t,strlen(t));
}
```

Résultats

```
Machine:~/Moi/Test> ./d2
coucou 4
coucou 4
coucou 4
Machine:~/Moi/Test>
```

Redirection

Par défaut

- Ces I/O standards sont hérités du processus père.
- Généralement, ils sont associes à un terminal (une fenêtre).

Possibilité de rediriger ces I/O Standards

Par défaut

Comment le shell fait pour gérer :

```
ls -l > toto
cat /etc/passwd | grep csh | wc -l
cat < /etc/passwd | grep csh | wc -l
```

Redirection

Rediriger la sortie standard vers un fichier

1. Fermer la sortie 1
2. Ouvrir un fichier, le descripteur sera le premier libre : 1
3. Faire toutes ses opérations comme de rien n'était

Redirection exemples

Exemple d3.c

```
#include <stdio.h>
#include <fcntl.h>
#include<string.h>
int main() {
  int d;
  char s[]="coucou\n";
  close(1);
  d=open("ficout.txt",
      O_CREAT|O_RDWR,0777);
  printf(s);
  fprintf(stdout,s);
  write(1,s,strlen(s));
}
```

Résultats

```
Machine:~/Moi> ./d3
Machine:~/Moi> cat ficout.txt
coucou
coucou
coucou
Machine:~/Moi>
```

Recouvrement et Redirection exemples

Exemple rd1.c

```
int main() {
    int d;
    close(1);
    d = open("ficout.txt",O_CREAT|O_RDWR,0777);
    execl("/bin/ls","ls","-l",NULL);
}
```

Résultats

```
Machine:~/Moi/Test> ../rd1
Machine:~/Moi/Test> more ficout.txt
total 4
drwxr-xr-x 2 blec aoc 4096 2006-09-29 14:00 d1
-rw-r--r-- 1 blec aoc    0 2006-09-29 14:00 f1
-rw-r--r-- 1 blec aoc    0 2006-09-29 14:00 f2
-rwxr-xr-x 1 blec aoc    0 2006-09-29 15:42 ficout.txt
Machine:~/Moi/Test>
```

Fork et Redirection

Fils = Père

- Le fils hérite des descripteurs ouverts de son père.
- Le fils hérite donc des I/O standards.

Code dp1.c

```
int main() { int r;
    int d;
    close(1);
    d = open("ficout.txt",O_CREAT|O_RDWR,0777);
    printf("Coucou %d\n",getpid());
    if ( (r=fork())==0 )
        printf("Fils %d\n",getpid());
    else
        printf("Père %d\n",getpid());
    printf("Tous %d\n",getpid());
}
```

Redirection exemples

Résultats

```
Machine:~/Moi/Test> ./dp1
Machine:~/Moi/Test> cat ficout.txt
Coucou 7951
Fils 7952
Tous 7952
Père 7951
Tous 7951
Machine:~/Moi/Test>
```

Les IPC
Inter Processes Communication

Enchaînement de commandes

Enchaînements séquentiels

```
date; whoami; echo ''fin de composition'' > toto
```

ou

```
( date; whomai; echo "fin de composition" ) > toto
```

- Enchaînements séquentiels ;
- Processus indépendants ;
- Aucune communication.

Enchaînement de commandes

Enchaînements séquentiels

```
date; whoami; echo ''fin de composition'' > toto
```

ou

```
( date; whomai; echo "fin de composition" ) > toto
```

- Enchaînements séquentiels ;
- Processus indépendants ;
- Aucune communication.

Séquentiel avec communication

Un moyen pour enchaîner des commandes avec communication entre les tâches : Utiliser la redirection.

```
$ ps -aux > /tmp/toto
$ wc -l < /tmp/toto
$ rm /tmp/toto
```

Concurrence et communication

Le tube

- Permet de lancer un certain nombre de processus concurrents, (Concurrents notamment pour l'accès aux ressources)
- Communiquant entre eux.
- Système doit assurer la synchronisation entre eux.
  - Bloque le processus lecteur quand le tube est vide.
  - Bloque le processus écrivain quand le tube est plein

Exemple :

Sortie standard redirigée vers entrée d'un tube et entrée standard redirigée vers la sortie du tube

Syntaxe :

```
commande_1 | commande_2 | ... | commande_n
```

Exemple : `$ ps -aux | wc -l`

Généralité sur les *tubes*

Échanges de données entre processus

- Les tubes (ou “pipe”) permettent à un groupe de processus d’envoyer des données à un autre groupe de processus.
- Les données sont envoyées directement en mémoire et ne sont pas stockées temporairement sur le disque dur (permet une grande rapidité).
- Un tube appartient aux mécanismes liés au SGF.
  - un tube correspond à un noeud dans un disque logique
  - un tube est désigné par un (des) descripteur(s).
  - un tube peut donc être manipulé par les primitives classiques : `read`, `write`, ...

Généralité sur les *tubes* II

Caractéristiques techniques

- Un mécanisme de communication unidirectionnel (tuyau).
- Communication d'un flot continu de caractères.
  Envois successifs apparaissent à l'extraction comme une seule émission.
- Un tube a deux extrémités :
  - une permettant d'écrire des données
  - une permettant de les lire.
- Chaque extrémité est un descripteur de fichier ouvert
  - soit en lecture.
  - soit en écriture.
- A un tube correspond au plus deux entrées dans la table des fichiers ouverts.

Tube ordinaire

- Un tube n'a pas de nom : `open()` non applicable.
- L'acquisition des descripteurs est réalisée par
  - Un appel à la primitive de création du tube
  - Un héritage : un processus fils hérite des descripteurs que la père possède.

Conséquences :

- Dialogue possible uniquement entre processus de la descendance d'un même processus père.
- Aucune possibilité de récupérer un nouvel accès si perte d'un des descripteur.

Création de *tube*

Primitive pipe()

Le prototype :

```
include <unistd>
int pipe(int p[2]);
```

- Appel à la primitive avec `int p[2]` en paramètre.
  - p[0] descripteur en lecture
  - p[1] descripteur en écriture
- valeur de retour 0 (ok) ou -1 (plantage).

Création de *tube*

Tube doit être créé **avant** le fork pour le partage entre père et fils

```
include <unistd>
int p[2];
res= pipe(p);
if (res = -1){
   printf(‘‘pas de païpe créé. fin programme’’); exit(1);}
if (fork()==0){
   close(p[0]);
   printf(‘‘Fils :  peux ecrire mais pas lire\n’’);
   close(p[1]);  exit(0);
}
else{
   close(p[1]);
   printf(‘‘Pere: peux lire mais pas ecrire\n’’);
   close(p[0]); exit(0);}
```

Lecture dans un *tube* : propriétés

La lecture d'un tube est bloquante :

Si aucune donnée n'est disponible en lecture,

- Le processus tentant de lire sera suspendu (instruction non traitée)
- Déblocage lorsque des données sont à nouveau disponibles.

Effet de bord : Synchronisation des processus entre eux : Les lectures synchronisées sur les écritures.

La lecture d'un tuyau est destructrice :

La lecture efface les données lues de l'espace mémoire associé au tube.
Effet de bord : Si plusieurs processus lisent le même tuyau, toute donnée lue (consommée) par l'un disparaît pour les autres.
(ex. Un processus écrit les caractères `ab` dans un tube lu par les processus A et B. A lit un caractère, puis B lit. Was passiert ?)

Lecture dans un *tube* : Exemple

Primitive de lecture

La lecture se fait par la primitive

`nb_lu=read(p[0],&buf,sizeof(buf))`.

Comportement

1. Si le tube n'est pas vide.
   On suppose qu'il contient `N` octets. La primitive extrait `nb_lu=min(N,sizeof(buf))` octets qui sont placés dans `buf`.
2. Si le tube est vide.
   - Si le nombre d'écrivains est nul alors la fin de fichier est atteinte : `nb_lu=0` ;
   - Si le nombre d'écrivain n'est pas nul
     Si lecture bloquante (cas par défaut) alors le processus est mis en attente.
     Si lecture est non bloquante : erreur retour immédiat avec valeur `nb_lu=-1`.

Écriture dans un *tube* : Propriétés

Écriture est bloquante :

- Un processus essayant d'écrire dans un tube plein se verra suspendu en attendant qu'un espace suffisant se libère.
  - Un tube de données à une capacité finie.
  - Le tube peut donc être un tube plein.
- Un processus essayant d'écrire si aucun lecteur disponible ne pourra pas il y aura un signal de pipe brisé.

Écriture en mode FIFO

Gestion des tubes telle que l'information écrite la première est la première lue.

Écriture dans un *tube* : Exemple

Primitive

La lecture se fait par la primitive

`nb_ecrit=write(p[1],&buf,sizeof(buf))`.

Comportement

1. Si le nombre de lecteur est nul. Fin du processus : Signal SIGPIPE est délivré au processus écrivain.
2. Si le nombre de lecteurs est non nul
   1. Si écriture est bloquante (cas par défaut) renvoie de `nb_ecrit` une fois les `nb_ecrit` caractères écrits dans le tube.
   2. Si écriture non bloquante `nb_ecrit` dépend de la taille libre dans buffer)

Écriture dans un *tube* : Exemple

Primitive

La fermeture se fait par la primitive `close(p[1]);`
La duplication se fait par la primitive `desc=dup(p[1]);`
Attention on ne peut avoir un descripteur sur un pipe que par l'intermédiaire d'un héritage (à les nantis !).
La fermeture du descripteur peut avoir des conséquences irréversibles

Exemple

```
#include <unistd.h>
#include <stdio.h>
main() {
  int p[2]; char ch[50];
  pipe(p);
  if (fork()==0){
    close(p[0]);
    printf("fils ecrit\n");
    sprintf(ch,"ca va papa?\n");
    write(p[1],ch,sizeof(ch));
  } else{
    close(p[1]);
    read(p[0],ch,sizeof(ch));
    printf("je suis le pere\n");
    printf("fils dit %s'',ch);}
}
```

Exemple

```
#include <unistd.h>
#include <stdio.h>
main() {
  int p[2]; char ch[50];
  pipe(p);
  if (fork()==0){
    close(p[0]);
    printf("fils ecrit\n");
    sprintf(ch,"ca va papa?\n");
    write(p[1],ch,sizeof(ch));
  } else{
    close(p[1]);
    read(p[0],ch,sizeof(ch));
    printf("je suis le pere\n");
    printf("fils dit %s'',ch);}
}
```

Résultat

```
Machine:~/Moi> ./pg1
Fils ecrit
je suis le pere
fils dit ca va papa?
Machine:~/Moi>
```

Points d'attentions

Problèmes

Pour envoyer des informations identiques à plusieurs processus, que faut il faire ?

il faut créer un tuyau vers chacun d'eux.

Comportement en fonction du nombre de lecteurs ou écrivains.

Un même processus peut disposer de plusieurs descripteurs.

- Nombre de lecteurs : Le nombre de descripteurs associés à l'entrée en lecture sur le tube.
  Si ce nombre est nul : Écriture est bloquante.
- Nombre d'écrivains : Le nombre de descripteurs associés à l'entrée en écriture.
  Si ce nombre est nul cela définit la fin de fichier pour la primitive read lorsque le tube est vide.

Règles de programmation

Problèmes

Un processus avec plusieurs lecteurs est source de problèmes potentiels.

- sur la cohérence des données
- sur le fonctionnement du programme :
  Omettre de fermer le côté inutile peut entraîner l'attente infinie d'un des processus si l'autre se termine.

Règles de programmation

Problèmes

Un processus avec plusieurs lecteurs est source de problèmes potentiels.

- sur la cohérence des données
- sur le fonctionnement du programme :
  Omettre de fermer le côté inutile peut entraîner l'attente infinie d'un des processus si l'autre se termine.

Usages

- le processus père doit fermer le côté lecture après l'appel à `fork()` (idem pour un tuyau ayant plusieurs écrivains) .
- le processus fils doit fermer le côté écriture après l'appel à `fork()` (idem pour un tube ayant plusieurs lecteurs).
- Ou alors échange de signaux.

Exemple de problème

Interblocage

```
#include <unistd.h>
main() {
int p1[2],p2[2];
char ch;
pipe(p1);
pipe(p2);
if (fork()==0){
  read(p1[0],ch,sizeof(ch));
  write(p2[1],ch,sizeof(ch));
} else{
  read(p2[0],ch,sizeof(ch));
  write(p1[1],ch,sizeof(ch));
}
}
```

Les tubes nommés

Même fonctionnement mais utilisation du SGF

- Principal inconvénient des tubes :
  fonctionnement avec des processus issus d'un ancêtre commun.
- Principal avantage :
  Les mécanismes de communication mis en jeu dans le noyau sont généraux.

Pour s'affranchir des inconvénients, les processus doivent désigner le tube qu'ils souhaitent utiliser. Ceci se fait grâce au système de fichiers :

Un tuyau nommé est donc un fichier

Les tubes nommés II

Primitives

Création du tube nommé

```
int mkfifo(const char *ref, mode_t mode);
```

Où :

- `ref` chemin du tube.
- `mode` (même fonctionnement que open) avec un ou exclusif `S_IRUSR S_IWUSR S_IWGRP`.

Suppression du tube

primitive `unlink`