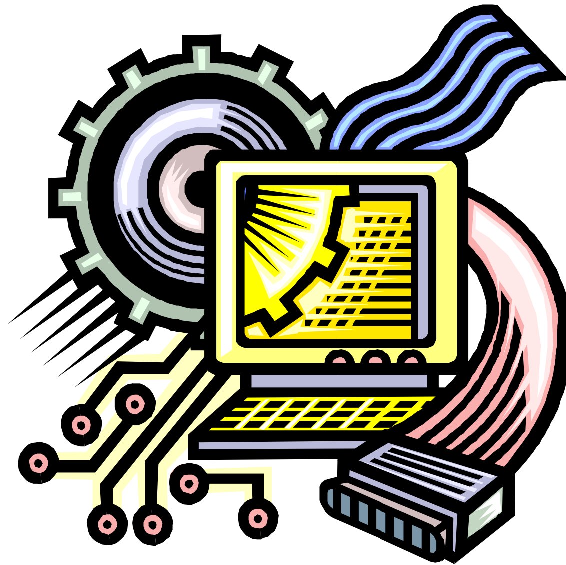
Processus- Ordonnancement
Processus Linux - Threads

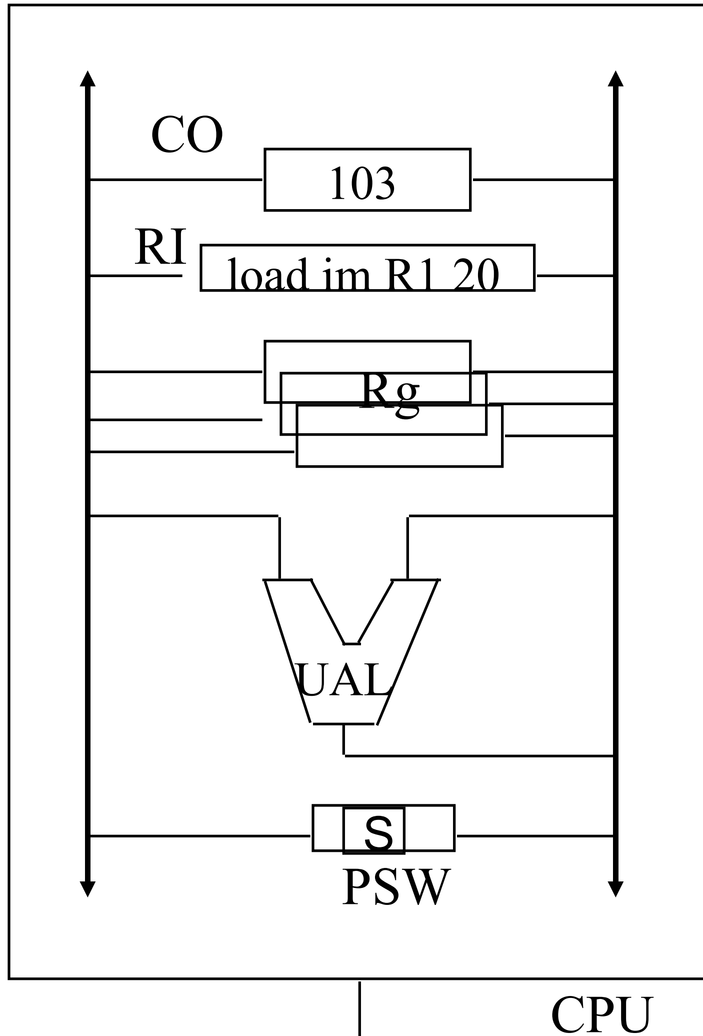


Processus

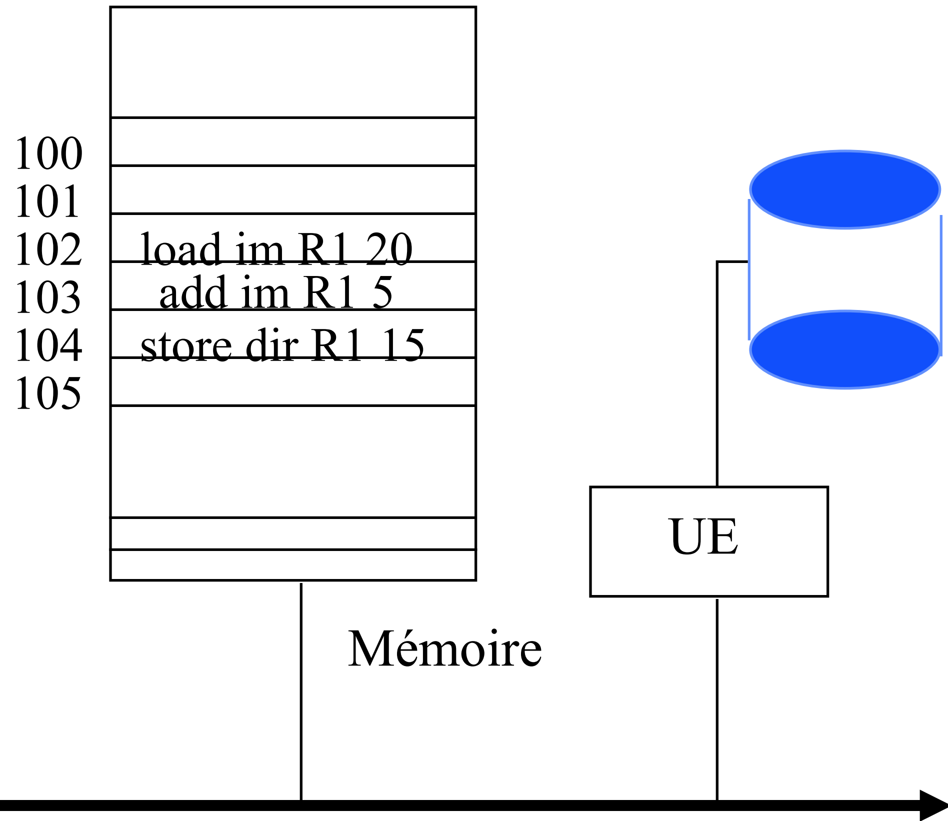
Un processus est une exécution de programme

Notion de processus

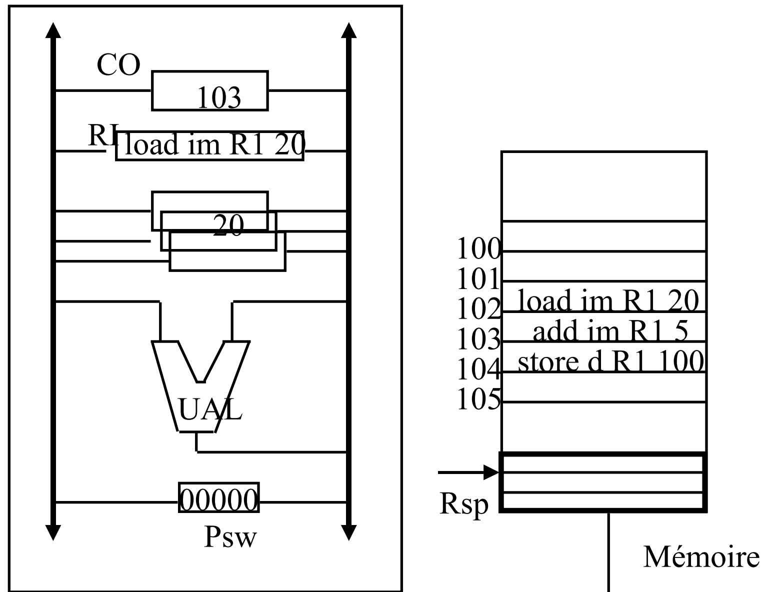
- ✓CO : compteur ordinal
- ✓PSW : registre d'état
- ✓RI : registre instruction

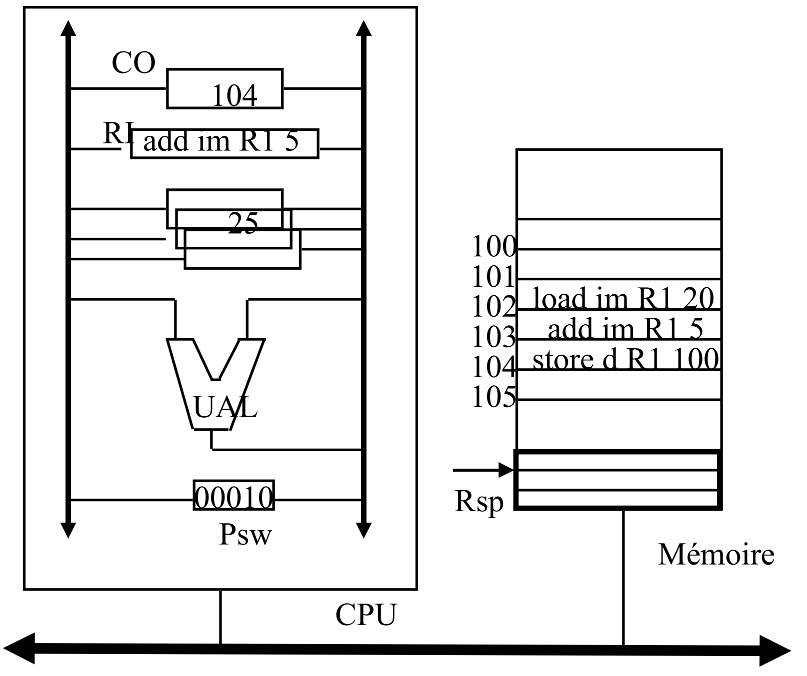


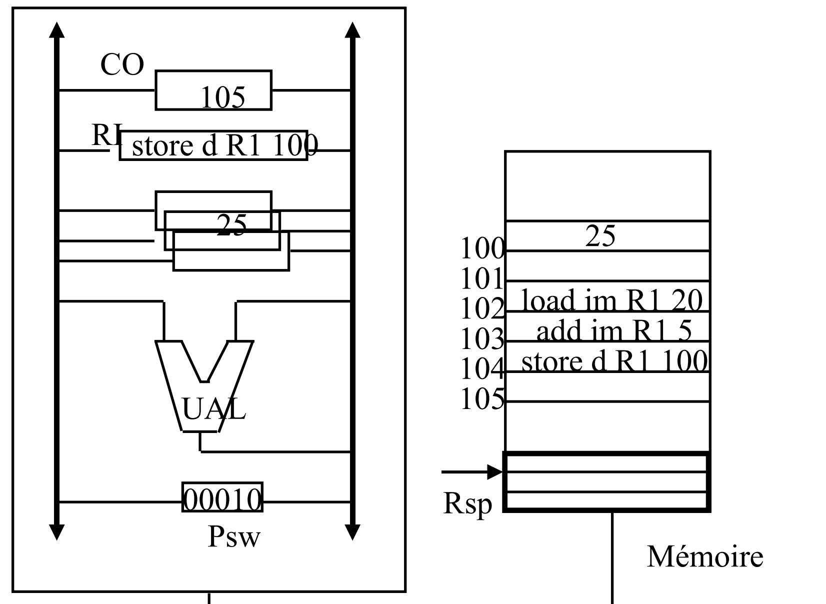
CPU



Bus externe


CO
103
RI
load im R1 20
20
UAL
00000
Psw
CPU
100
101
102
103
104
105
load im R1 20
add im R1 5
store d R1 100
Rsp
Mémoire
Bus externe


CO
104
RI
add im R1 5
25
UAL
00010
Psw
CPU
100
101
102
103
104
105
load im R1 20
add im R1 5
store d R1 100
Rsp
Mémoire


CO
105
RI
store d R1 100
25
UAL
00010
Psw
CPU
100
101
102
103
104
105
25
load im R1 20
add im R1 5
store d R1 100
Rsp
Mémoire
Bus externe

Notion de processus

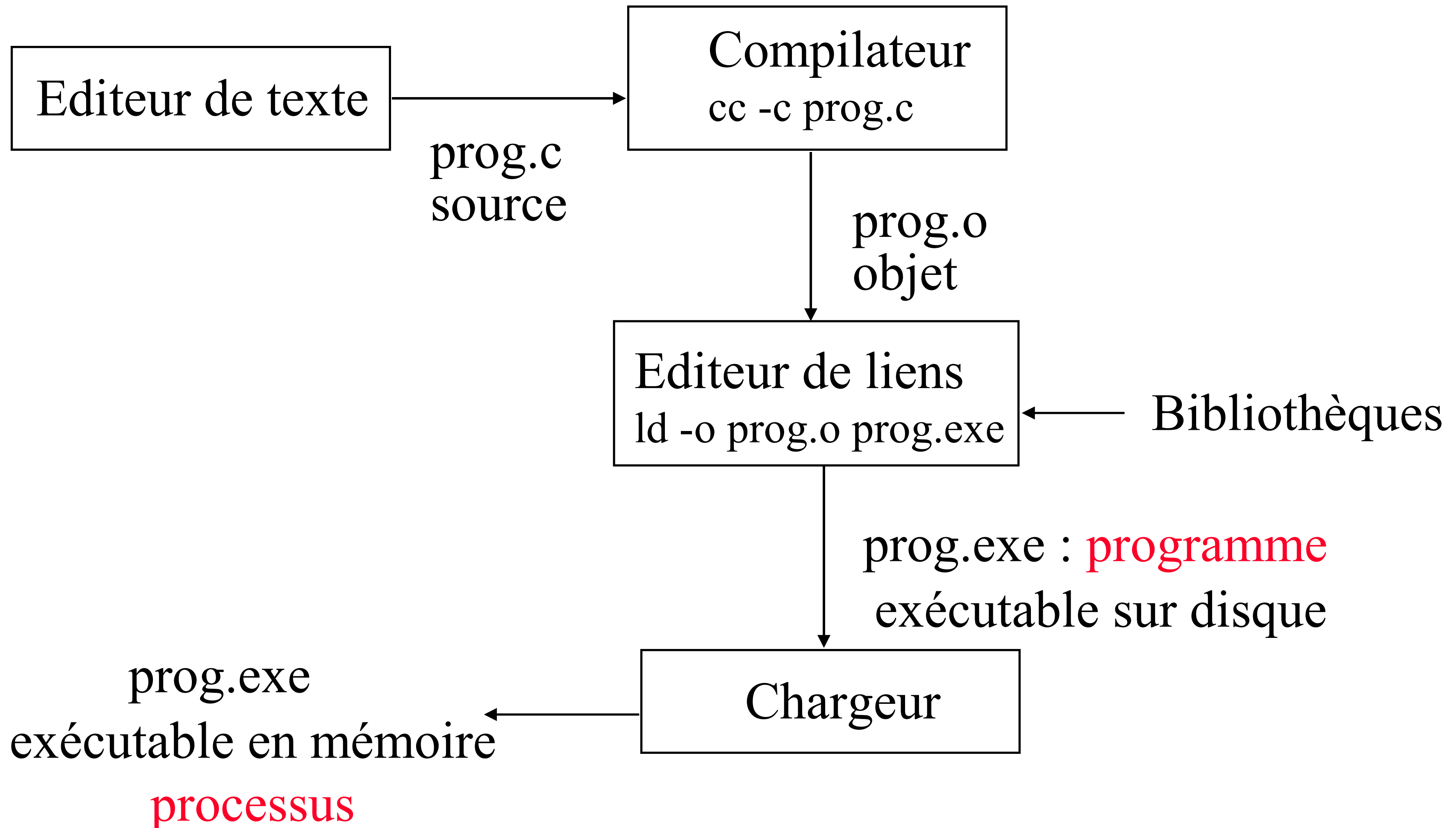
- **Définition**

  - **Un processus est un programme en cours d ’exécution auquel est associé un environnement processeur (CO, PSW, registres généraux) et un environnement mémoire appelés contexte du processus.**

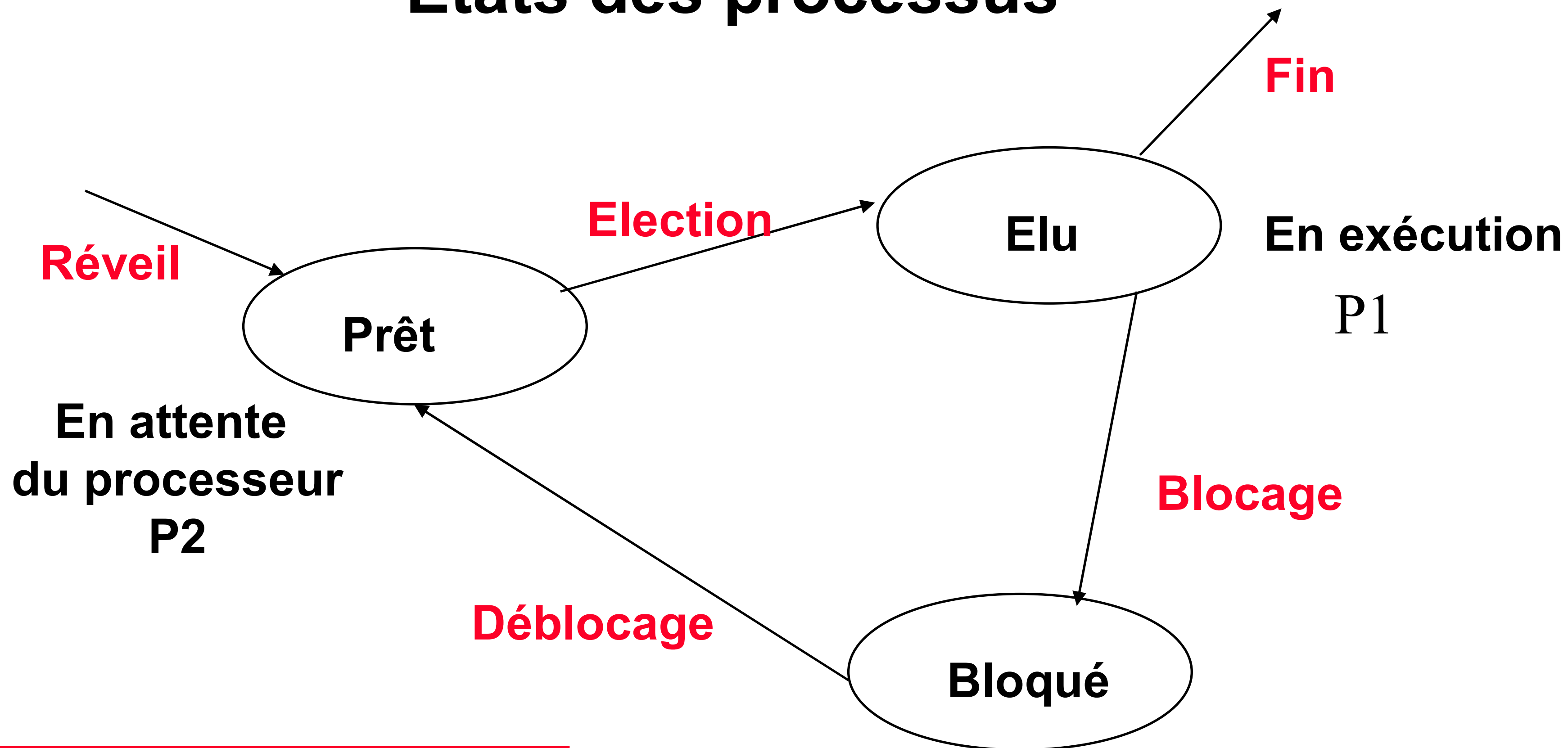
  - **Un processus est l ’instance dynamique d ’un programme et incarne le fil d ’exécution de celui-ci**

  - **Un processus évolue dans un espace d’adressage protégé**

Du programme au processus



Système multiprocessus
Etats des processus



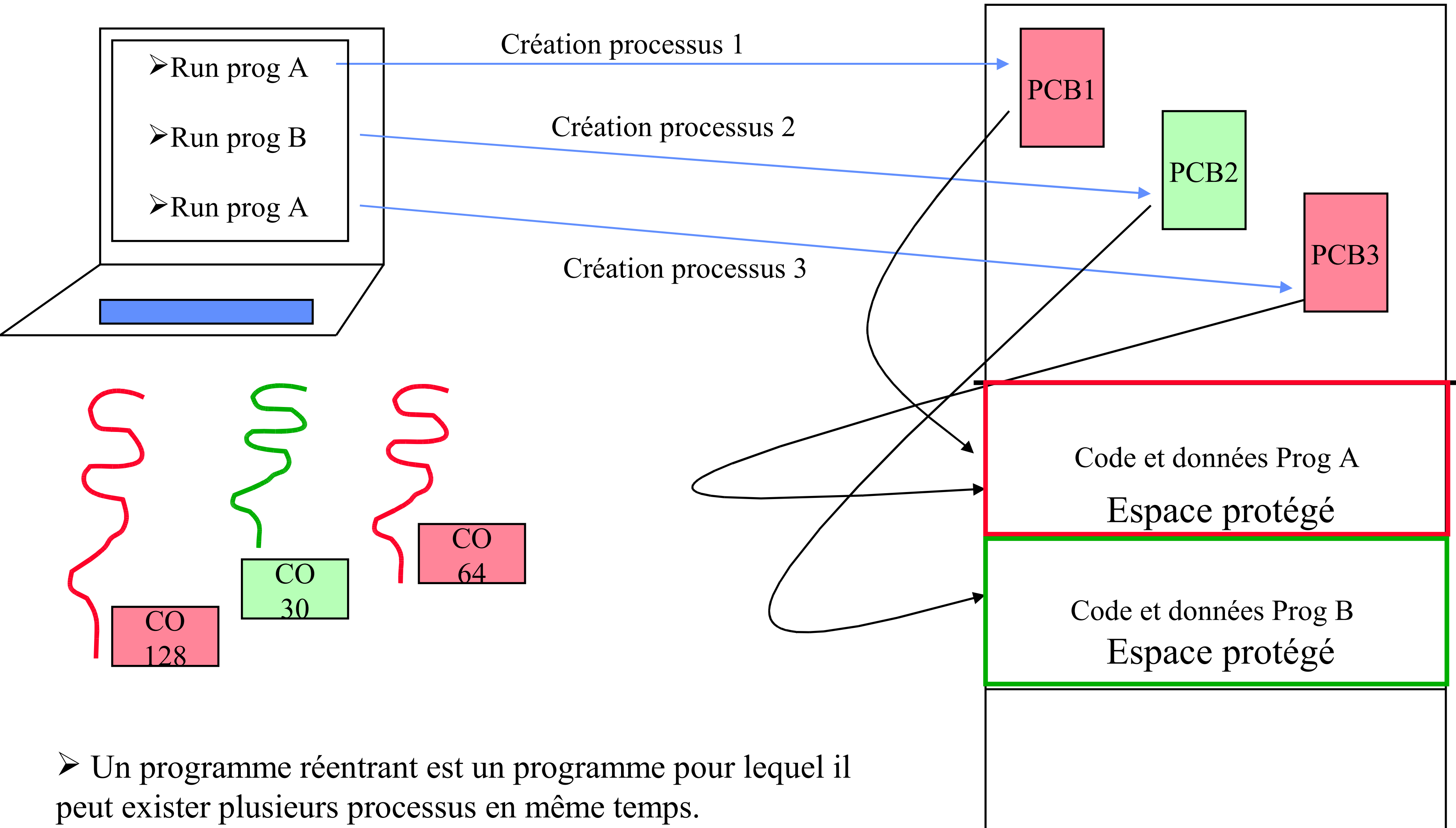
Une ressource désigne toute entité (matérielle, logicielle)
dont a besoin un processus pour s'exécuter
(processeur, variable)

bloc de contrôle de processus PCB

identificateur processus
état du processus
compteur instructions
contexte pour reprise (registres et pointeurs, piles,..)
pointeurs sur file d' attente et priorité(ordonnancement)
informations mémoire (limites et tables pages/segments
informations de comptabilisation et sur les E/S, périphériques alloués, fichiers ouverts,..

Bloc de contrôle de processus ou PCB

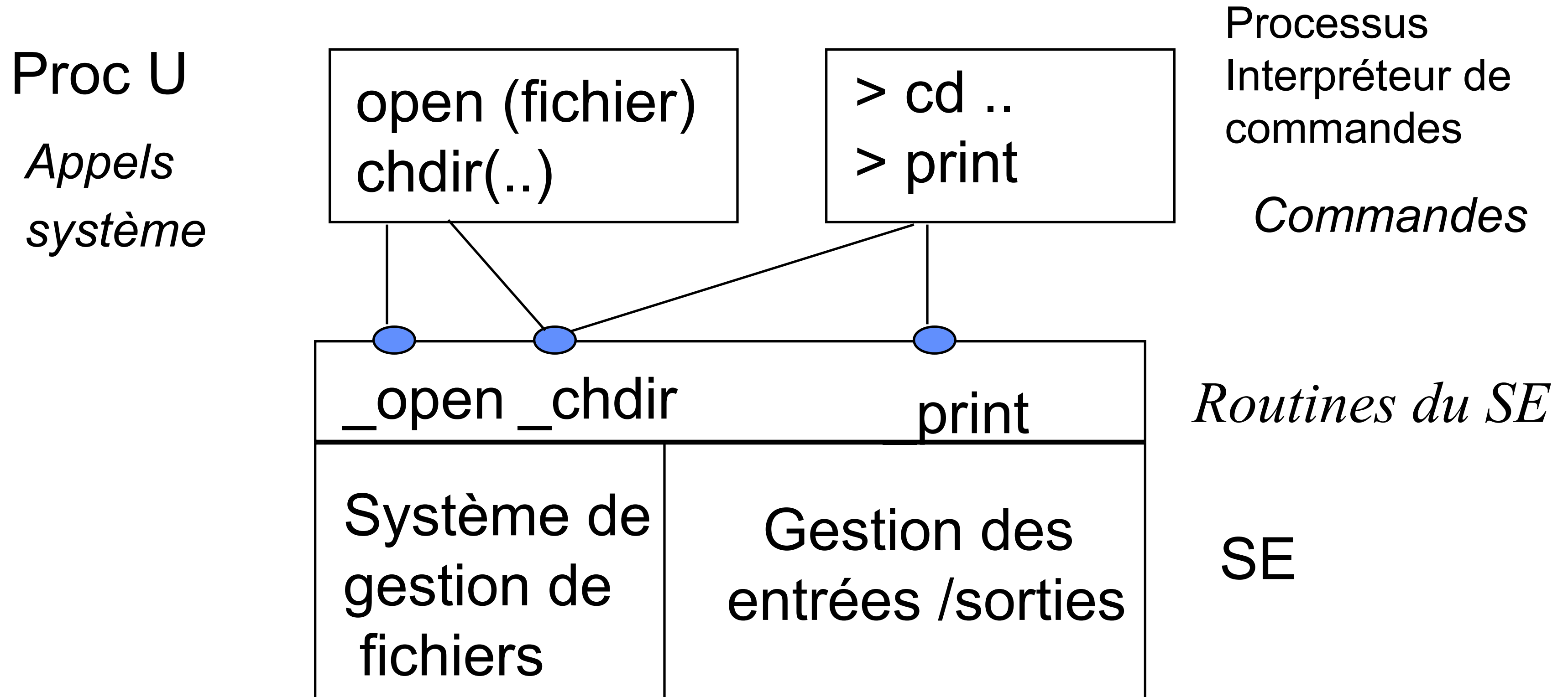
Notion de processus
programme réentrant



➢ Un programme réentrant est un programme pour lequel il peut exister plusieurs processus en même temps.

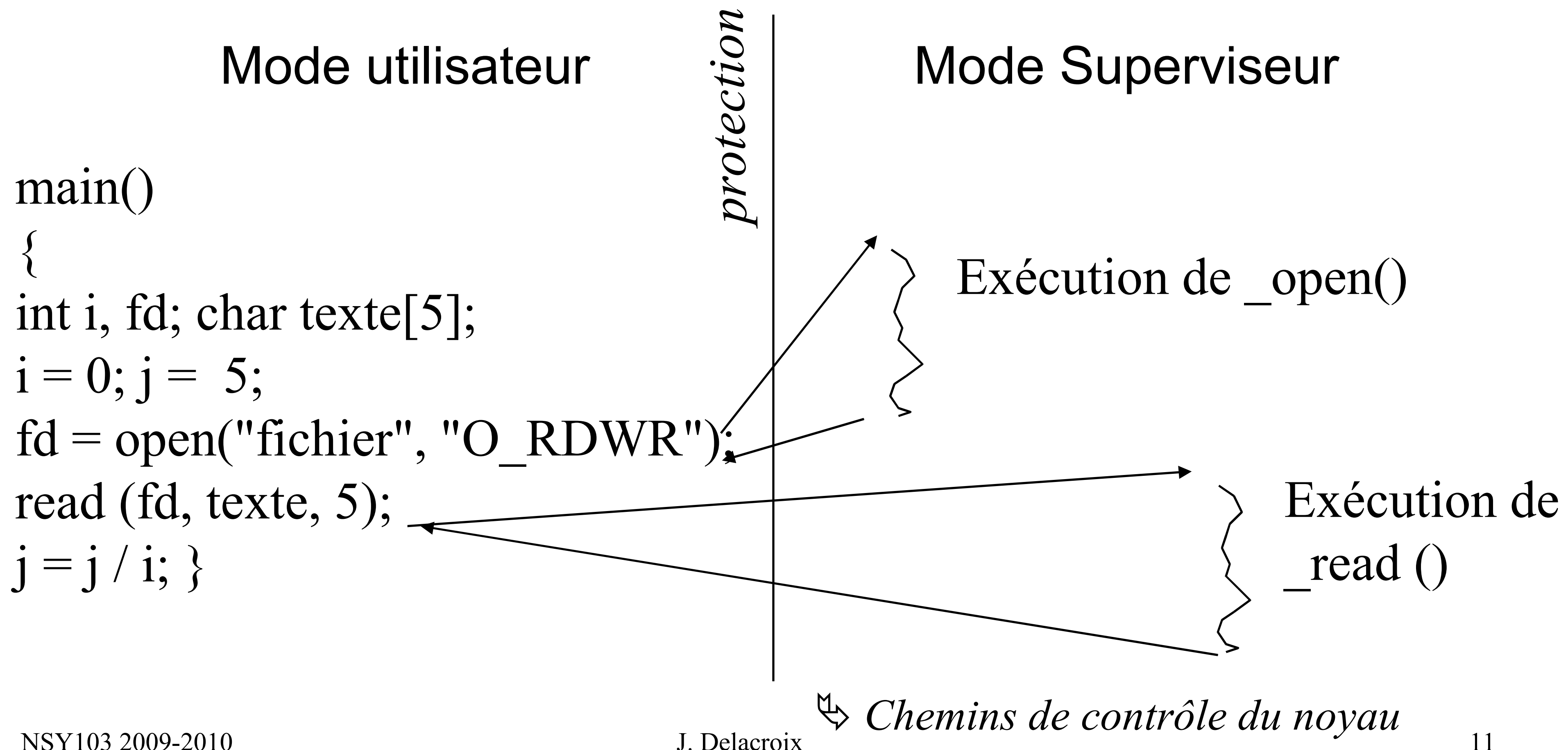
Modes d'exécution d'un processus

- **Les fonctionnalités du système d'exploitation sont accessibles par le biais des commandes ou des appels système**



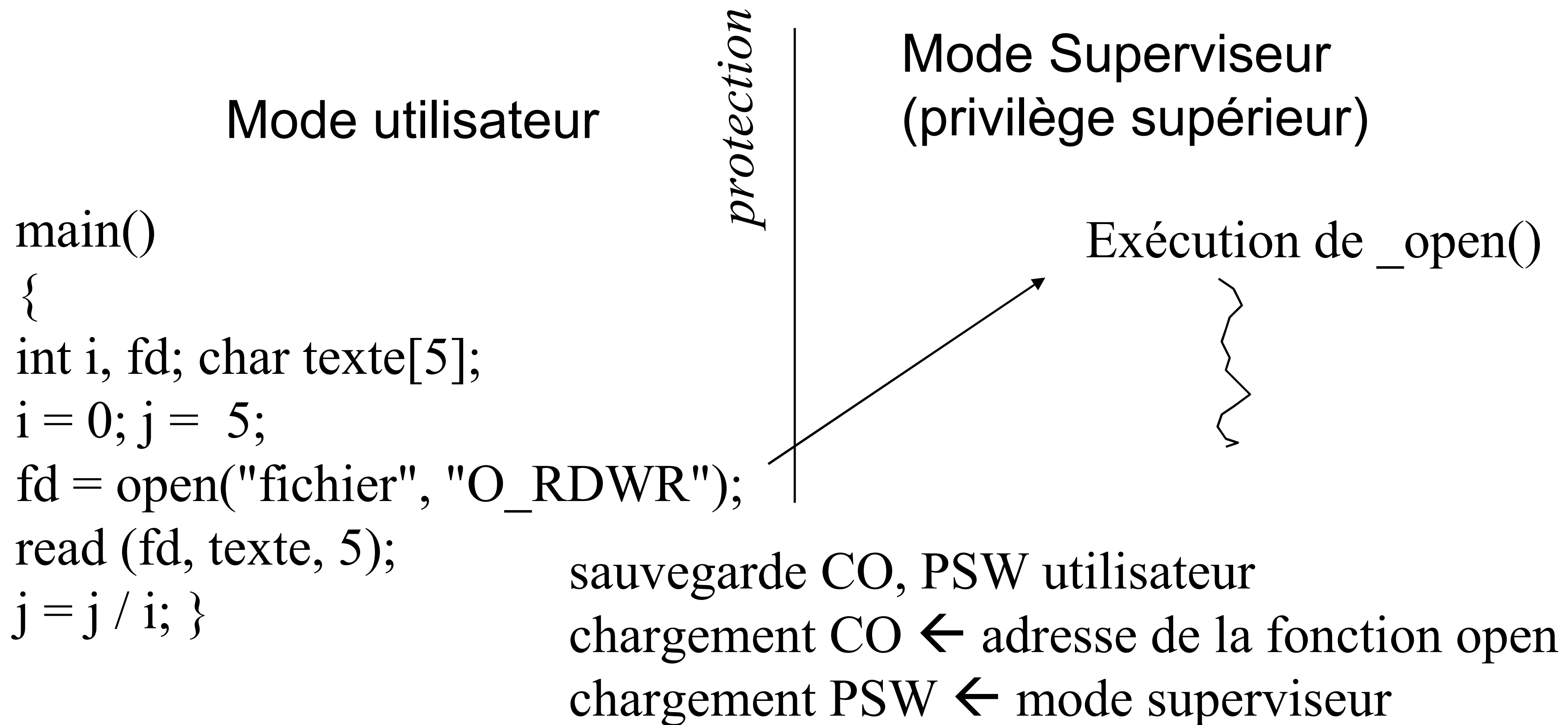
Modes d’exécution d’un processus

- Lors de l'exécution d'un appel système, le processus utilisateur passe d'un mode d'exécution dit utilisateur à un mode d'exécution dit superviseur.



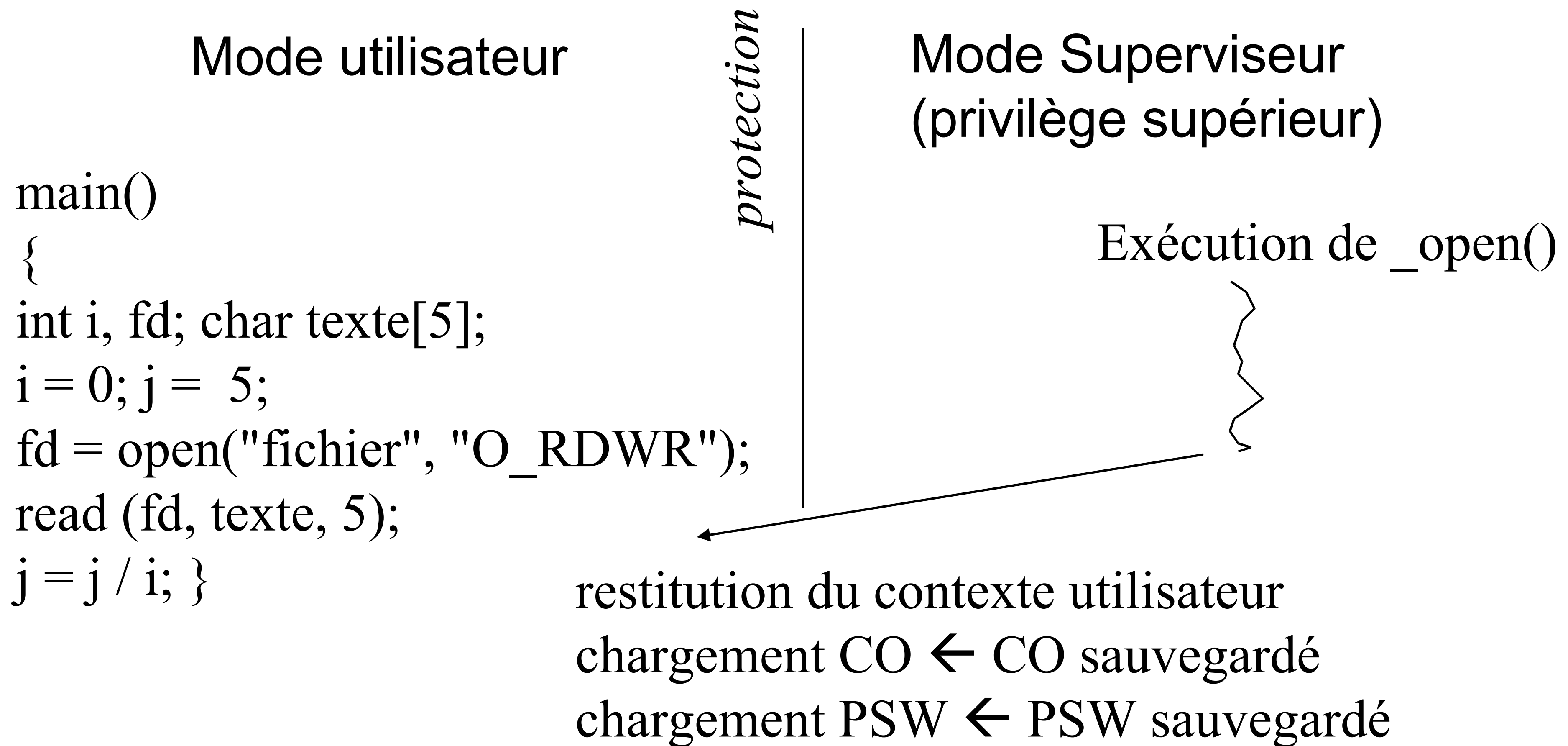
Modes d'exécution d'un processus

- Le passage du mode utilisateur au mode superviseur s'accompagne d'opérations de commutation de contexte : sauvegarde de contexte utilisateur



Modes d'exécution d'un processus

- Le passage du mode superviseur au mode utilisateur s'accompagne d'opérations de commutation de contexte : restitution de contexte utilisateur



Modes d'exécution d'un processus

Mode utilisateur | *protection* | Mode Superviseur

```
main()
{
int i, fd; char texte[5];
i = 0; j =  5;
fd = open("fichier", "O_RDWR");
read (fd, texte, 5);
j = j / i; }
```

Exécution de open()
APPELS SYSTEME

TRAPPE
erreur irrécouvrable
arrêt du programme

IT
Exécution du
traitant d'it Horloge

Trappe = interruption synchrone
IT = interruption asynchrone

Ordonnancement

Ordonnancement dans un système multiprocessus

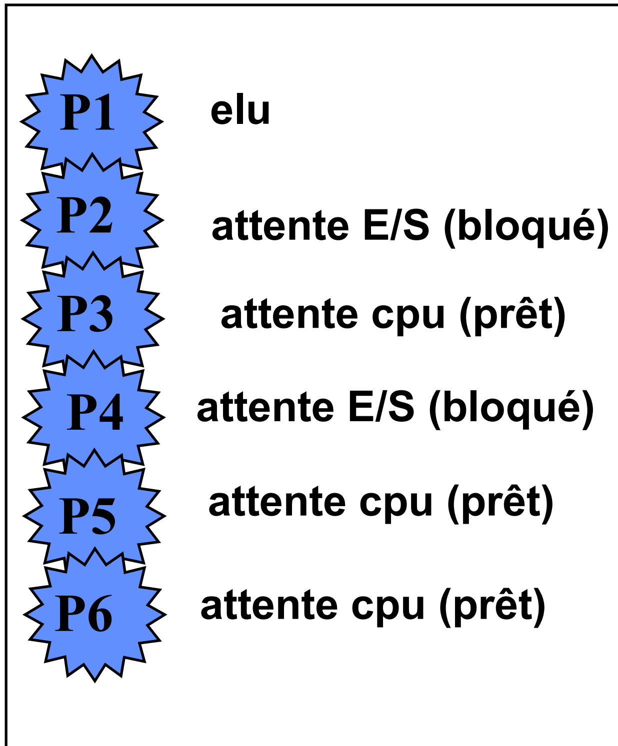
Cette fonction planifie l'exécution des processus

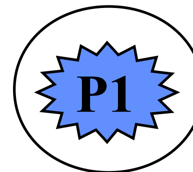
Système multiprocessus

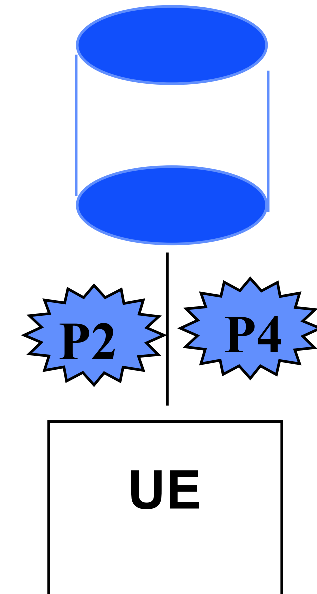
P1 se termine : quel processus exécuter parmi les processus prêts P3, P5, P6

P2 achève son entrées-sortie et devient prêt : P1 doit-il poursuivre son exécution ou laisser sa place à P2 (préemption) ?

Mémoire Centrale



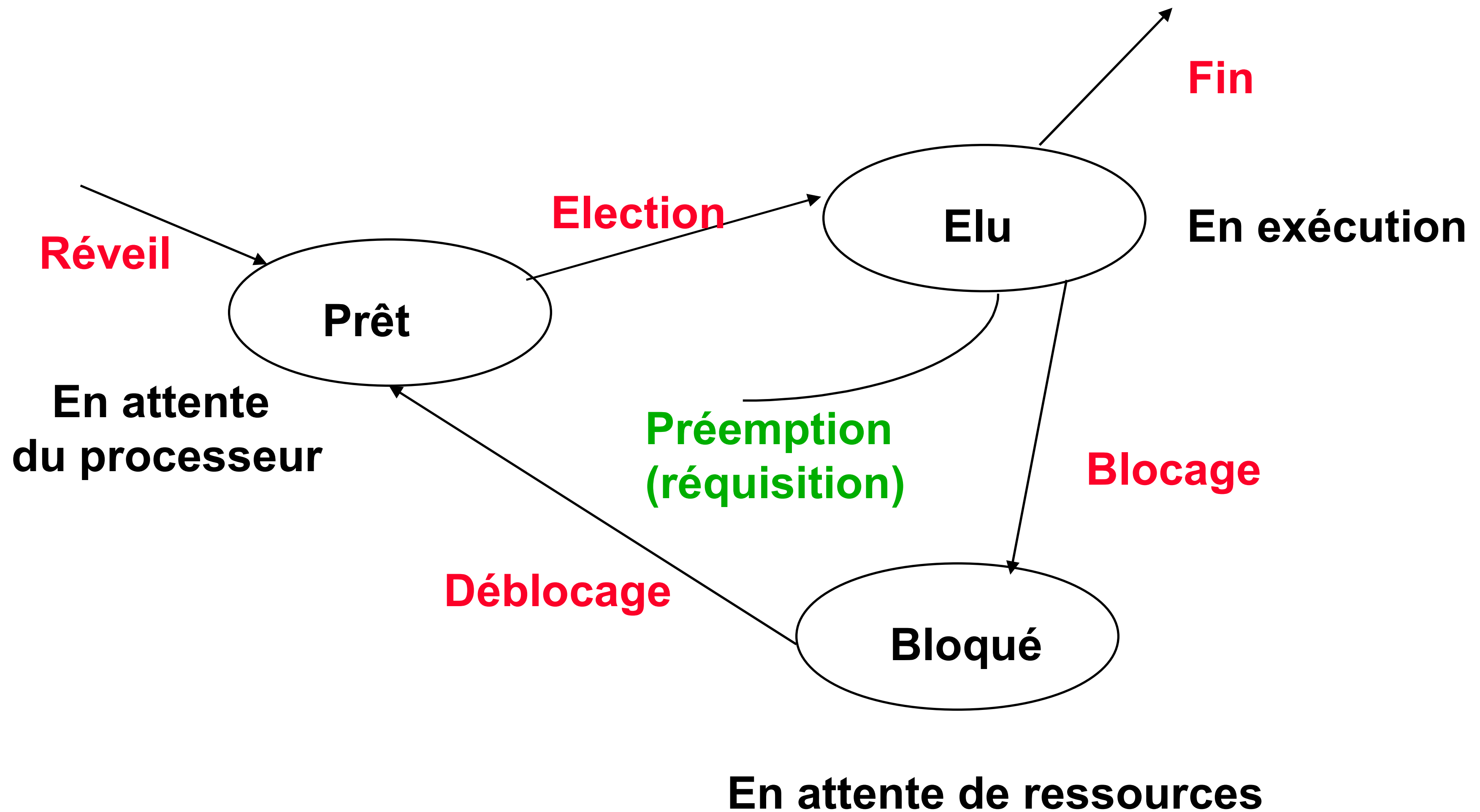


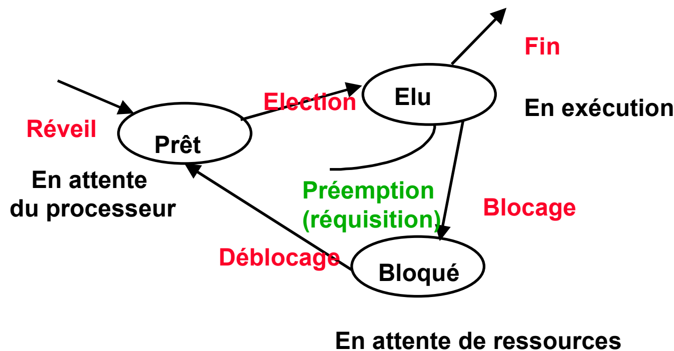


Processeur

Bus

Système multiprocessus
Etats des processus



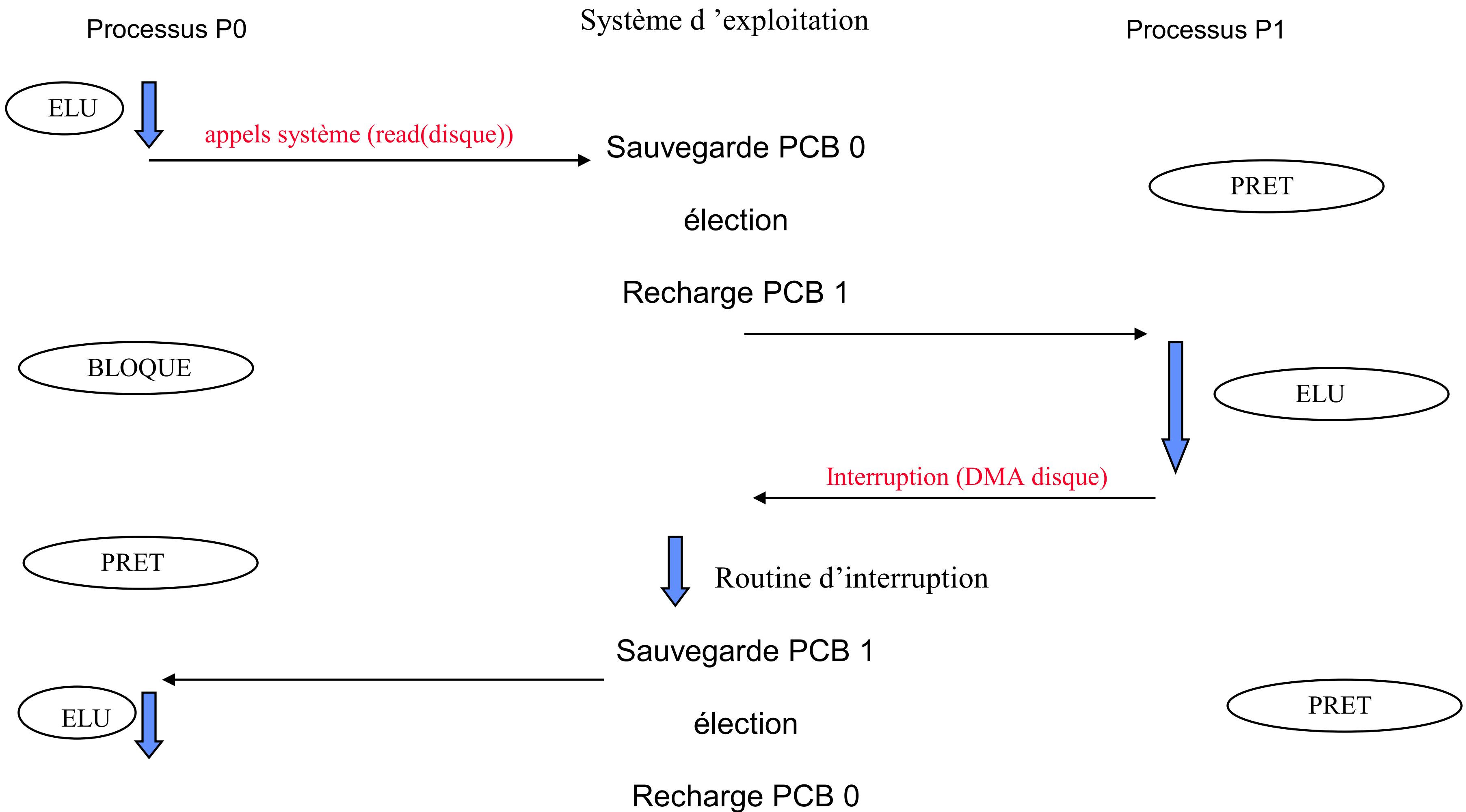


Système multiprocessus
Etats des processus

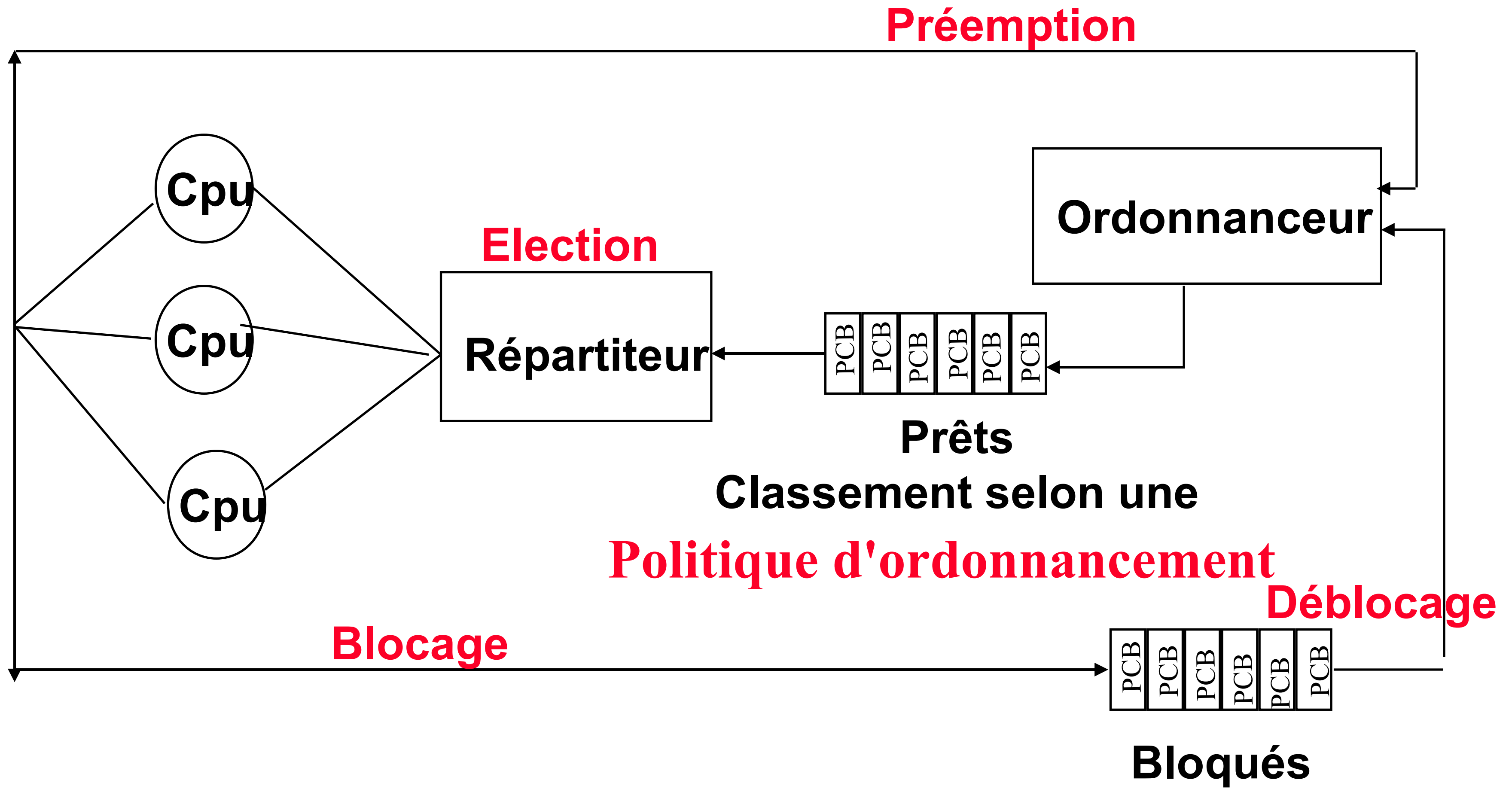
- **Election** : allocation du processeur

- **Préemption** : réquisition du processeur
  - ➢ ordonnancement non préemptif : un processus élu le demeure sauf s 'il se bloque de lui-même
  - ➢ ordonnancement préemptif : un processus élu peut perdre le processeur
    - s 'il se bloque de lui-même (état bloqué)
    - si le processeur est réquisitionné pour un autre processus (état prêt)

Système multiprocessus Ordonnancement



Système multiprocessus
Ordonnanceur et répartiteur

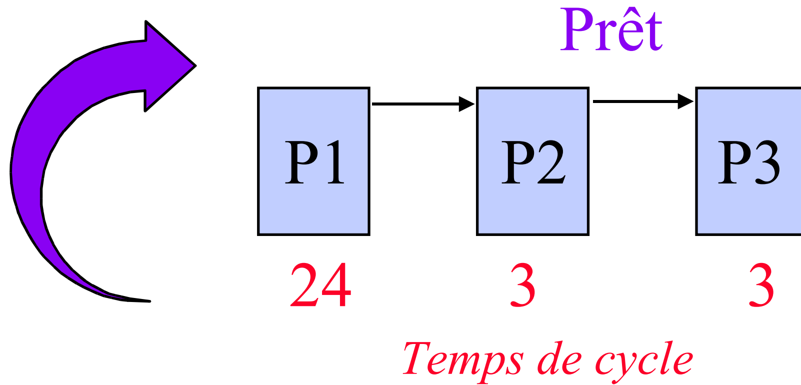


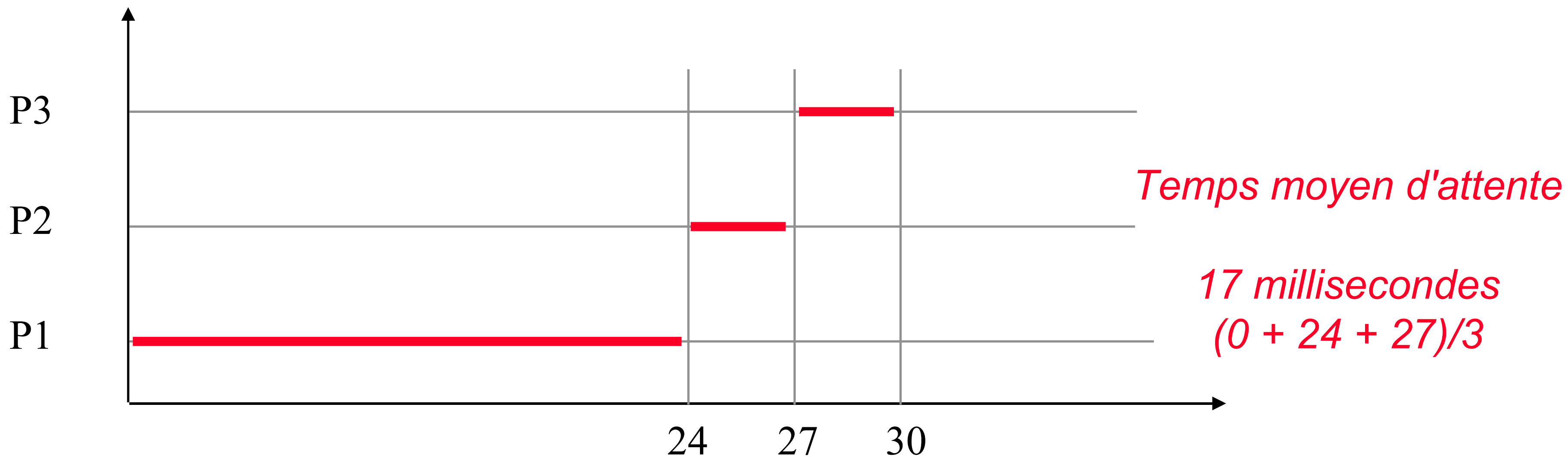
Politiques d'ordonnancement

- Premier arrivé, premier servi
  - FIFO, sans réquisition
- Par priorités constantes
- Par tourniquet (round robin)
- Par files de priorités de priorités constantes multiniveaux avec ou sans extinction de priorité

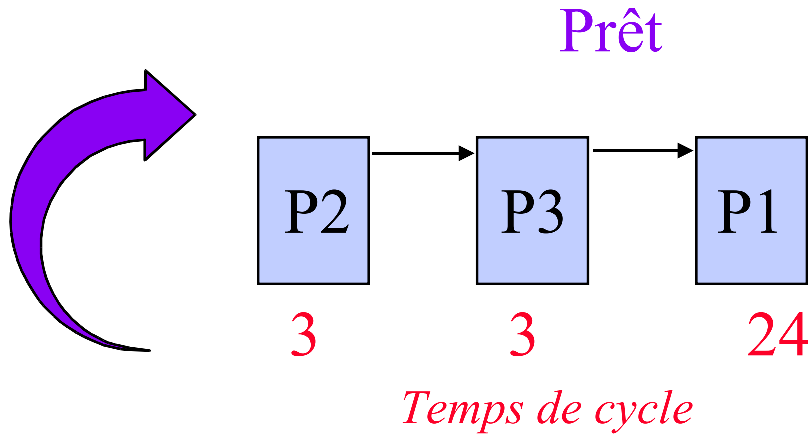
Algorithme : Premier Arrivé Premier Servi

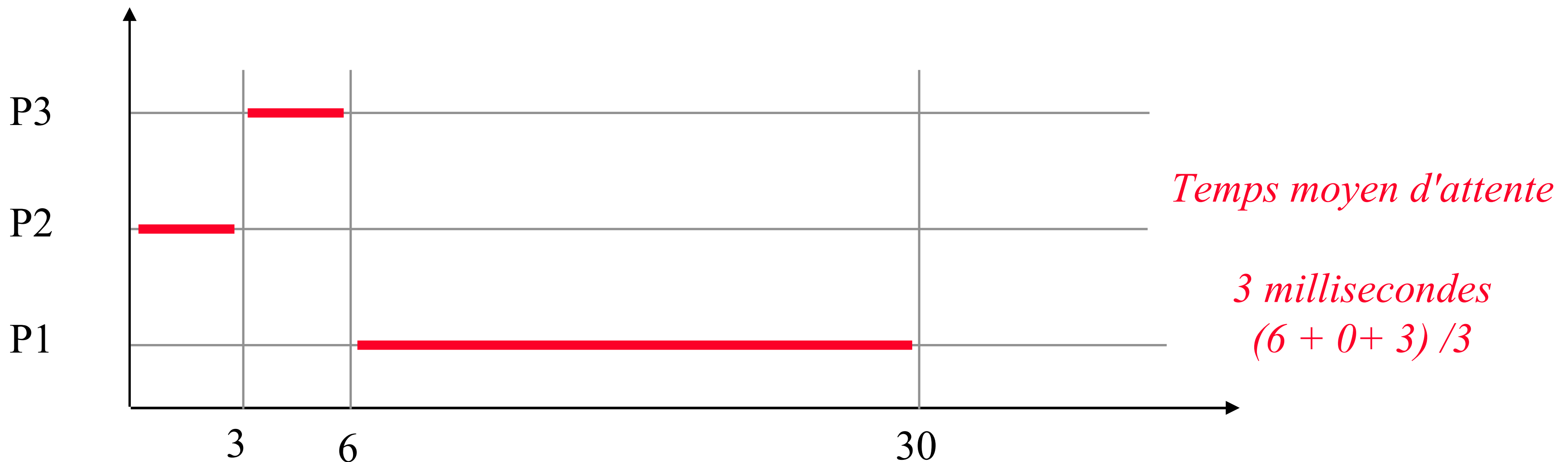
- **FIFO, sans réquisition**





Algorithme : Premier Arrivé Premier Servi

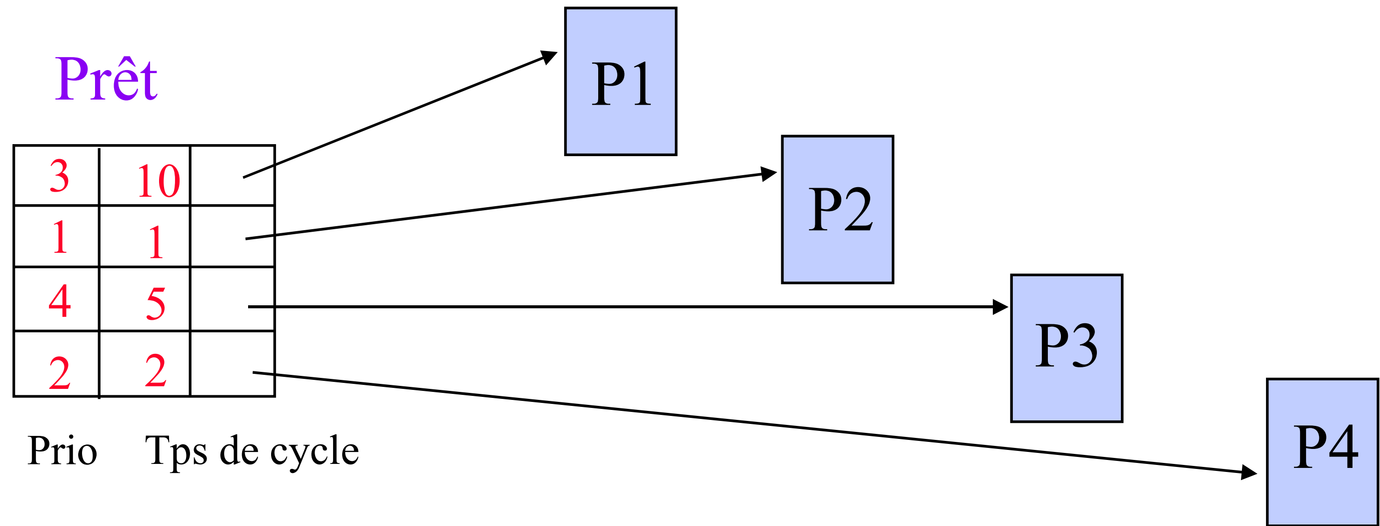




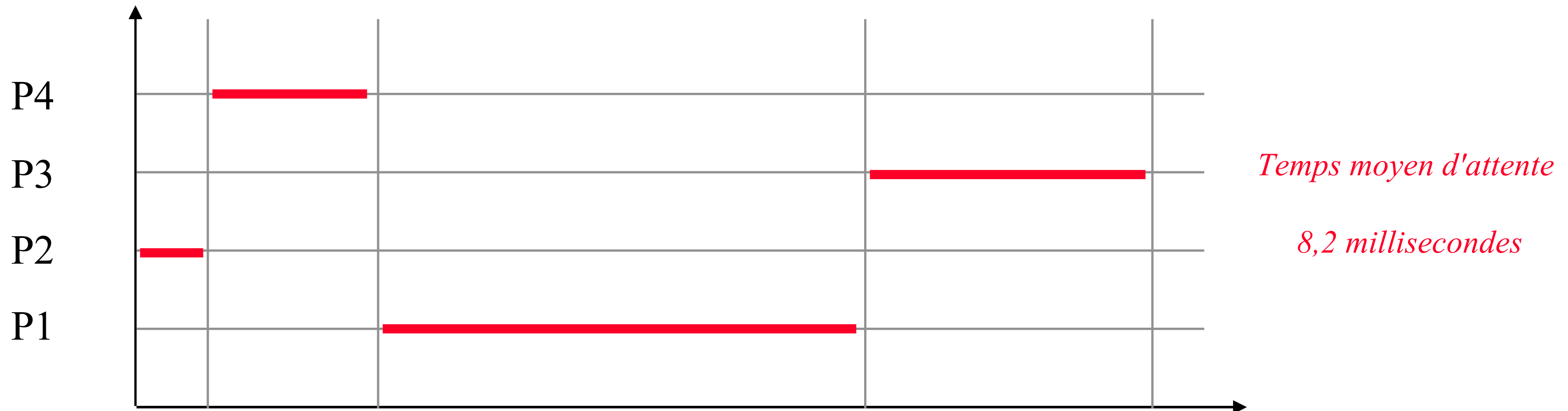
Politiques d'ordonnancement

- Premier arrivé, premier servi
- Par priorités constantes
  - chaque processus reçoit une priorité
  - le processus de plus forte priorité est élu
  - Avec ou sans réquisition
- Par tourniquet (round robin)
- Par files de priorités de priorités constantes multiniveaux avec ou sans extinction de priorité

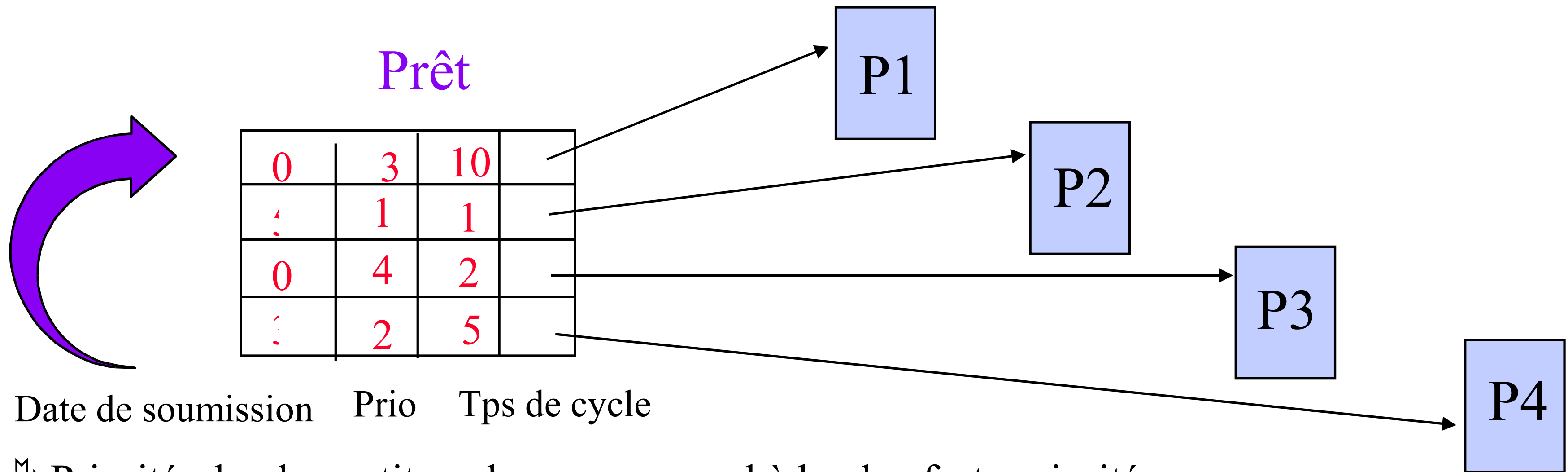
Algorithme : avec priorités



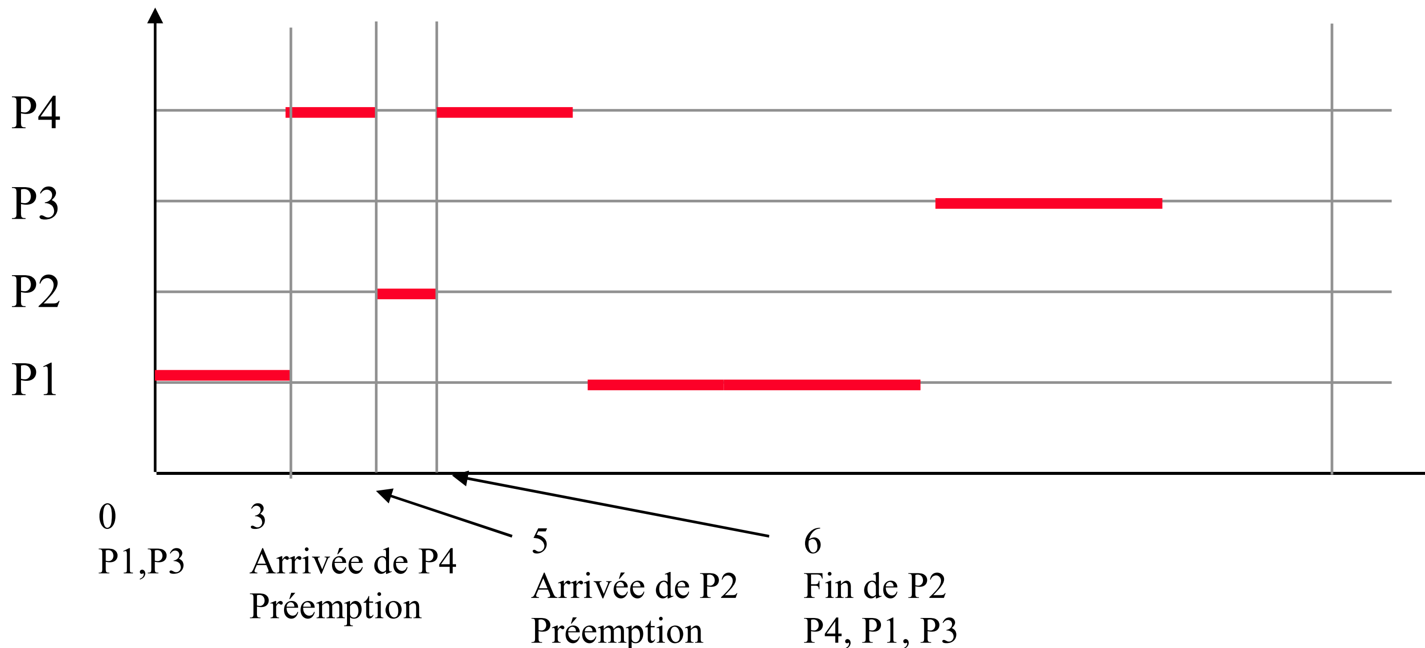
↳Priorité : la plus petite valeur correspond à la plus forte priorité



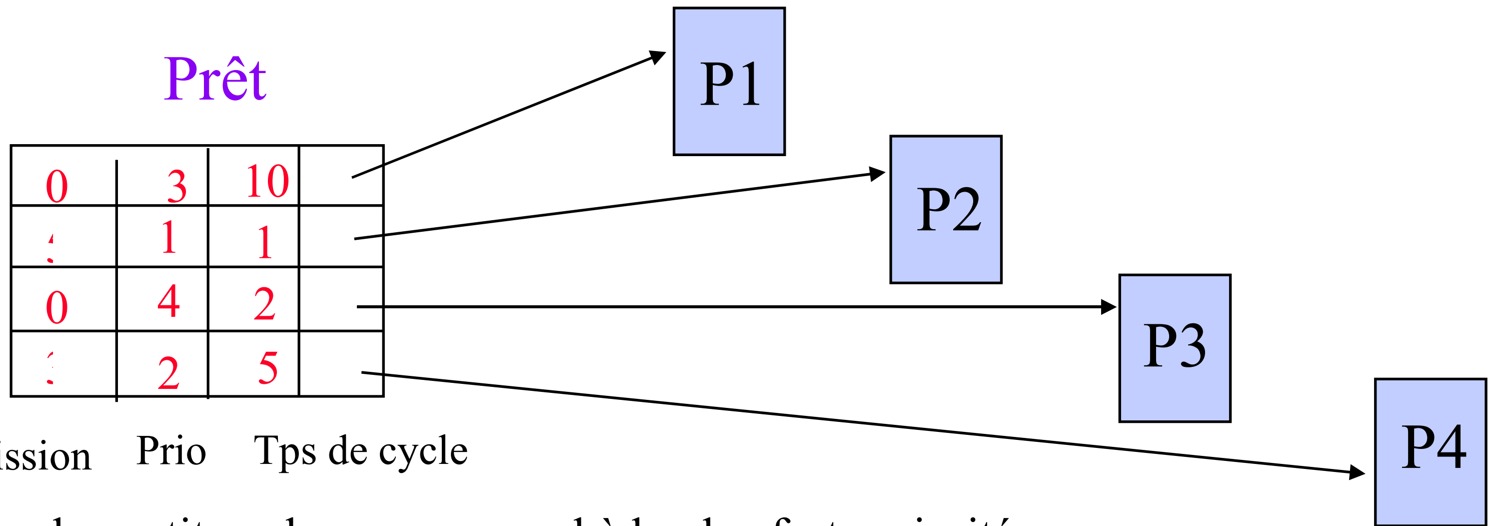
Algorithme : avec priorités préemptif



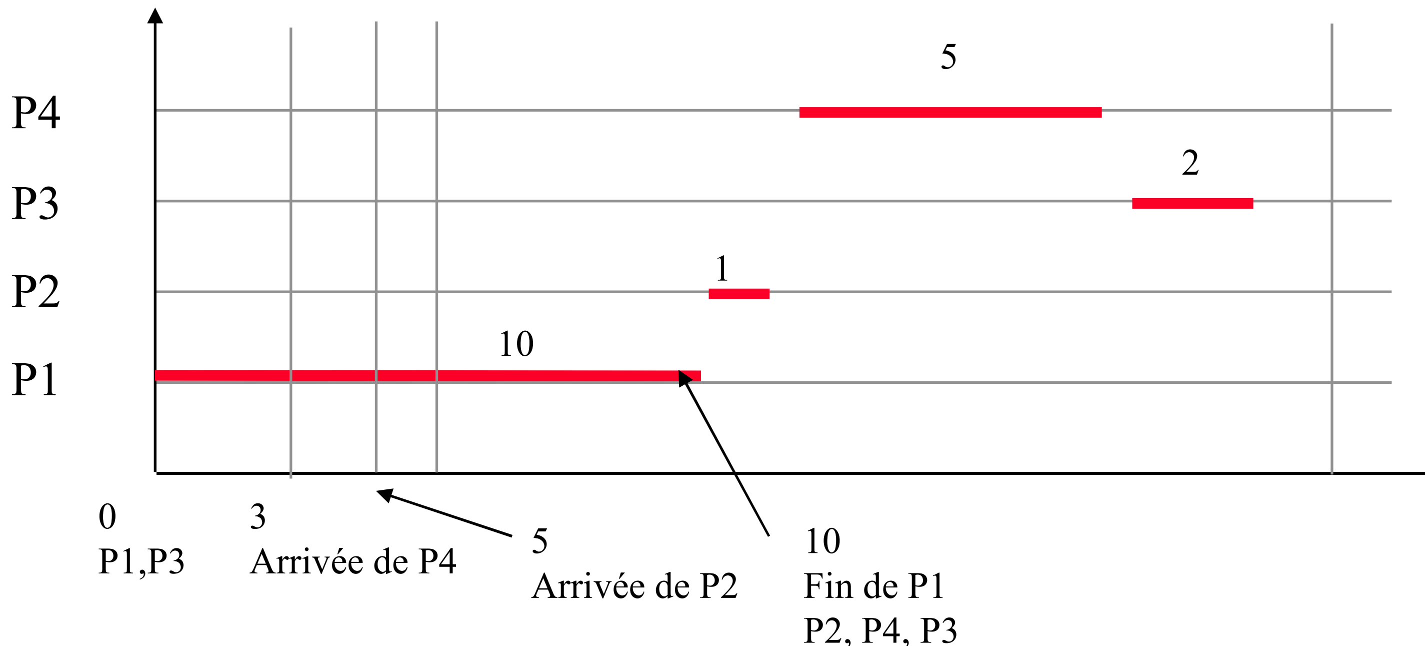
- Priorité : la plus petite valeur correspond à la plus forte priorité



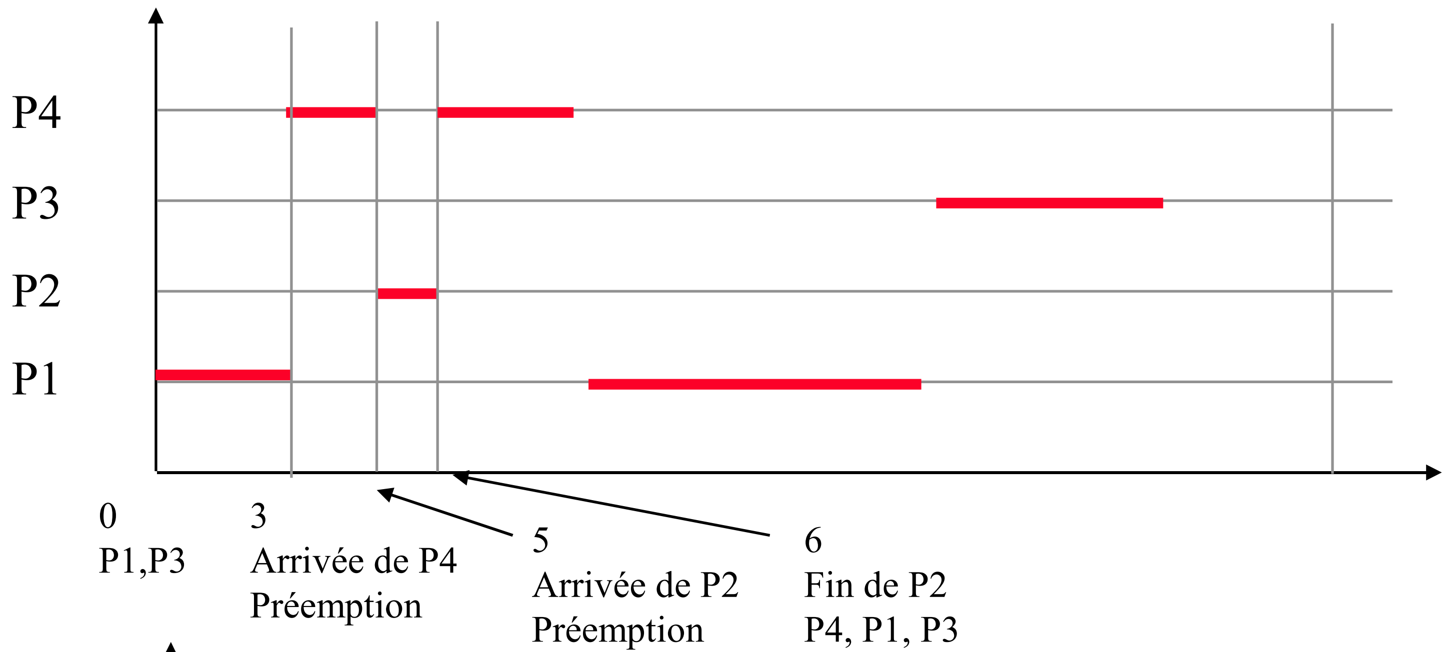
Algorithme : avec priorités non préemptif



↳Priorité : la plus petite valeur correspond à la plus forte priorité

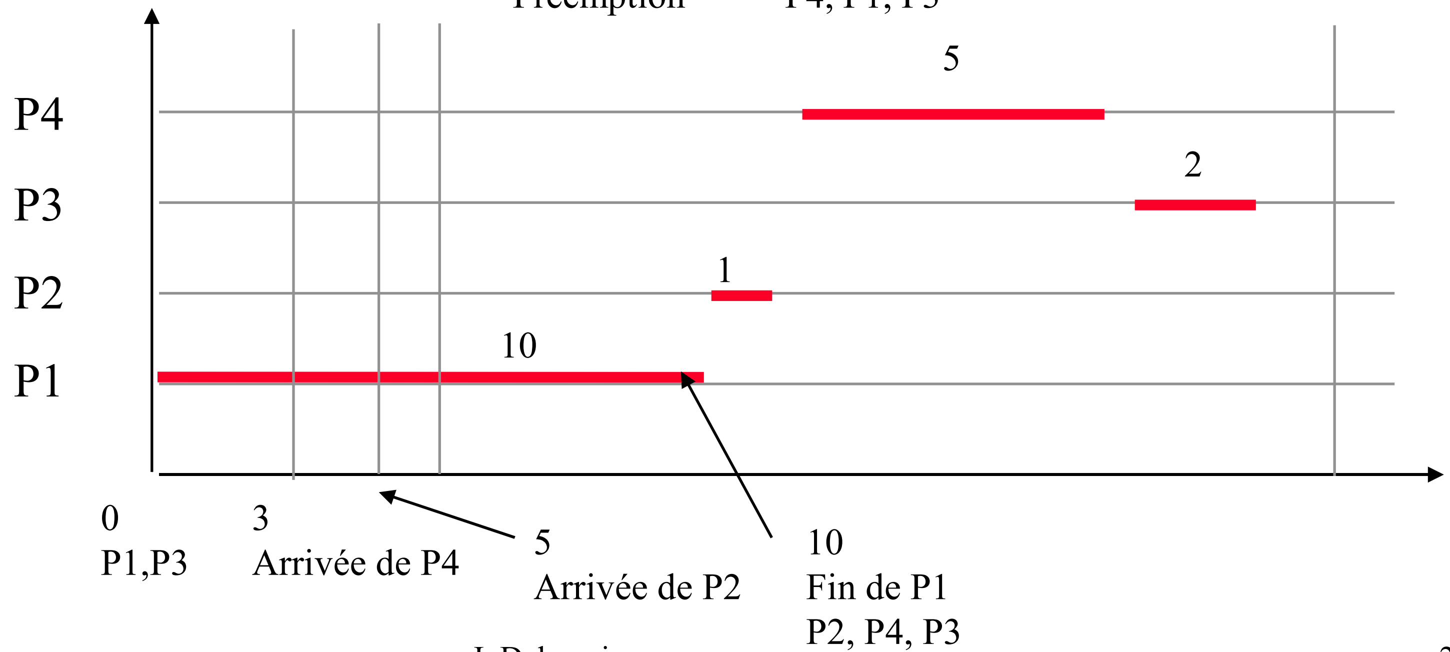


P
R
E
E
M
P
T
I
F



N
O
N

P
R
E
E
M
P
T
I
F



Politiques d'ordonnancement

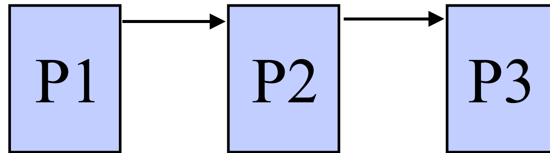
- **Premier arrivé, premier servi**
- **Par priorités constantes**

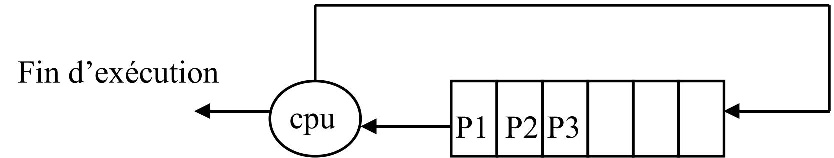
- **Par tourniquet (round robin)**
  - **Définition d'un quantum = tranche de temps**
  - **Un processus élu s'exécute au plus durant un quantum; à la fin du quantum, préemption et réinsertion en fin de file d'attente des processus prêts**

- **Par files de priorités de priorités constantes multiniveaux avec ou sans extinction de priorité**

Algorithme : tourniquet

Prêt

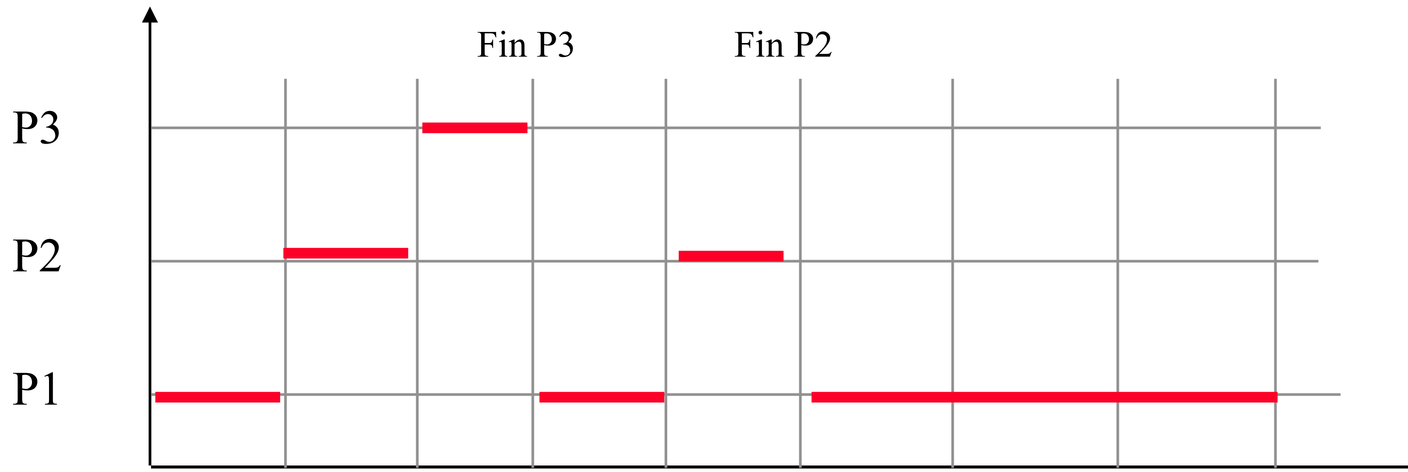




Quantum = 4

20 7 3

Temps de cycle



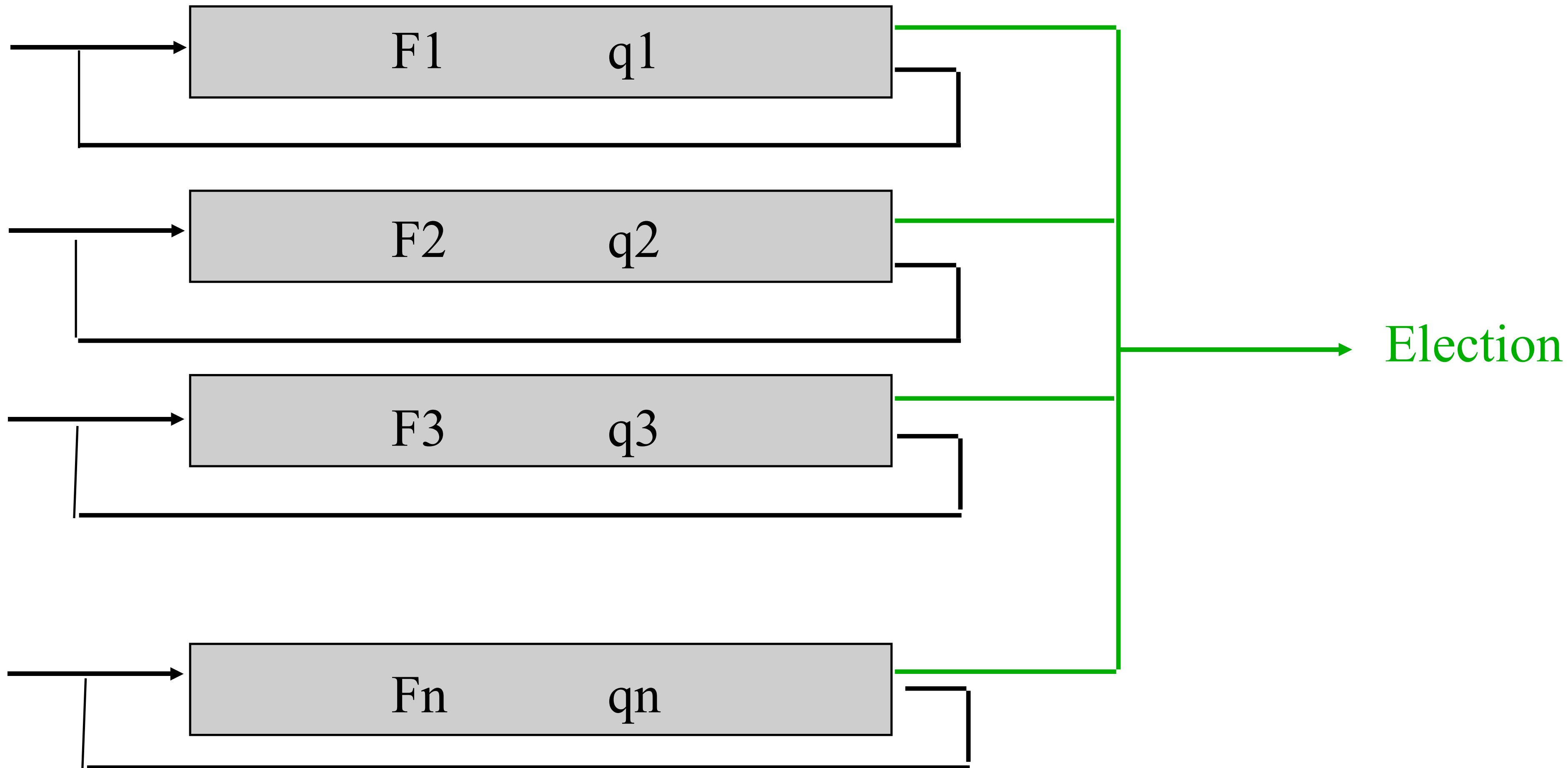
P1	P2	P3	P1	P2	P1
P2	P3	P1	P2	P1	
P3	P1	P2			

Politiques d'ordonnancement

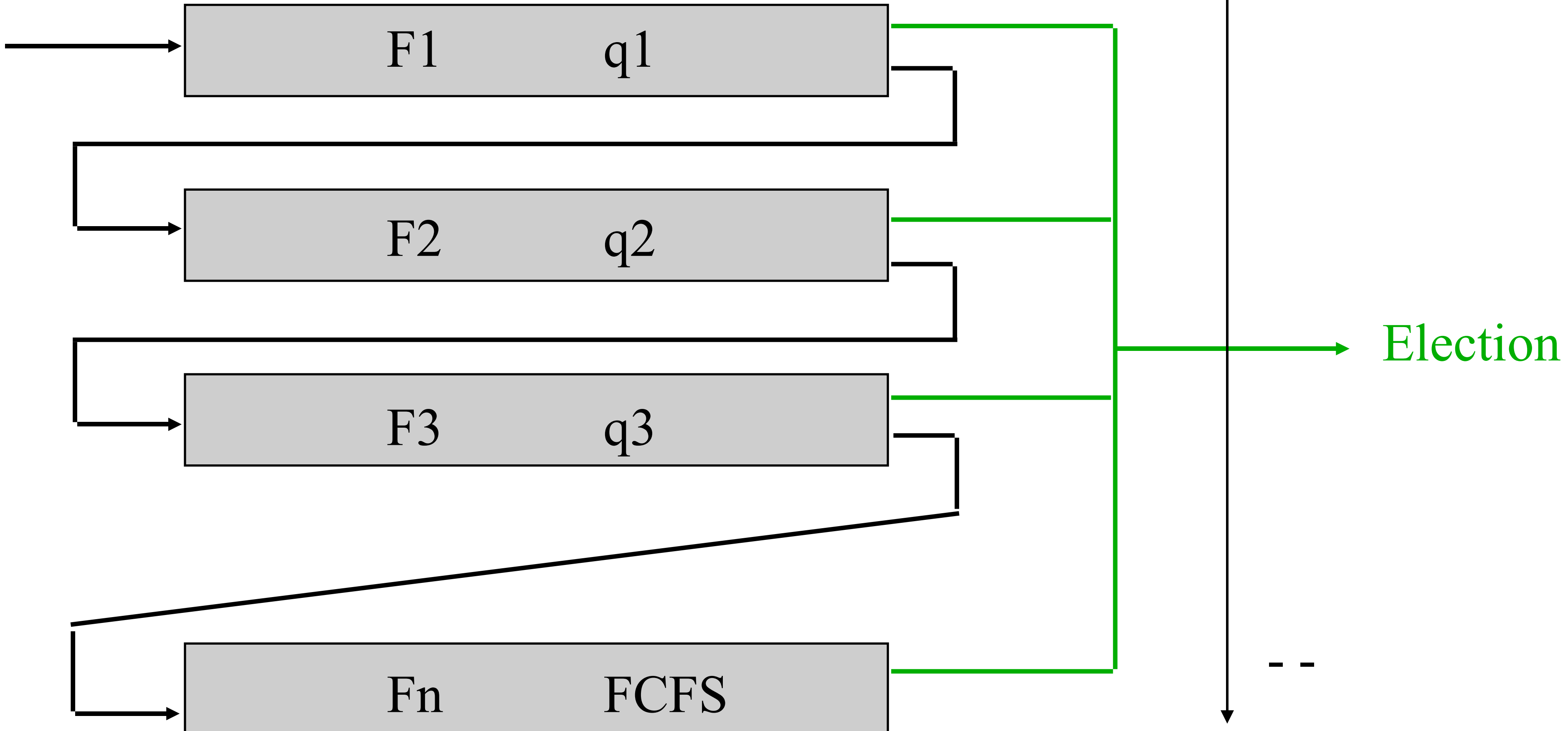
- Premier arrivé, premier servi
- Par priorités constantes
- Par tourniquet (round robin)

- Par files de priorités de priorités constantes multiniveaux avec ou sans extinction de priorité
  - chaque file est associée à un quantum éventuellement différent
  - sans extinction : un processus garde toujours la même priorité
  - avec extinction : la priorité d'un processus décroit en fonction de son utilisation de la cpu

Algorithme : multifiles sans extinction



Algorithme : multifiles avec extinction



Politiques d'ordonnancement

Politique d’ordonnancement	Commentaire
FIFO	Pénalise les processus de court temps d’exécution
Priorité	Risque de famine des processus de faible priorité
Tourniquet	Valeur de quantum par rapport au coût des commutations de contexte

Ordonnancement dans le système LINUX

Système multiprocessus
Ordonnancement LINUX

Le système Linux est un gestionnaire de processus.

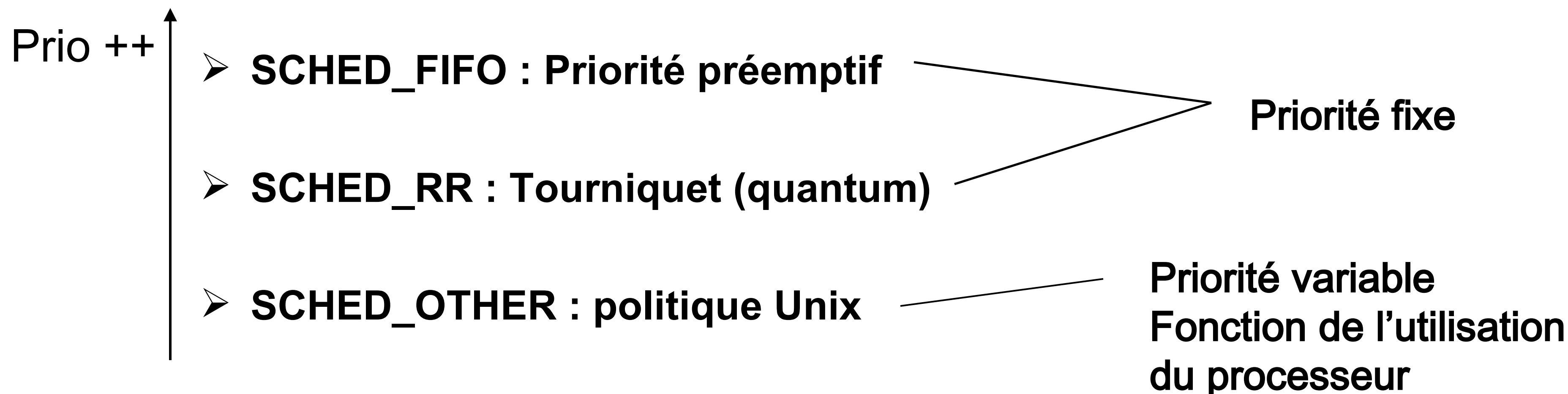
Il offre des services aux processus

Il ne comporte pas à proprement parler de processus qui exécutent son code.
Ce sont les processus utilisateurs qui en passant en mode noyau exécutent le code du système

L'ordonnancement est lancé à chaque fois qu'un processus utilisateur s'apprête à repasser en mode utilisateur depuis le mode noyau et la variable noyau need_resched = true.

Ordonnancement : système LINUX

- **Trois classes d 'ordonnancement (norme POSIX) :**

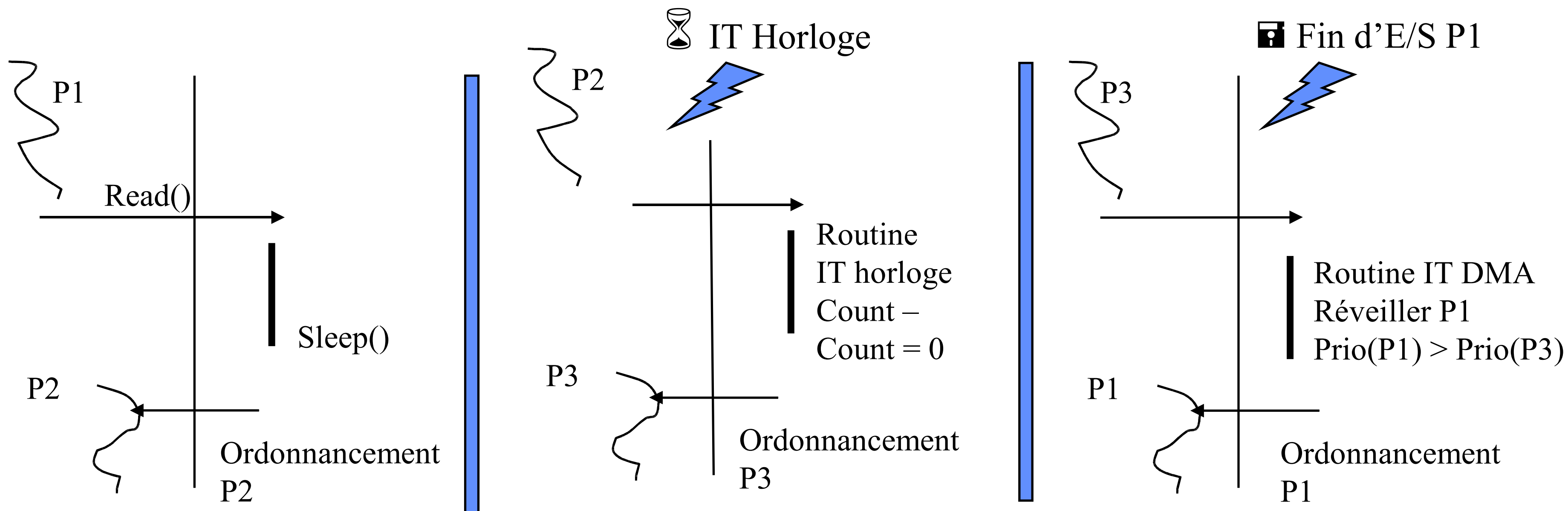


A l'instant t, le système élit (fonction GOODNESS du noyau)
-Le processus SCHED_FIFO de plus forte priorité qui s'exécute jusqu'à sa fin ou jusqu'à préemption par un processus FIFO plus prioritaire
-Le processus SCHED_RR de plus forte priorité pour un quantum
-Le processus SCHED_OTHER de plus forte priorité

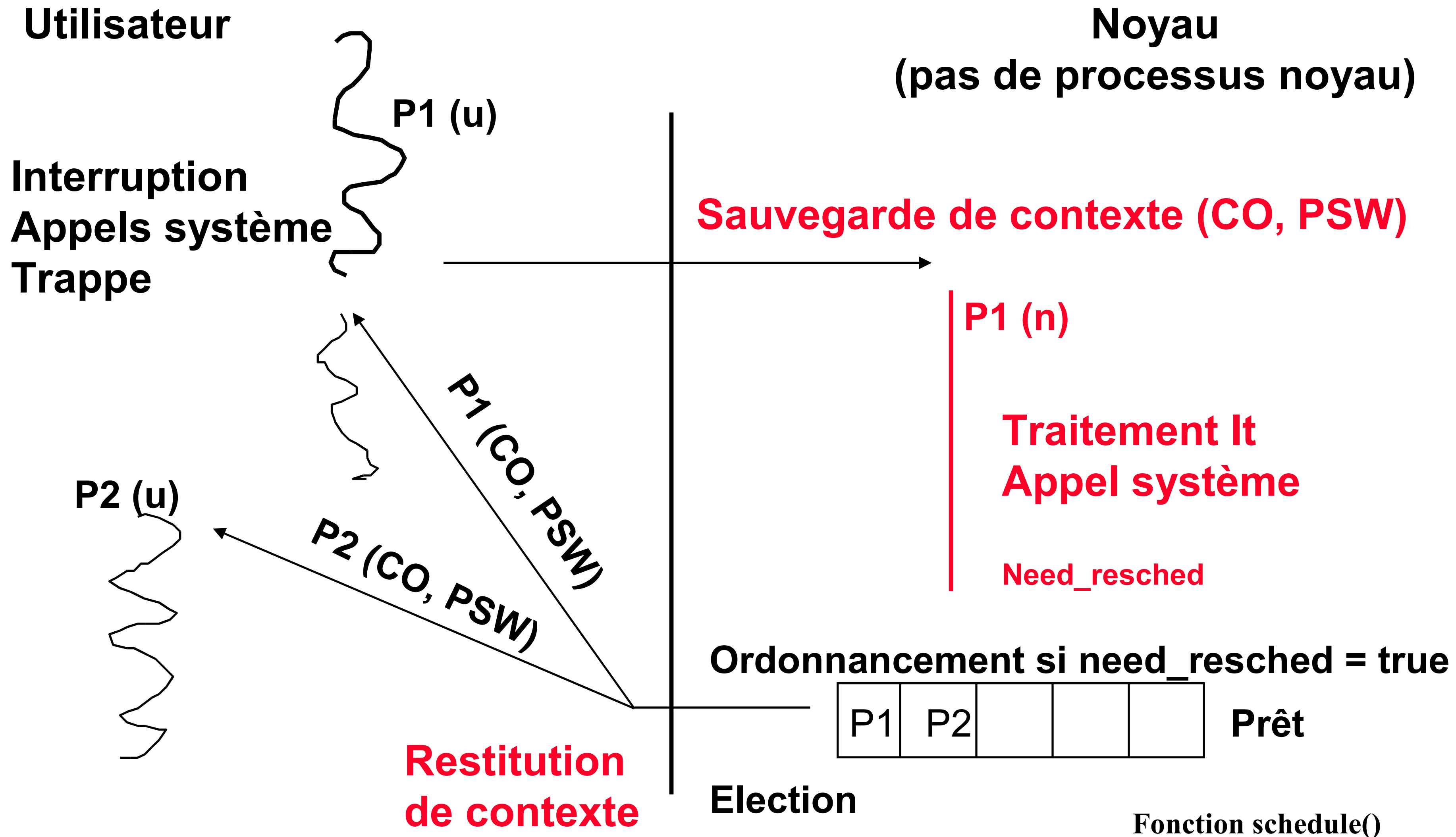
Système multiprocessus
Ordonnancement LINUX

Positionnement de need_resched :

- Un processus passe en mode bloqué ;
- Un processus a épuisé son quantum;
- Un processus temps réel plus prioritaire est réveillé.



Ordonnancement dans le système LINUX



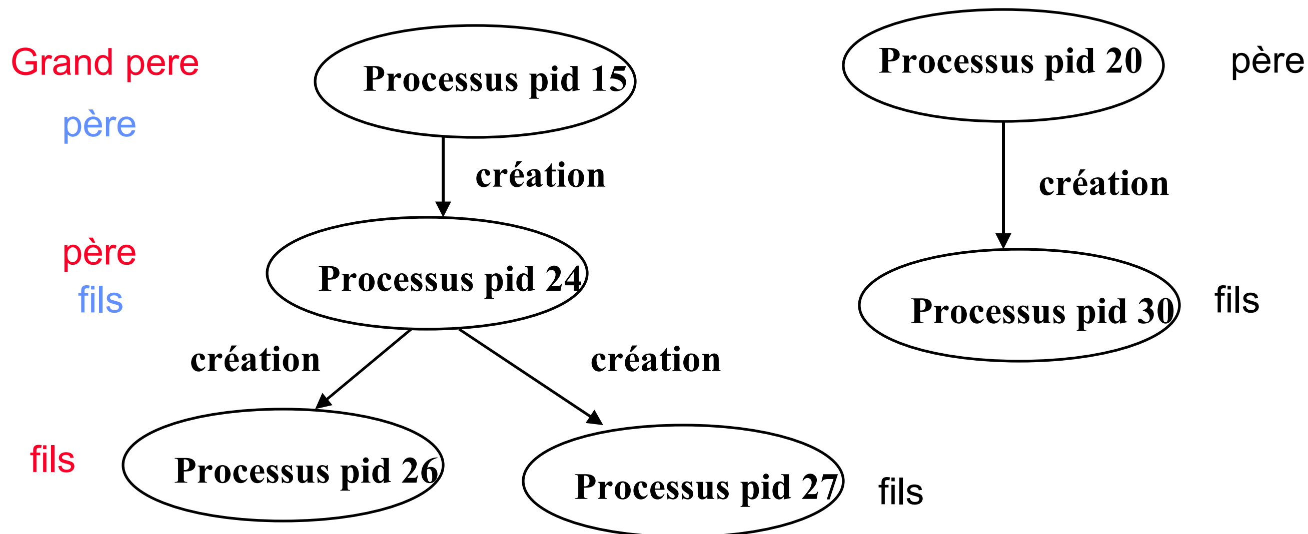
Processus
Processus LINUX

Caractéristiques Générales

Un processus Unix/Linux est identifié par un numéro unique, le PID.

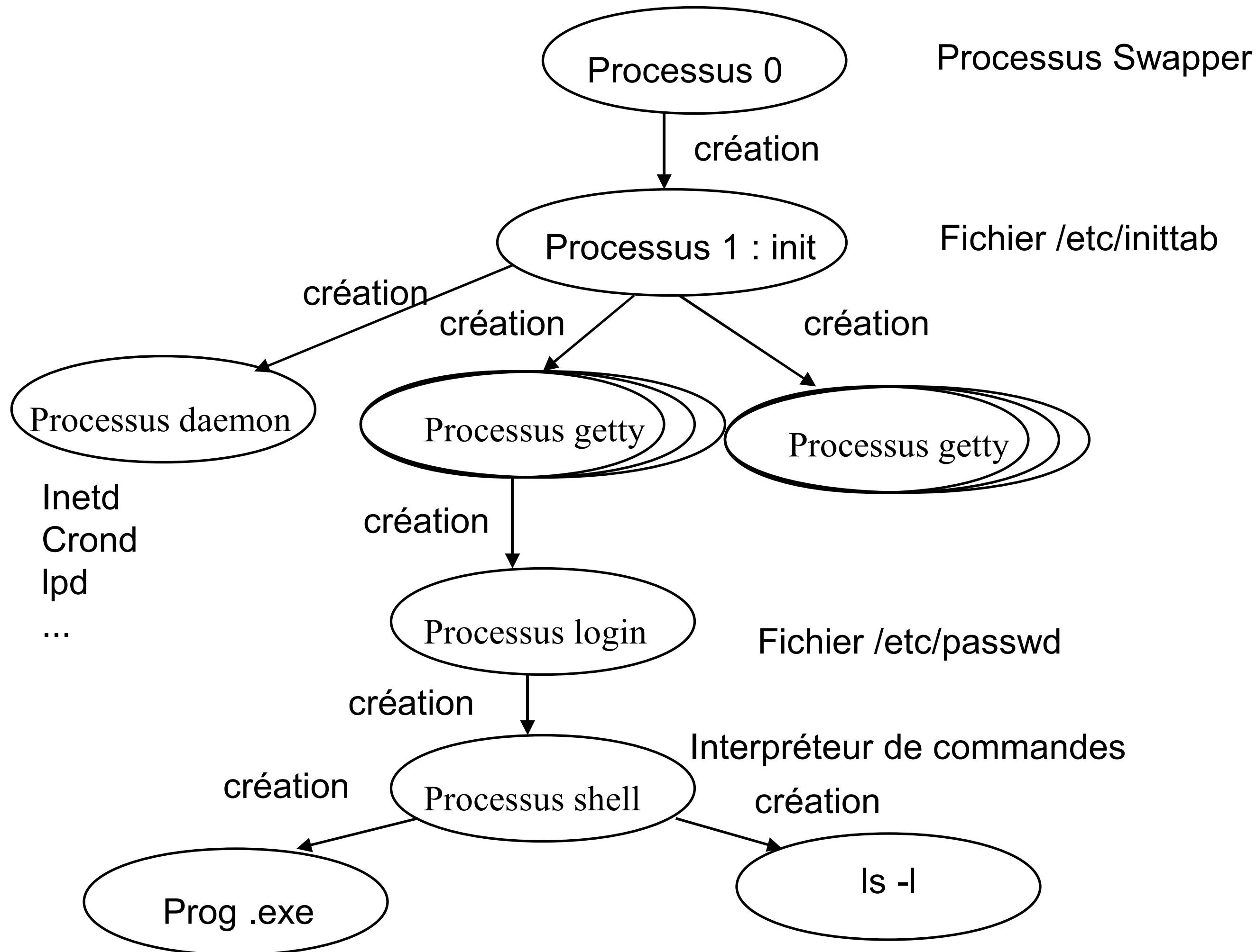
Processus Unix / Linux

- **Tout processus Linux peut créer un autre processus Linux**
  - **Arborescence de processus avec un rapport père - fils entre processus créateur et processus crée**

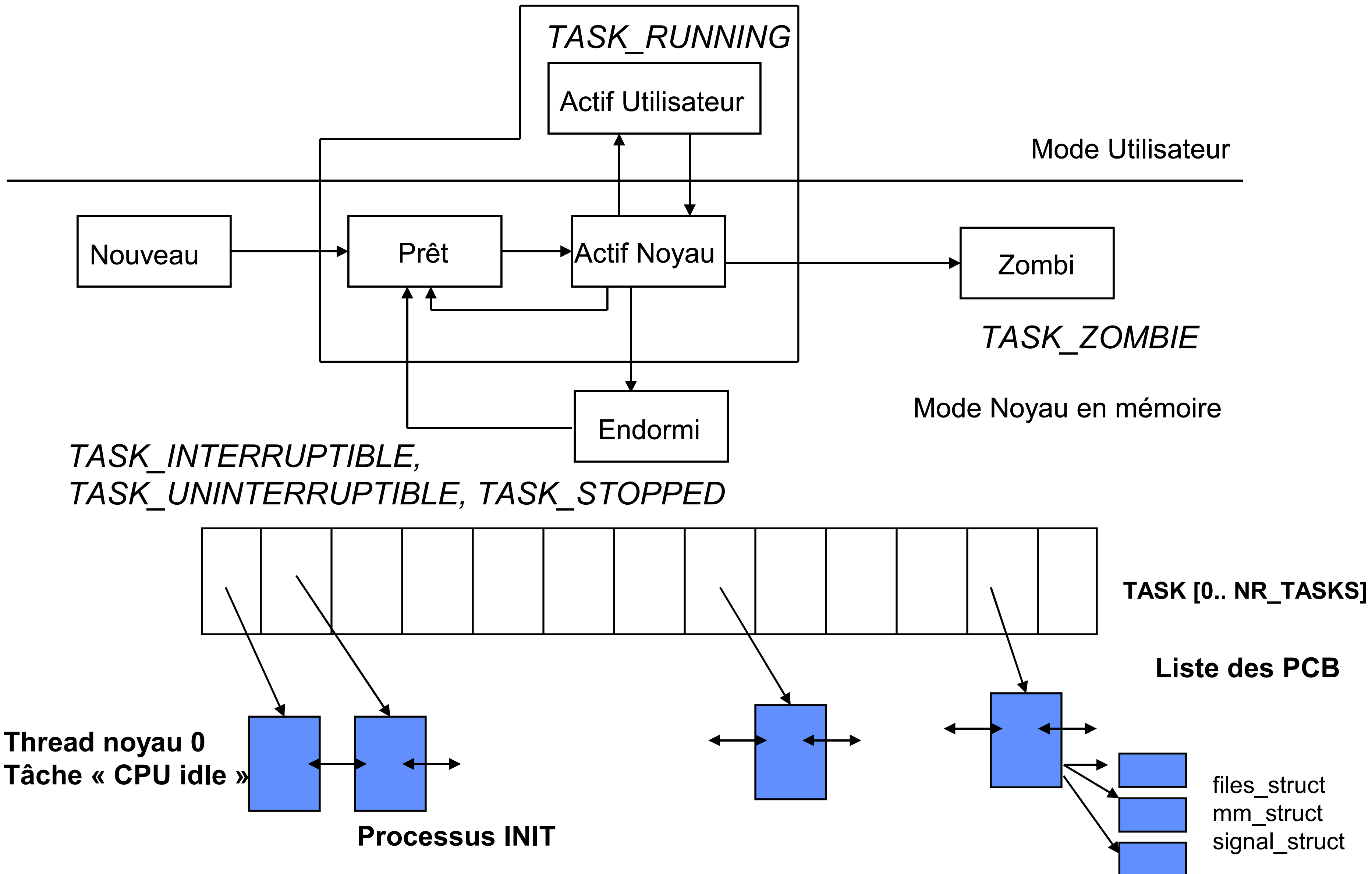


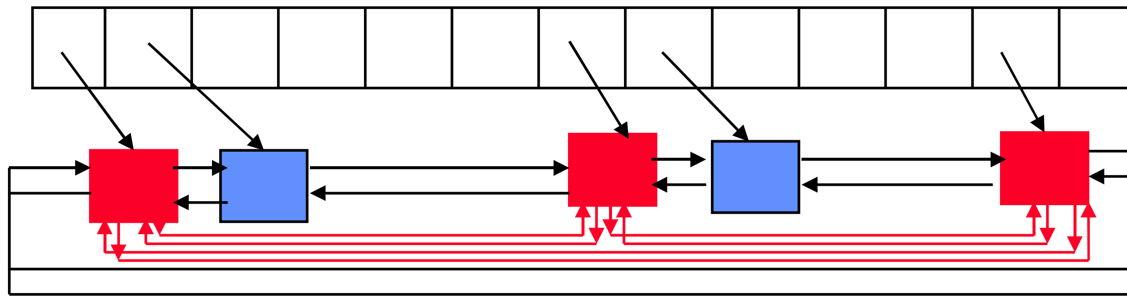
Processus Unix

- **Tout le système Unix repose sur ce concept arborescent**



Processus Linux





Liste des PCB RUNNING
(run_queue) *next_run, *prev_run;

Liste des PCB *next_task, *prev_task

TASK_STRUCT

volatile long state; - - *état du processus*
long counter; - - *quantum*
long priority; -- *priorité SCHED_OTHER*
struct task_struct *next_task, *prev_task; -- *chainage PCB*
struct task_struct *next_run, *prev_run; -- *chainage PCB Prêt*
int pid; -- *pid du processus*
struct task_struct *p_opptr, *p_pptr, *p_cptr,; -- *pointeurs PCB père originel, père actuel, fils*
long need_resched; -- *ordonnancement requis ou pas*
long utime, stime, cutime, cstime;
-- temps en mode user, noyau, temps des fils en mode user, noyau
unsigned long policy; -- *politique ordonnancement SCHED_RR, SCHED-FIFO, SCHED_OTHER*
unsigned rt_priority; -- *priorité SCHED_RR et SCHED_FIFO*
struct thread_struct tss; -- *valeurs des registres du processeur*
struct mm_struct *mm; -- *contexte mémoire*
*struct files_struct *files ; -- table fichiers ouverts*
*struct signal_struct *sig ; -- table de gestion des signaux*

Primitives et commandes générales

Primitives et commandes générales

- **Primitive getpid, getppid**

```
#include <unistd.h>

pid_t getpid(void)
```

retourne le pid du processus appelant

```
pid_t getppid(void)
```

retourne le pid du père du processus appelant

Primitives et commandes générales

- **Commande ps**

  - **délivre la liste des processus avec leur caractéristiques (pid, ppid, état, terminal, durée d 'exécution, commande associée…)**

```
     % ps
     PID    TTY     S      TIME     CMD
20841 ttyq0   S       0:00.27 -csh (csh)
20844 ttyq0   R       0:00.00 -ps (ps)

     %ps ax
     PID TTY     S         TIME     CMD
      0   ??          R       14:30:59 [kernel idle]
      1   ??          S       13:43.28 /sbin/init -a
      683 ??        S         0:00.49 telnetd
     777 ??         S        0:06.30 /usr/lbin/lpd
      1177 ??      S          5:13.60 /usr/sbin/inetd
```

Primitives et commandes générales

- **Commande kill**

  - **délivre un signal à un processus de numéro pid**

```
% ps
PID    TTY     S      TIME     CMD
20841 ttyq0   S      0:00.27 -csh (csh)
20847 ttyq0   S      0:00.45  essai
20844 ttyq0   R      0:00.00 -ps (ps)

%kill -9 20847 (signal n°9 (SIGKILL))
% ps
PID    TTY     S      TIME     CMD
20841 ttyq0   S      0:00.27 -csh (csh)
20844 ttyq0   R      0:00.00 -ps (ps)
```

CREATION de PROCESSUS

Primitive de création de processus

- **Primitive fork**

#include <unistd.h>
pid_t fork(void)

> - **La primitive fork() permet la création dynamique d ’un nouveau processus qui s ’exécute de manière concurrente avec le processus qui l ’a créé.**
> - **Tout processus Unix/Linux hormis le processus 0 est crée à l ’aide de cette primitive.**
> - **Le processus créateur (le père) par un appel à la primitive fork() crée un processus fils qui est une copie exacte de lui-même (code et données)**

Primitive de création de processus

MODE UTILISATEUR

PROCESSUS PID 12222

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
          i = i + j;

ret = fork();

}
```

MODE SYSTEME

Exécution de l 'appel système fork

Si les ressources noyau sont disponibles

- allouer une entrée de la table des processus au nouveau processus

- allouer un pid unique au nouveau processus

- dupliquer le contexte du processus parent (code, données, pile)

- retourner le pid du processus crée à son père et 0 au processus fils

Primitive de création de processus

MODE UTILISATEUR

MODE SYSTEME

PROCESSUS PID 12222

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;

ret = fork();

 12223

}
```

PROCESSUS PID 12223

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;

ret = fork();

 0

}
```

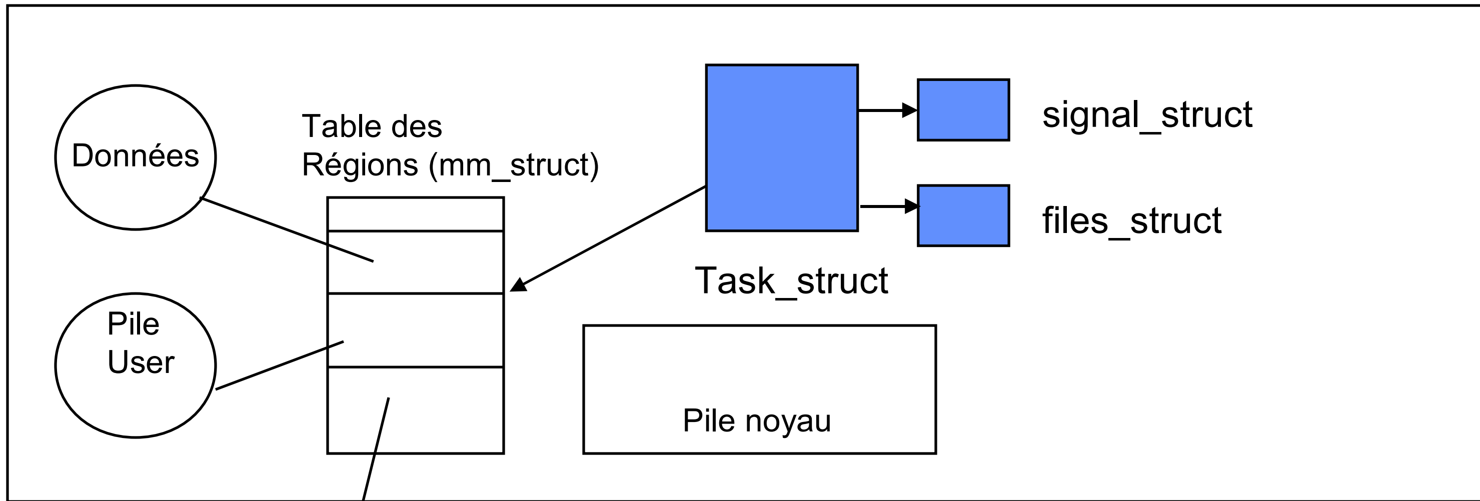
Exécution de l 'appel système fork

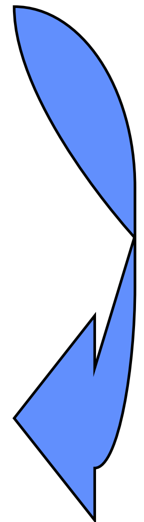
Si les ressources noyau sont disponibles

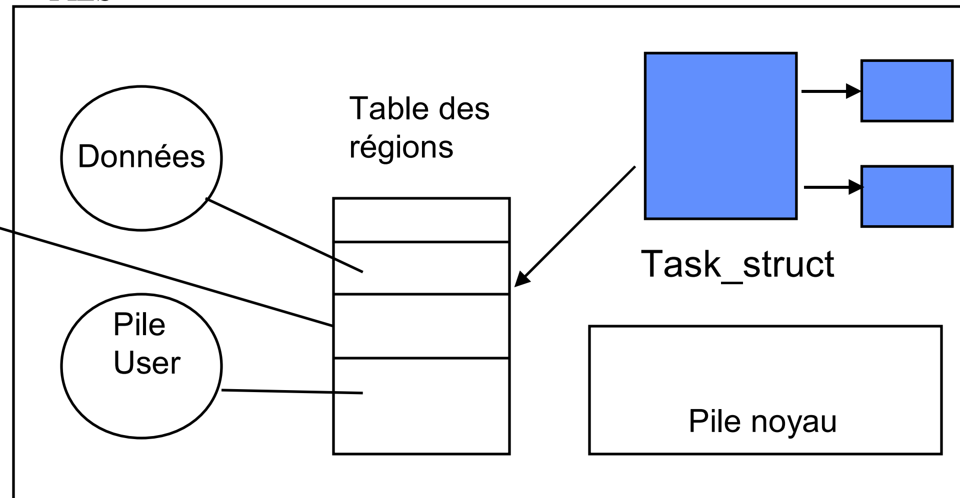
retourner le pid du processus crée à son père

et 0 au processus fils

Primitive de création de processus







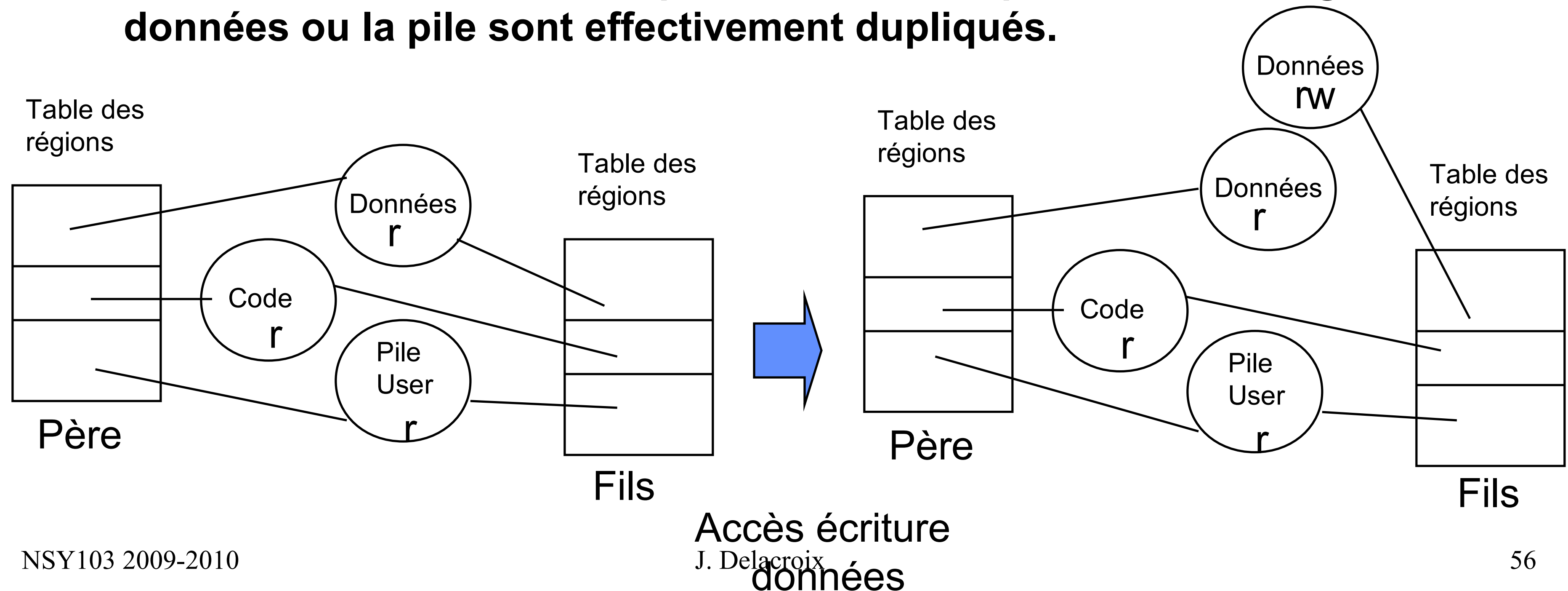
Primitive de création de processus

- **Mécanisme du *copy on write* (Copie sur écriture):**

Lors de sa création, le fils partage avec son père:

- le segment de code;
- le segment de données et la pile sont également partagés et mis en accès lecture seule .

Lors d'un accès en écriture par l'un des deux processus, le segment de données ou la pile sont effectivement dupliqués.



Primitive de création de processus

PROCESSUS
PID 12222

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;

ret = fork();

}
```

PROCESSUS
PID 12223

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;

ret = fork();

}
```

12223

0

- Chaque processus père et fils reprend son exécution après le fork()

- Le code et les données étant strictement identiques, il est nécessaire de disposer d 'un mécanisme pour différencier le comportement des deux processus après le fork()

On utilise pour cela le code retour du fork() qui est différent chez le fils (toujours 0) et le père (pid du fils crée)

Primitive de création de processus

PROCESSUS
PID 12222

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;

ret= fork();

if (ret == 0)
  printf(" je suis le fils ")
else
{
    printf ("je suis le père");
    printf ("pid de mon fils %d", ret)
 }
}
```

Pid du fils : 12223
getpid : 12222
getppid :shell

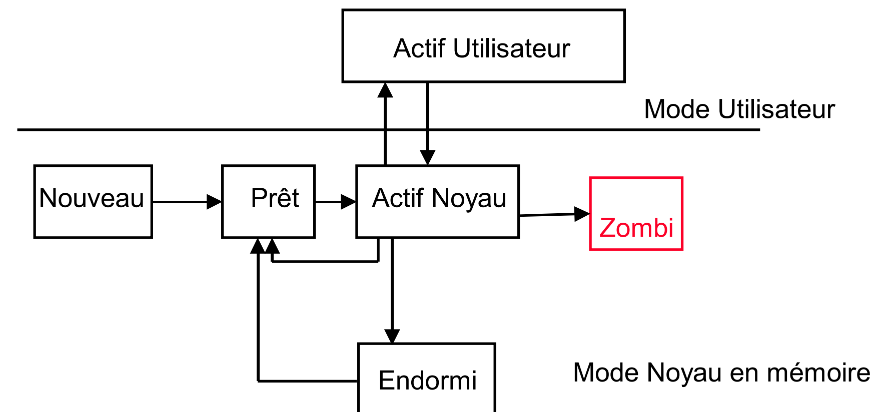
PROCESSUS
PID 12223

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;

ret= fork();

if (ret == 0)
 printf(" je suis le fils ")
else
{
    printf ("je suis le père");
    printf ("pid de mon fils, %d" , ret)
 }
}
```

getpid : 12223
getppid :12222

Synchronisation père / fils

Synchronisation père / fils



- **Primitive exit()**

```
#include <stdlib.h>
void exit (int valeur);
pid_t wait (int *status);
```

- **un appel à la primitive exit() provoque la terminaison du processus effectuant l'appel avec un code retour *valeur*. (Par défaut, le franchissement de la dernière } d'un programme C tient lieu d'exit)**
- **un processus qui se termine passe dans l'état Zombie et reste dans cet état tant que son père n'a pas pris en compte sa terminaison**
- **Le processus père "récupère" la terminaison de ses fils par un appel à la primitive wait ()**

Primitive de création de processus

PROCESSUS
PID 12222

```
Main ()
{
pid_t ret ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;
ret= fork();
if (ret == 0)
{
      printf(" je suis le fils ");
      exit(); }
else
{
    printf ("je suis le père");
    printf ("pid de mon fils, %d" , ret)
    wait();
 }

}
```

PROCESSUS
PID 12223

```
Main ()
{
pid_t ret;
int i, j;

for(i=0; i<8; i++)
        i = i + j;
ret= fork();

if (ret== 0)
{
      printf(" je suis le fils ");
      exit; }
else
{
    printf ("je suis le père");
    printf ("pid de mon fils, %d" ,ret);
    wait();
 }

}
```

Synchronisation père / fils

- **Lorsqu'un processus se termine (exit), le système démantèle tout son contexte, sauf l'entrée de la table des processus le concernant.**

- **Le processus père, par un wait(), "récupère" la mort de son fils, cumule les statistiques de celui-ci avec les siennes et détruit l'entrée de la table des processus concernant son fils défunt.**

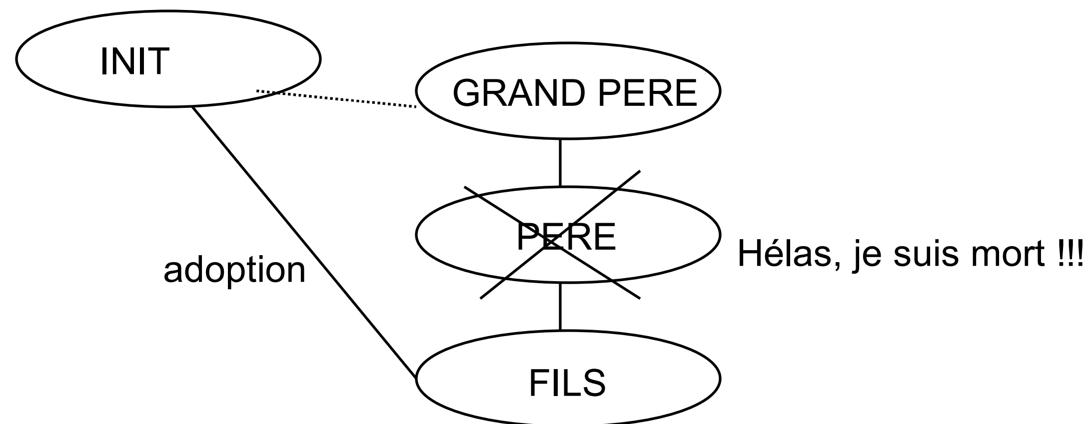
  Le processus fils disparaît complètement.

- **La communication entre le fils zombie et le père s'effectue par le biais d'un signal transmis du fils vers le père (signal SIGCHLD ou mort du fils)**

- **ATTENTION : mauvaise synchronisation = saturation table des processus = blocage du système**

Génétique des processus Unix

- **Decès et adoption**
  - **Un processus fils défunt reste zombie jusqu'à ce que son père ait pris connaissance de sa mort**
  - **Un processus fils orphelin, suite au décès de son père (le processus père s'est terminé avant son fils) est toujours adopté par le processus 1 (init).**



Primitives de recouvrement

Il s'agit d'un ensemble de primitives (famille exec) permettant
à un processus de charger en mémoire, un nouveau code exécutable
(execl, execlp, execle, execv, execvp, execve).

Primitives de recouvrement

- **int main (int argc, char *argv[], char *arge[]);**
  - **argc est le nombre de composants de la commande**
  - **argv est un tableau de pointeurs de caractères donnant accès aux différentes composantes de la commande**
  - **arge est un tableau de pointeurs de caractères donnant accès à l'environnement du processus.**

Sh ---> fork puis exec(calcul, 3, 4)

% calcul 3 4

on a argc = 3,

argv[0]="calcul",argv[1]="3"

et argv[2] = "4"

```
Calcul.c
Main(argc,argv)
{
int somme;
if (argc <> 3) {printf("erreur"); exit();}
somme = atoi(argv[1]) + atoi(argv[2]);
exit();
}
```

ATOI() : conversion caractère --> entier

Primitives de recouvrement (execl)

```
#include <sys/types.h>
#include <sys/wait.h>
int execl(const char *ref, const char *arg, …, NULL)
```

Chemin absolu du code exécutable

arguments

PROCESSUS

```
Main ()
{
pid_t pid ;
int i, j;

for(i=0; i<8; i++)
        i = i + j;

pid= fork();

if (pid == 0)
{
      printf(" je suis le fils ");
      execl("/home/calcul","calcul","3","4", NULL);
                  executable        paramètres
}
else
{
    printf ("je suis le père");
    printf ("pid de mon fils, %d" , pid);
    wait();
 }

}
```

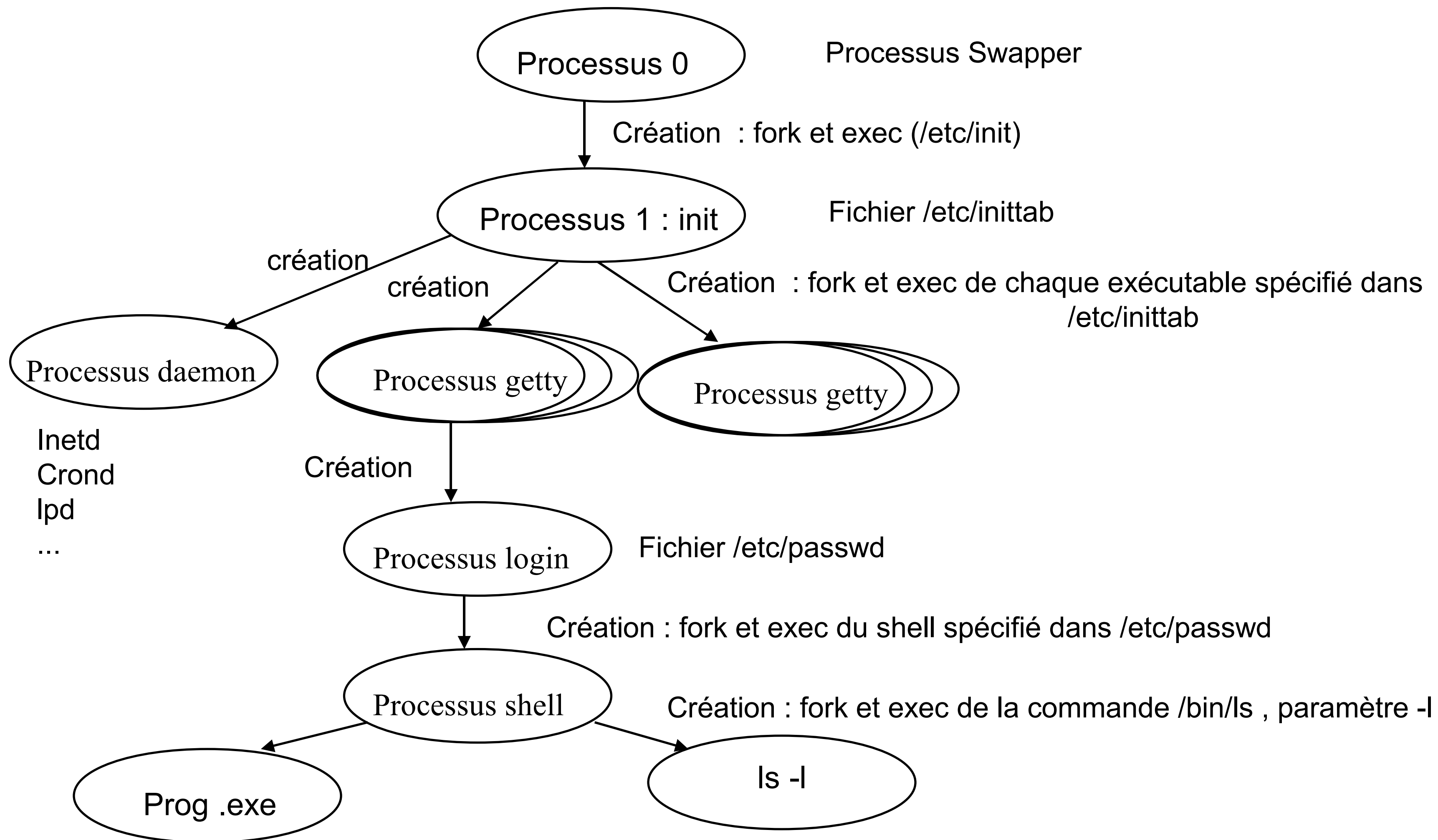
Création d'un processus fils par duplication du code et données du père

Le processus fils recouvre le code et les données hérités du père par ceux du programme calcul. Le père transmet des données de son environnement vers son fils par les paramètres de l'exec

Le père attend son fils

Processus Unix

- **Tout le système Unix repose sur ce concept arborescent**



Primitives de recouvrement

- **Il y a six primitives qui diffèrent**

  - **par la manière dont les arguments argv sont passés**
    - **liste (execl, execlp, execle)**
    - **tableau (execv, execvp, execve)**

  - **par la manière dont le chemin de l'exécutable à charger est spécifié**
    - **en utilisant la variable d'environnement PATH (execlp, execvp)**
    - **relativement au répertoire de travail (les autres)**

  - **par la modification de l'environnement (execve, execle)**

Notion de processus léger (threads, activités)

Notion de processus léger

- **Définition**
  - **Extension du modèle traditionnel de processus**
  - **Un thread ou processus léger est un fil d 'exécution au sein d 'un processus. On peut avoir plusieurs fils d 'exécution au sein du processus.**

Fil d 'exécution	Ressources	Espace d 'adressage	Processus classique

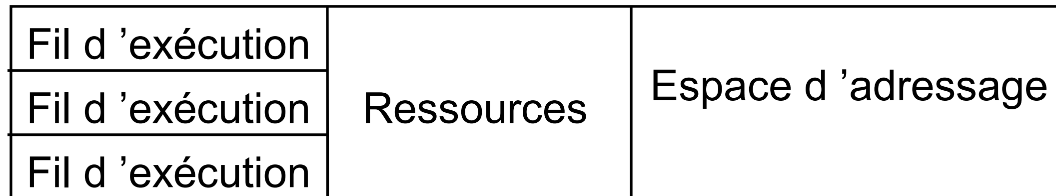
Fil d 'exécution Fil d 'exécution Fil d 'exécution	Ressources	Espace d 'adressage	Processus à threads

Notion de processus léger

Processus classique

Contexte processeur
CO, Pile

Contexte mémoire



Processus à threads

Contexte processeur
CO, Pile

Contexte processeur
CO, Pile

Contexte processeur
CO, Pile

Contexte mémoire

Notion de processus léger

- **Primitives**

  - **Création**

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);

  - **synchronisation entre threads**

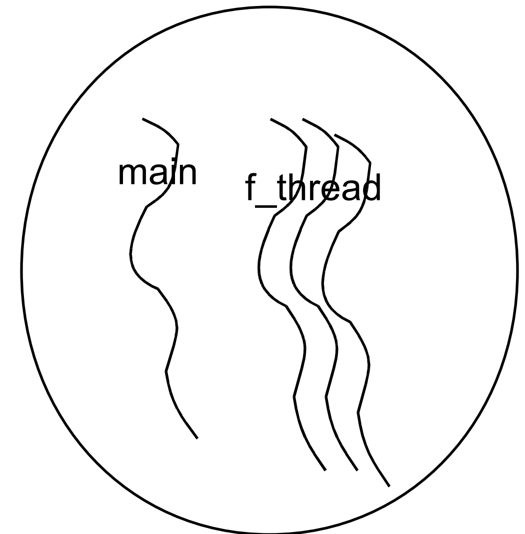
int pthread_join (pthread_t thread, void **value_ptr);

  - **terminaison de threads**

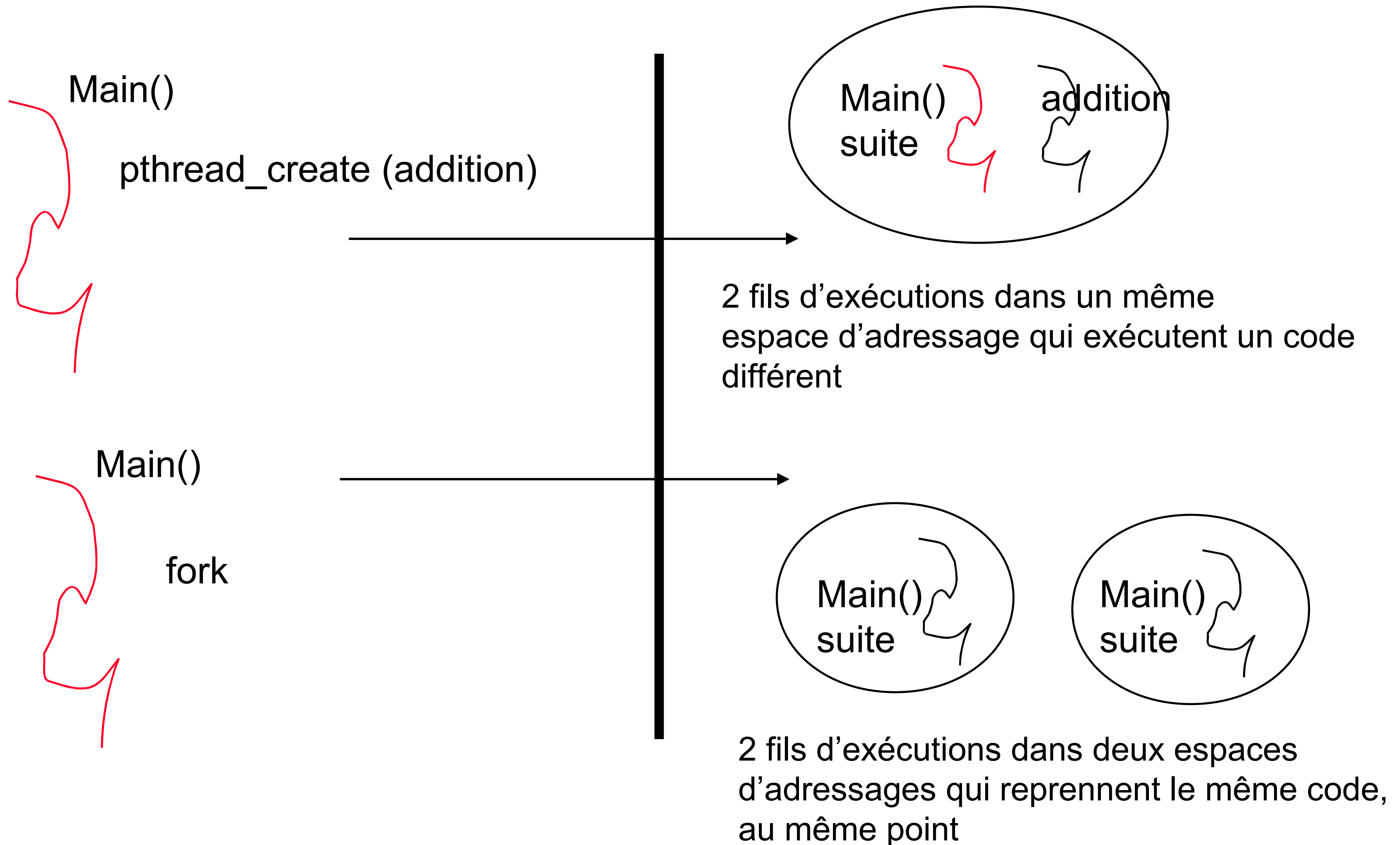
int pthread_exit (void **value_ptr);

Notion de processus léger

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

pthread_t pthread_id[3];

void f_thread(int i) {
printf ("je suis la %d-eme pthread d'identite %d.%d\n", i, getpid(),
pthread_self());
}

main()
{
int i;
for (i=0; i<3; i++)
if (pthread_create(pthread_id + i, pthread_attr_default, f_thread,
i) == -1)
      fprintf(stderr, "erreur de creation pthread numero %d\n", i);
printf ("je suis la thread initiale %d.%d\n", getpid(),
pthread_self());
pthread_join();
}
```



Notion de processus léger



Notion de processus léger

Processus classique (lourd)

espace d 'adressage protégé à un fil d 'exécution

Commutation de contexte
toujours changement d 'espace d 'adressage

Communication
outils entre espace d 'adressage (tubes, messages queues)

Pas de parallélisme dans un espace d 'adressage

Processus à threads (léger)

espace d 'adressage protégé à n fils d 'exécution (n ≥ 1)

Commutation de contexte : allégée
pas de changement d 'espace d 'adressage entre fils d 'un même processus

Communication : allégée
les fils d 'exécution d 'un même processus partagent les données, mais attention à la synchronisation

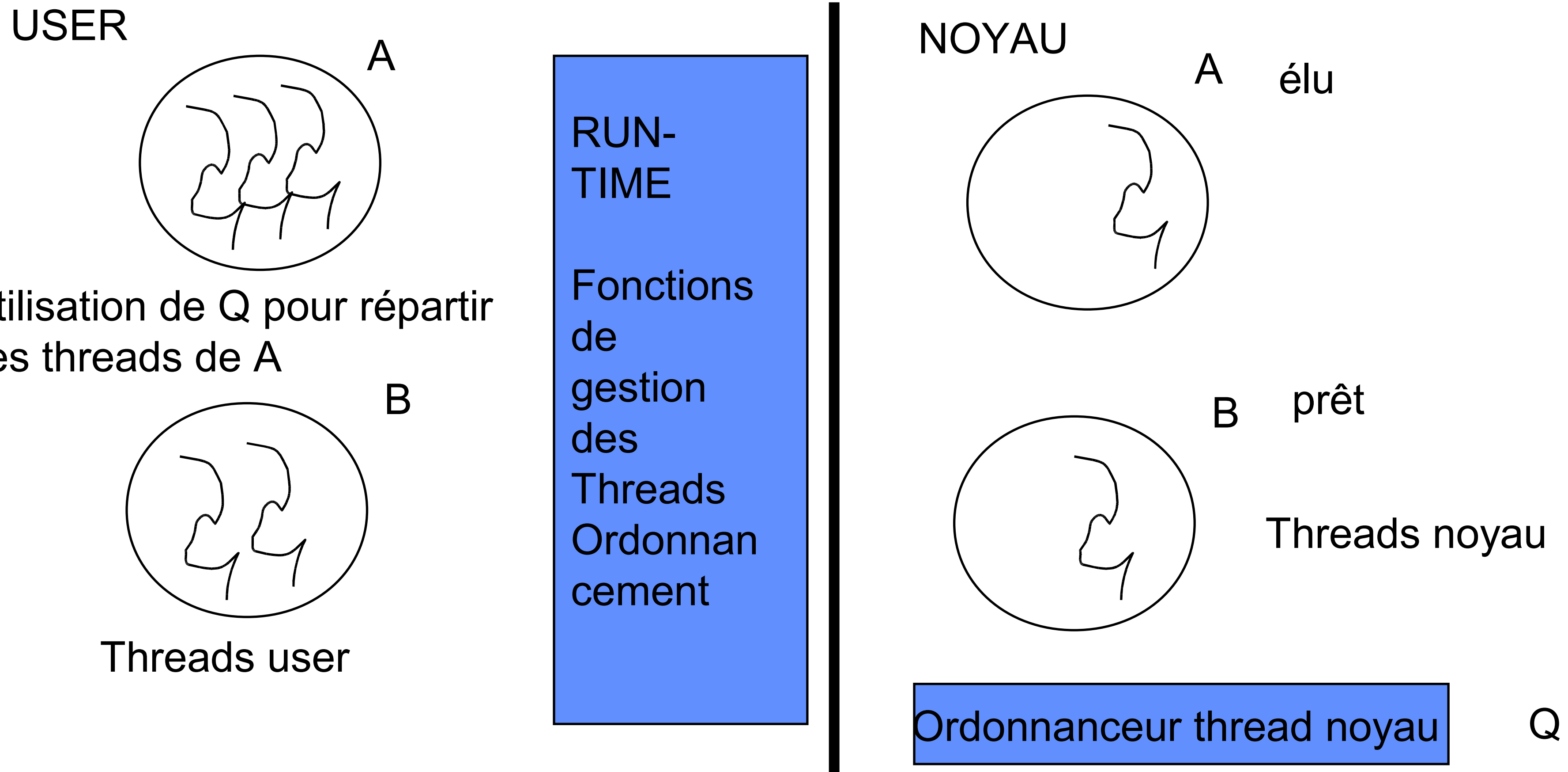
Parallélisme dans un espace d 'adressage

Notion de processus léger

- **Deux implémentations principales**
  - **Au niveau utilisateur : le noyau ignore l'existence éventuelle de plusieurs threads dans un processus**
  - **Au niveau noyau : le noyau connaît l'existence de plusieurs threads dans un processus.**

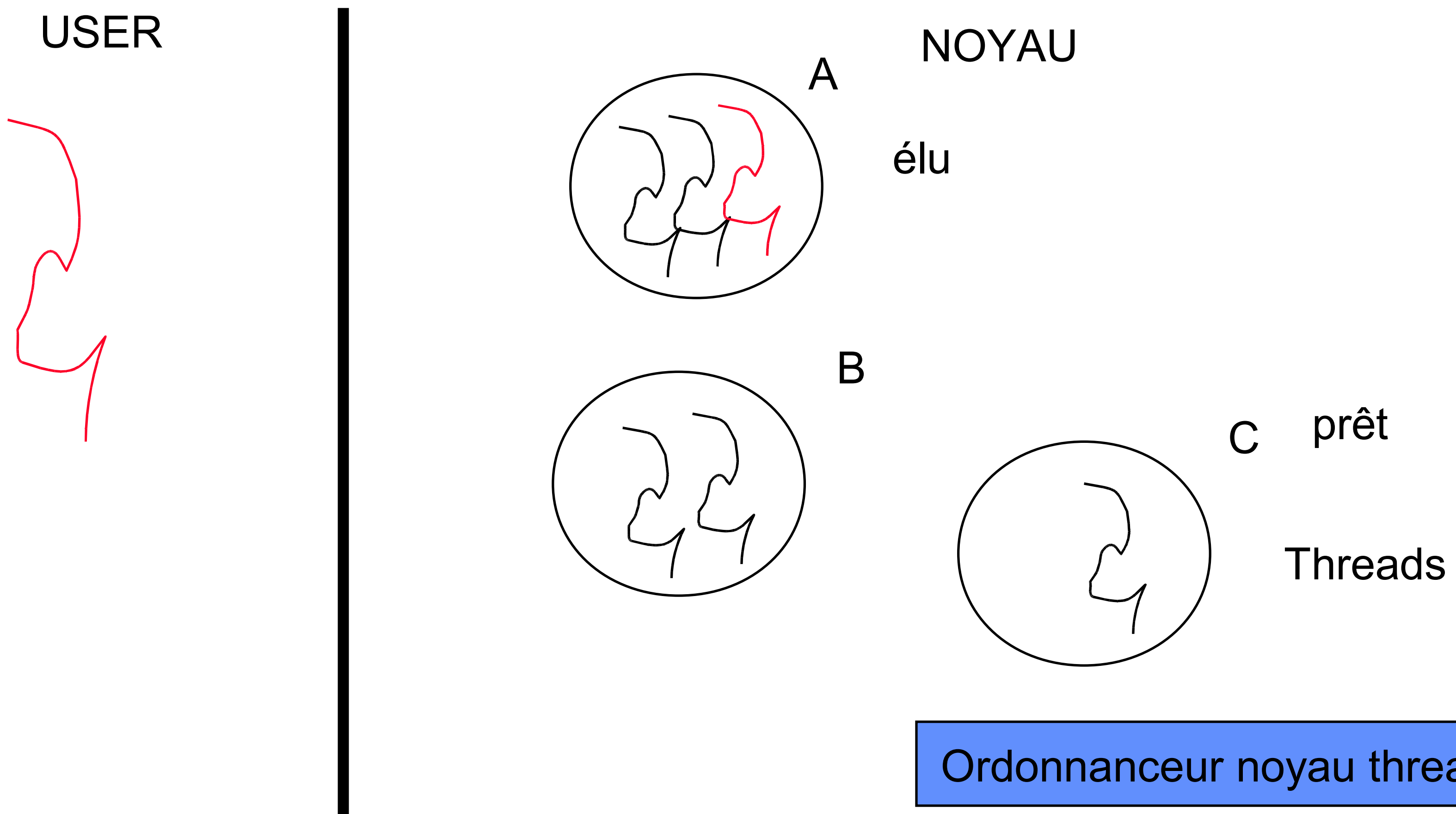
Notion de processus léger

- **Deux implémentations principales**
  - **Au niveau utilisateur : le noyau ignore l'existence éventuelle de plusieurs threads dans un processus**



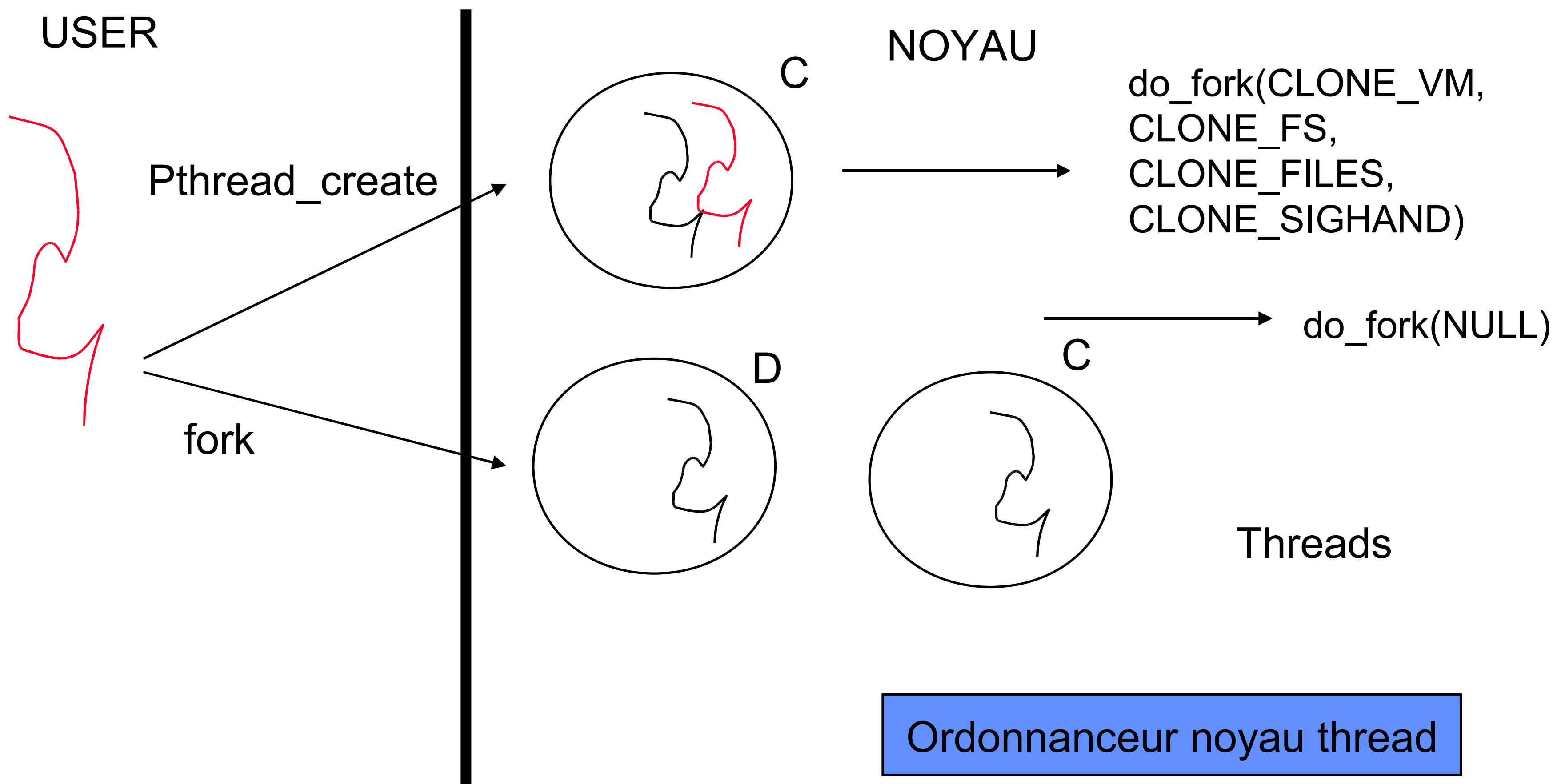
Notion de processus léger

- **Deux implémentations principales**
  - **Au niveau noyau : le noyau connait l'existence éventuelle de plusieurs threads dans un processus (LINUX)**



Notion de processus léger

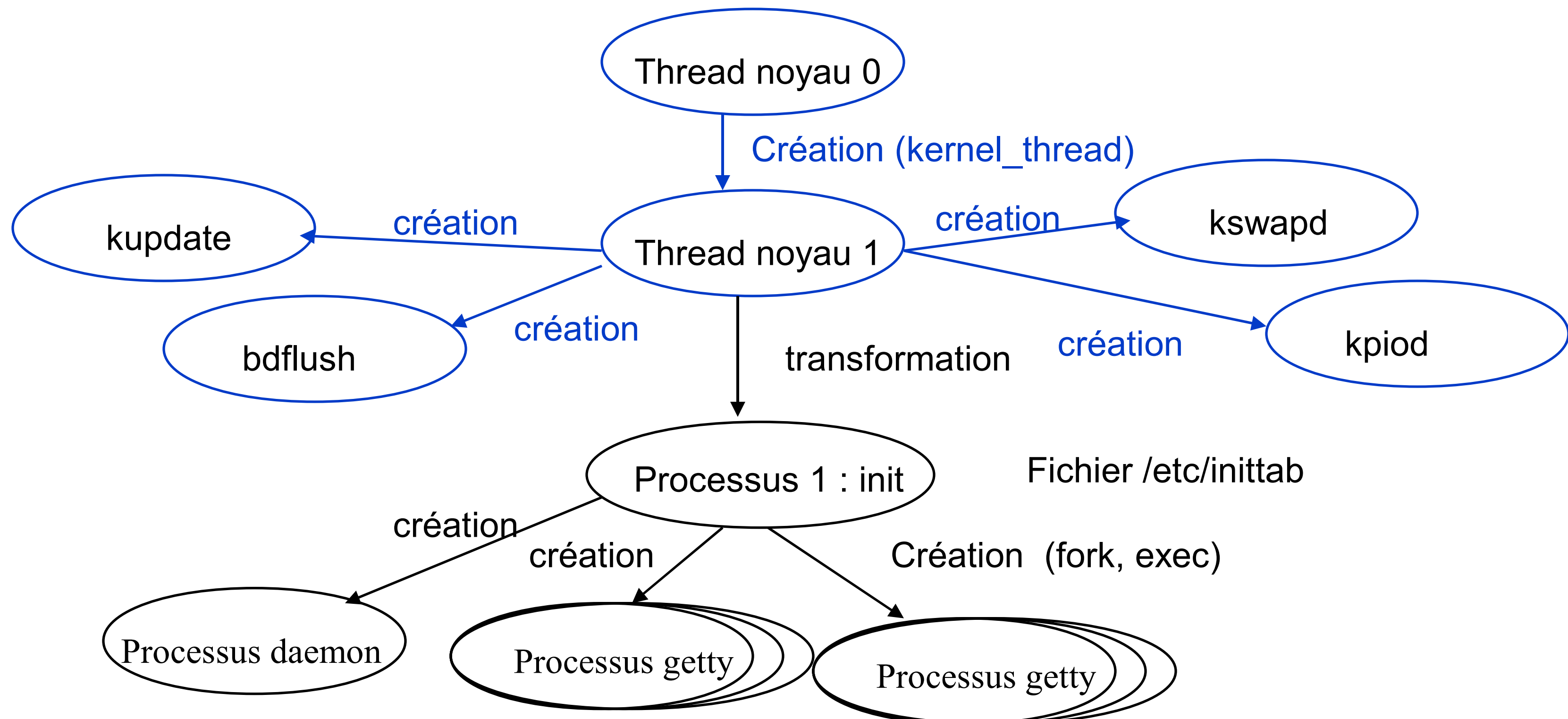
- **Deux implémentations principales**
  - **Au niveau noyau : le noyau connait l'existence éventuelle de plusieurs threads dans un processus**



Linux : les threads noyau

Le noyau Linux comporte un ensemble de 5 threads noyau qui s'exécutent en mode superviseur et remplisse des tâches spécifiques:

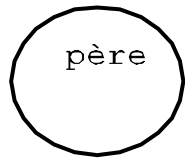
- **thread 0 : s'exécute lorsque le CPU est idle;**
- **kupdate, bdflush : gestion du cache disque;**
- **kswapd, kpiod : gestion de la mémoire centrale.**



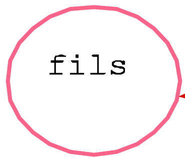
Implémentation noyau création d'un processus

Mode utilisateur

```
main()
{
int i, j, fd;
int ret;

ret = fork()

   }
}
```



retour fork()
-1 ou pid fils



retour fork()
1

Bibliothèque
routine d'enveloppe

```
fork () {

copie des paramètres
dans les registres
généraux

le numéro  l'appel
système  est mis
dans eax
eax <-- 5


int 0x80

if -256 < (eax) < 0
{
errno <- eax;
eax <- -1
}
}
```

Mode superviseur

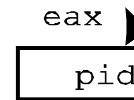
Gestionnaire
d'appel système
system_call()

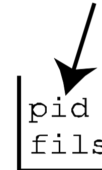
copie des registres
généraux dans la pile
noyau (eax, ebx, ecx, edx, esi et edi)
CALL *sys_call_table (eax)

sys_fork() --> do_fork()

return(pid)

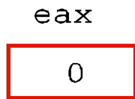
eax

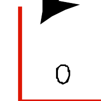




fils

eax





restauration des
registres généraux
ret_from_sys_call()

restauration des
registres généraux
ret_from_sys_call()

6 registres du cpu permettent le passage
d'un paramètre
entre le noyau et le contexte utilisateur

ret_from_sys_call (recharge les registres avec les valeurs contenues dans la pile noyau)

Primitive de création de processus

fork()→ sys_fork()→ do_fork(0, Co user père)
Pthread_create(fn) → sys_clone(flags, fn) → do_fork(flags, fn)
Flags = CLONE_VM, CLONE_FS, CLONE_FILES, CLONE_SIGHAND

```
int do_fork()
{
		int nr;
		unsigned long new_stack;
		struct task_struct *p;

		p = (struct task_struct *) kmalloc();  -- allouer une structure task_struct
		new_stack = alloc_kernel_stack();  --allouer une pile noyau
		nr = find_empty_process();  -- trouver une entrée libre dans la table des processus
		p->pid = get_pid(clone_flags);  -- affecter un nouveau PID
		p->utime = p->stime = 0;  -- mise à 0 des temps cpu
		p->cutime = p->cstime = 0;
		task[nr] = p; -- PCB chainé à l'entrée de la table des processus
		SET_LINKS(p);  -- chainage file des processus
		nr_tasks++; -- un processus de plus

		/* copy all the process information */  -- copie du contexte du père ou  non selon les flags
		copy_files(clone_flags, p);
		copy_sighand(clone_flags, p);
		copy_mm(clone_flags, p);
		copy_thread(nr, clone_flags, usp, p, regs);  -- copie du contexte processeur du père et
initialialisation : eax = 0; CO user = CO père ou CO fonction.
		wake_up_process(p);   -- le processus passe dans l'état RUNNING
		return p->pid;  -- retour du pid créé

}
```