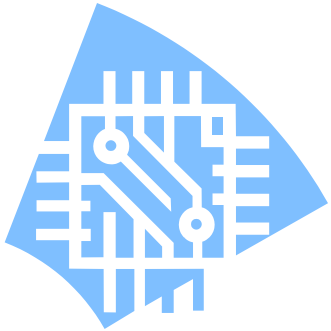
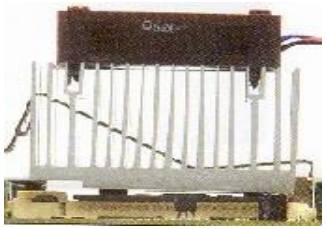
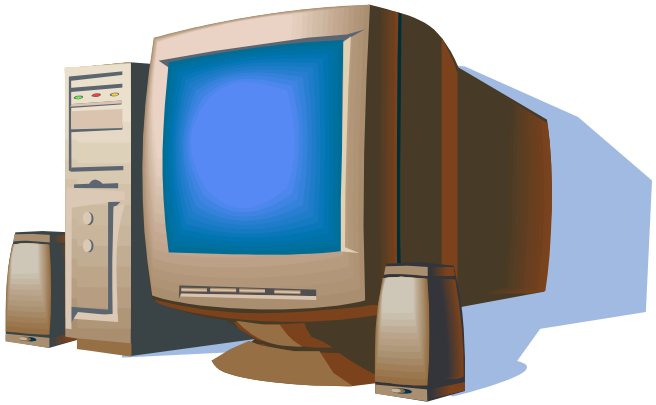
Rôle du système d’exploitation





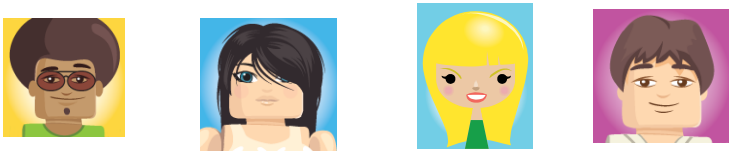


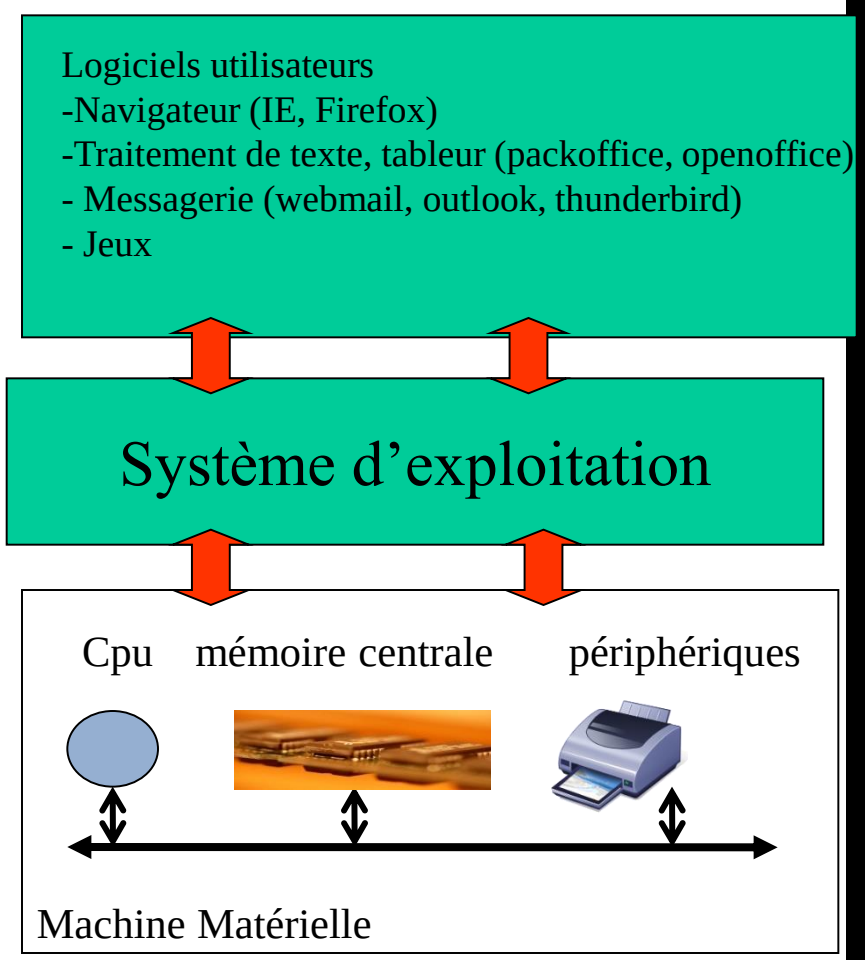
Définition d'un système d'exploitation

Ensemble de programmes qui réalisent l'interface entre le matériel de l'ordinateur et les utilisateurs. Il a deux objectifs principaux :

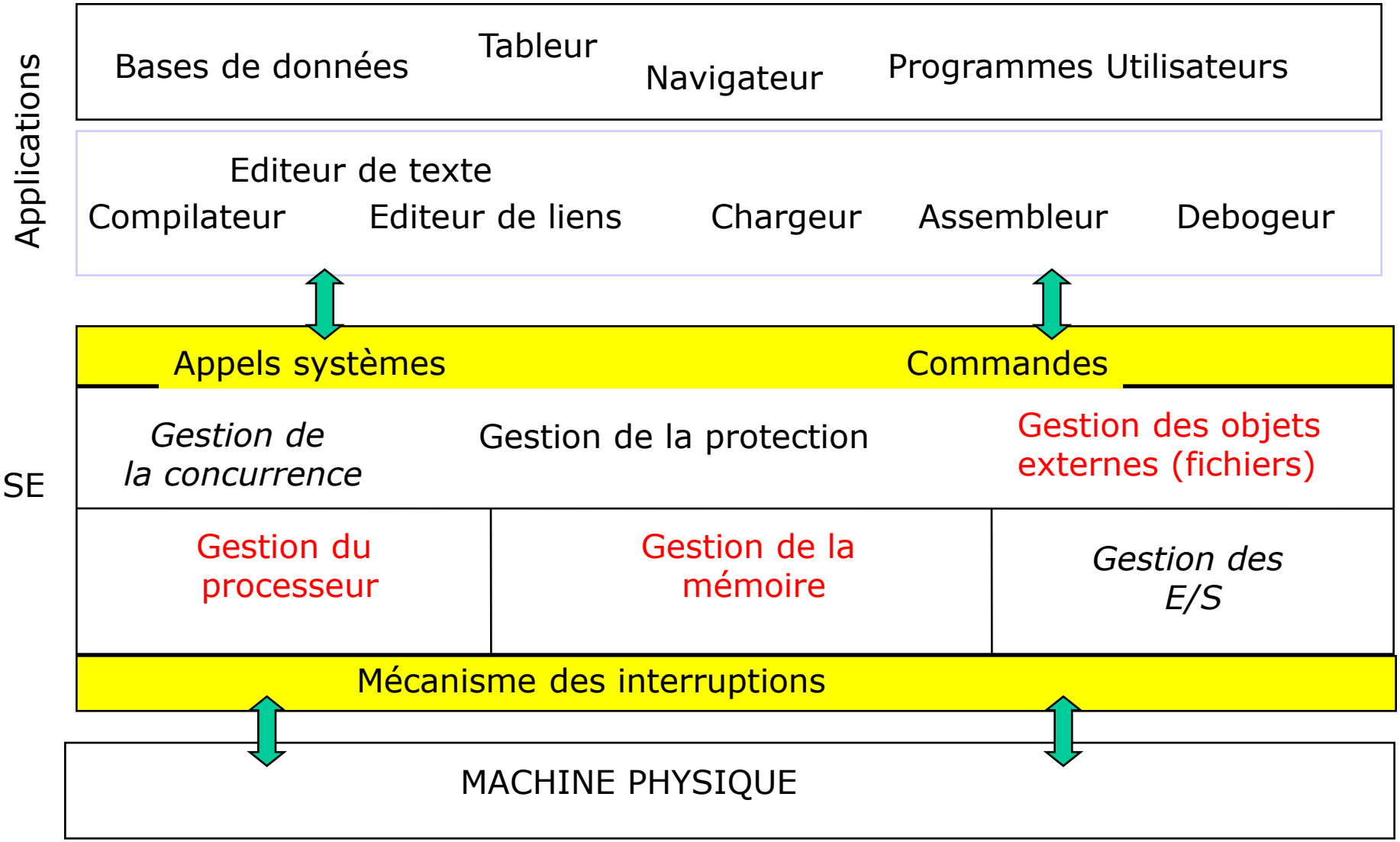
- **construction au dessus du matériel d'une machine virtuelle plus facile d'emploi et plus conviviale**
- **prise en charge de la gestion de plus en plus complexe des ressources et partage de celle-ci**

Comme son nom le suggère, le SE a en charge l'exploitation de la machine pour en faciliter l'accès, le partage et pour l'optimiser





FONCTIONS D'UN SYSTEME D'EXPLOITATION



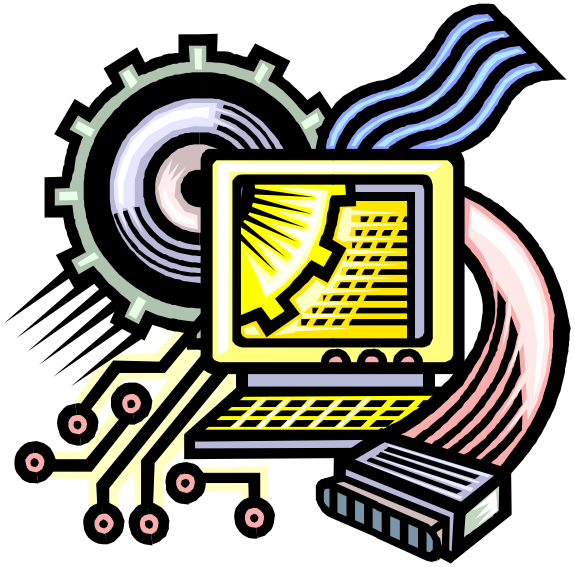
FONCTIONS D'UN SYSTÈME D'EXPLOITATION

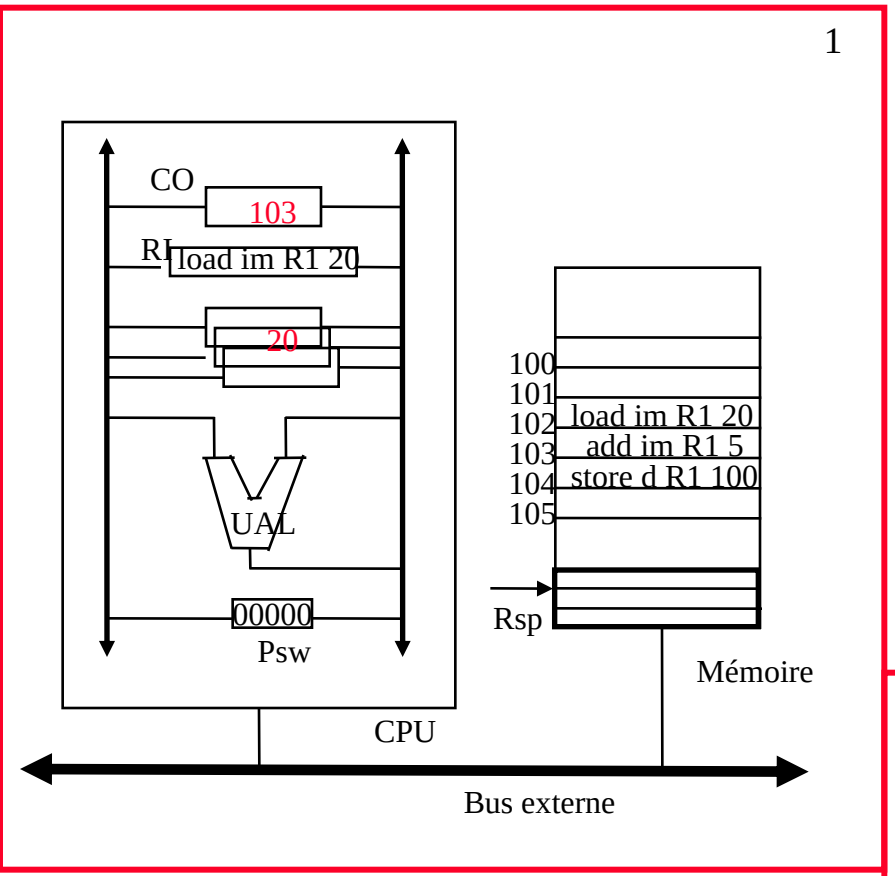
Gestion du processeur

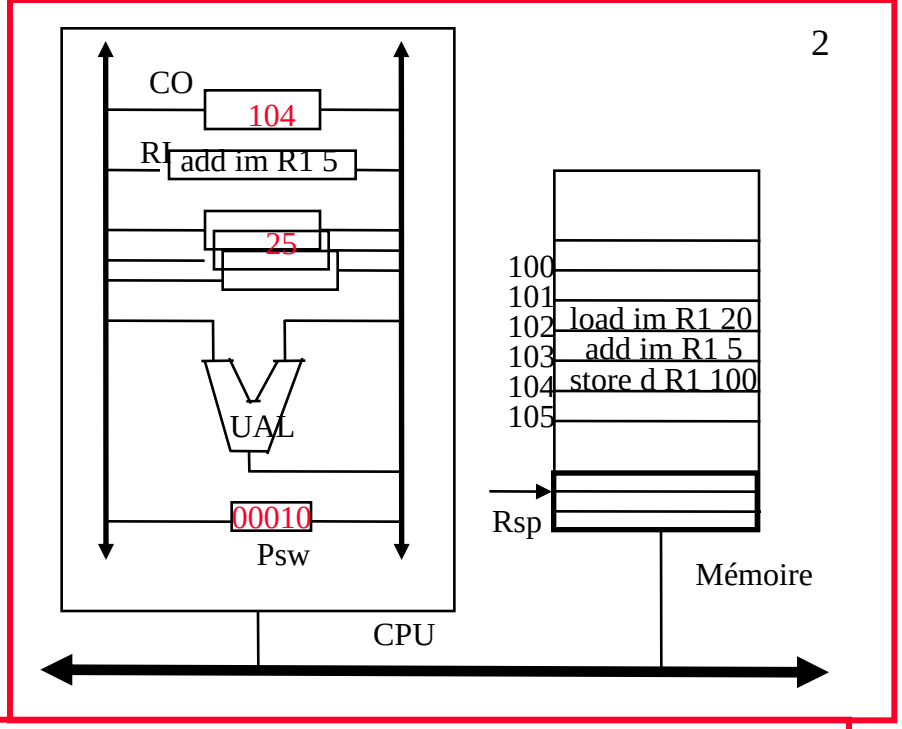
le système doit gérer l'allocation du processeur aux différents programmes pouvant s'exécuter (**processus**). Cette allocation se fait par le biais d'un **algorithme d'ordonnancement** qui planifie l'exécution des programmes

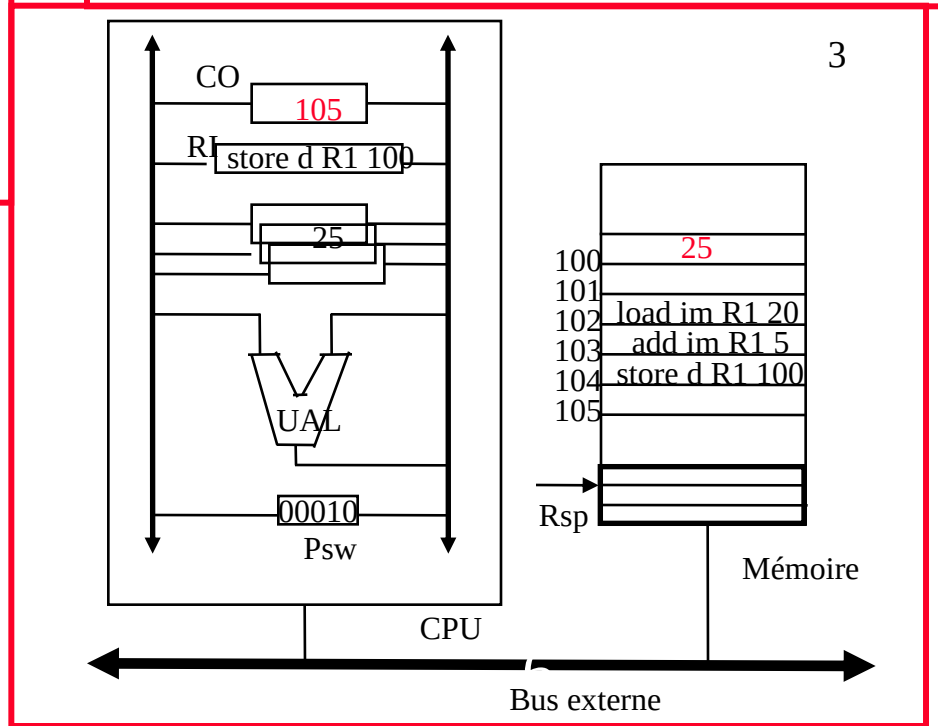
- Un processus est un programme en cours d'exécution
- Un algorithme d'ordonnancement est un programme du système qui planifie l'exécution des processus

I.
Processus









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

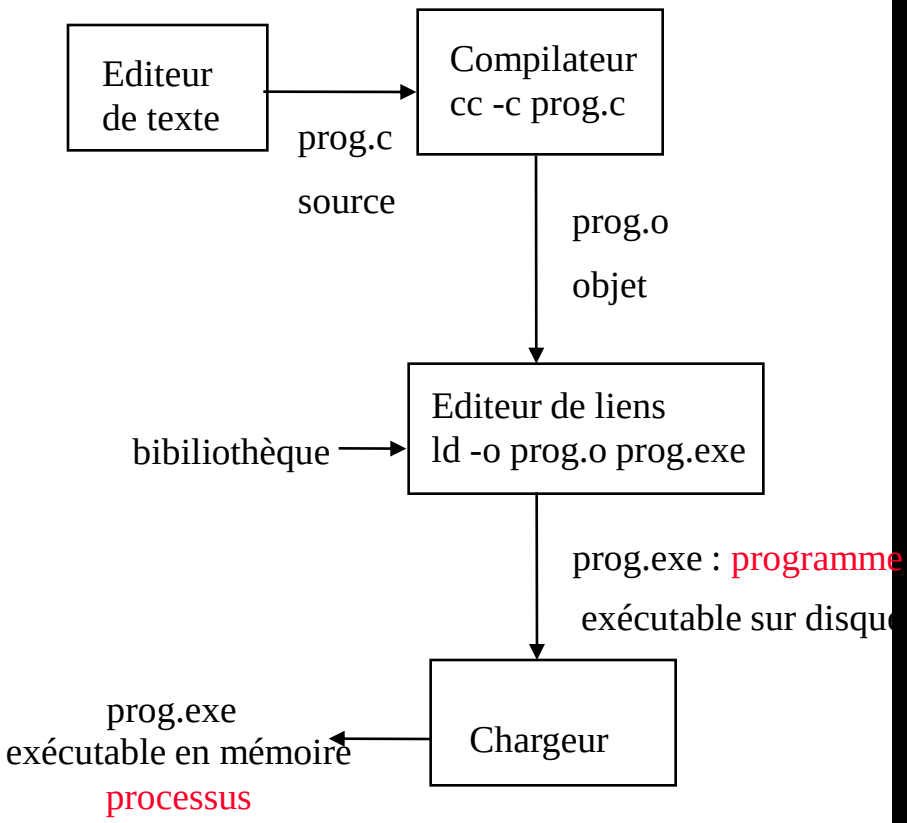
Notion de processus

- **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.**

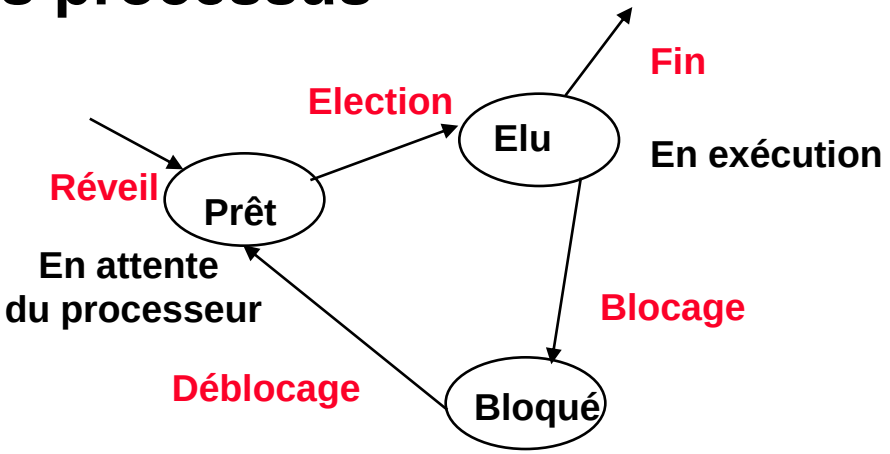
  - **Un processus est l ’instance dynamique d ’un programme et incarne le fil d ’exécution de celui-ci**

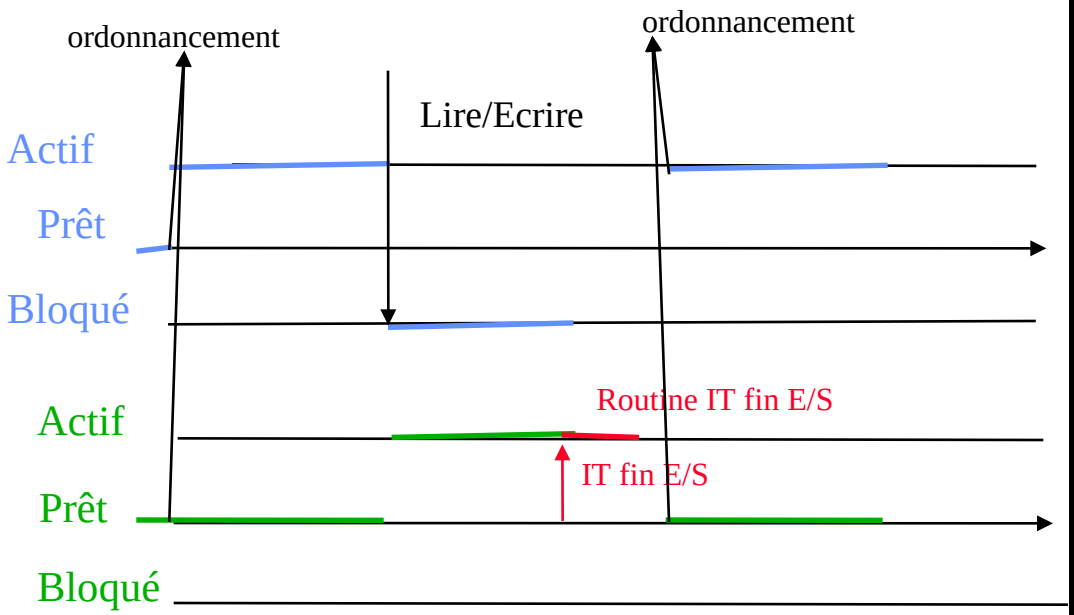
  - **Un processus évolue dans un espace d’adressage protégé**



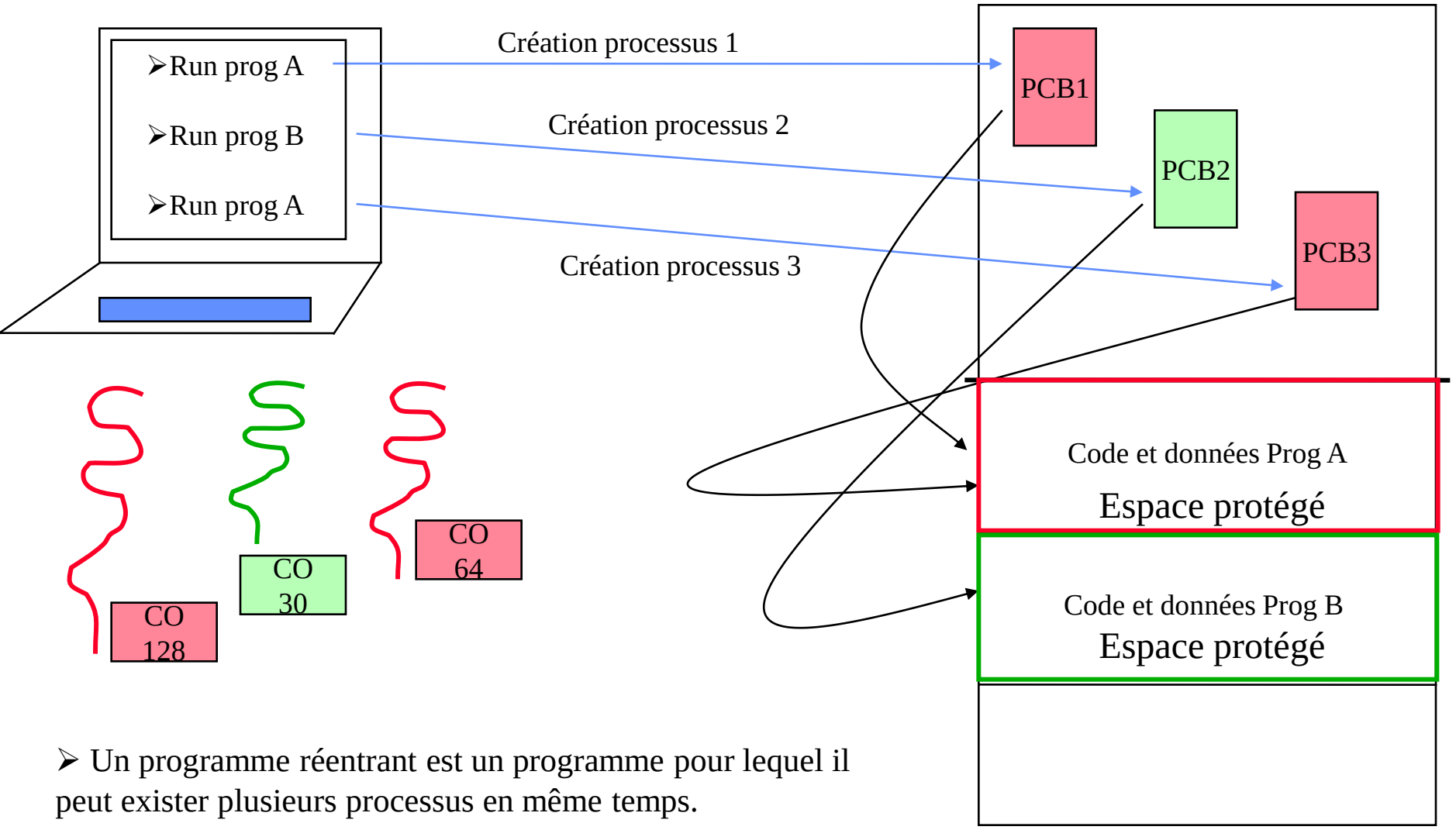
bloc de contrôle de processus PCB
Etats des processus

identificateur processus
état du processus
contexte pour reprise (registres et pointeurs, piles,..) compteur instructions
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,..





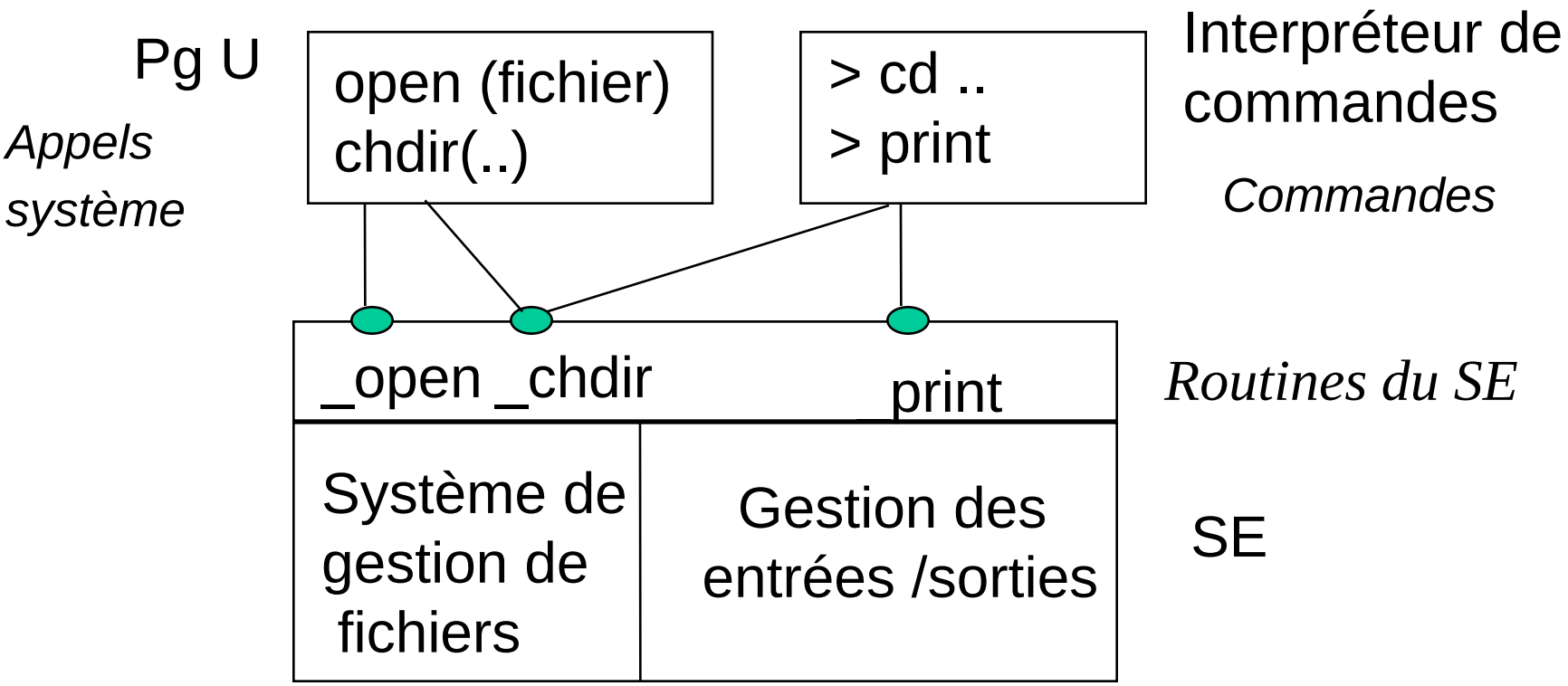
Notion de processus programme réentrant



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

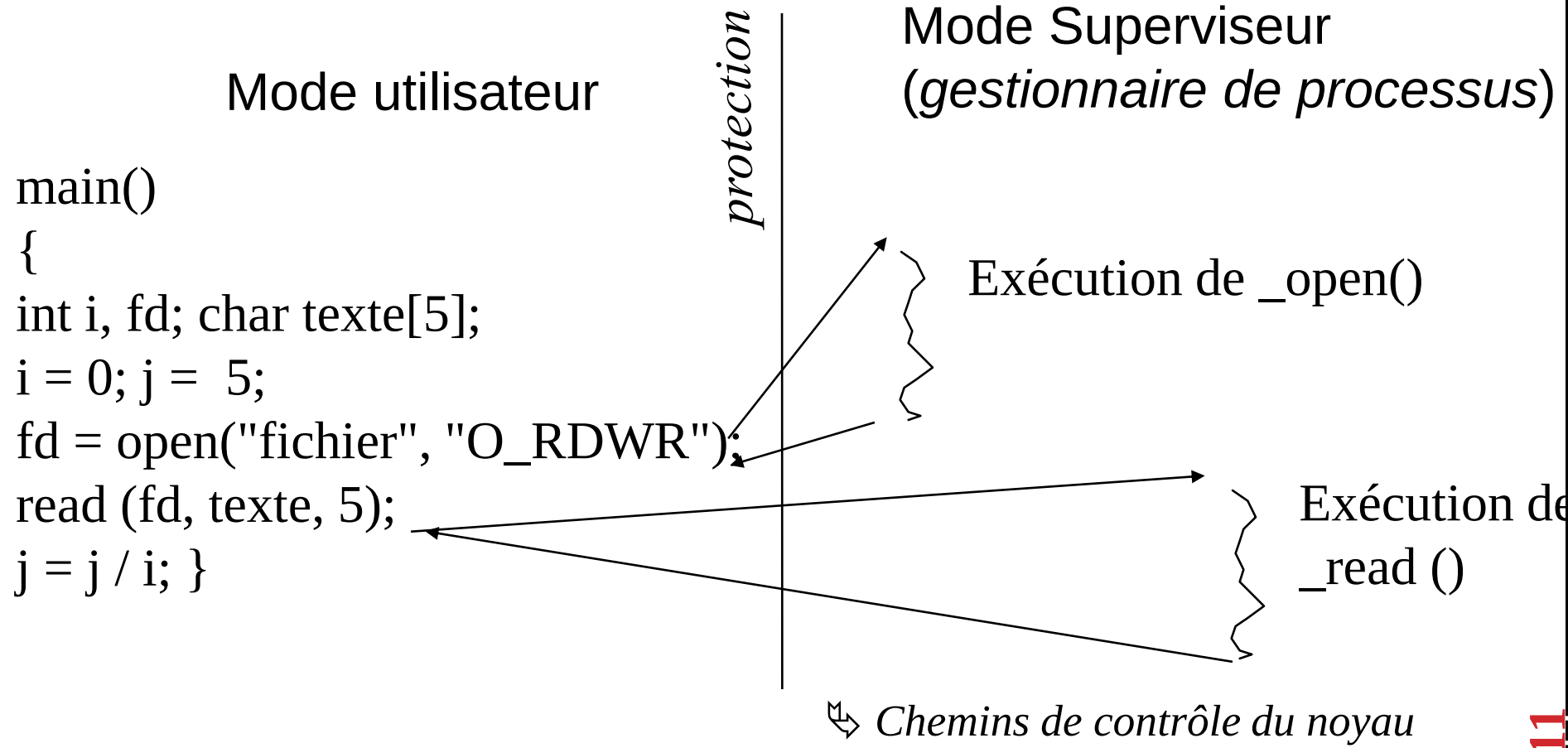
NOTIONS DE BASE

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



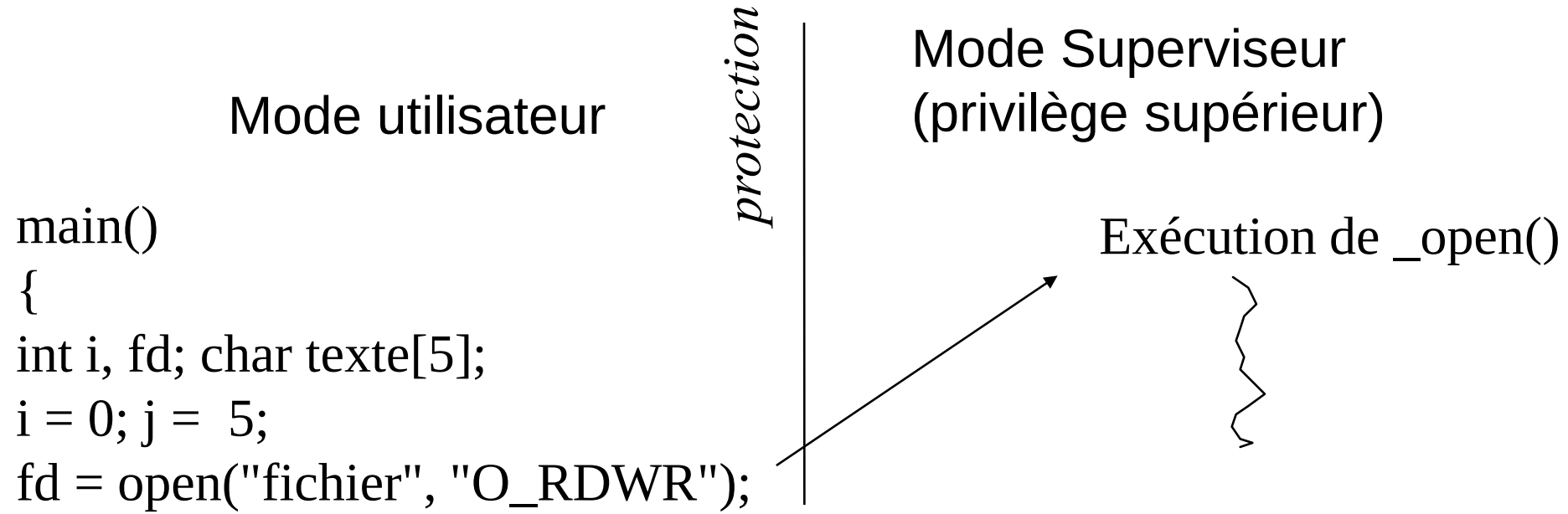
NOTIONS DE BASE : modes d'exécutions

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



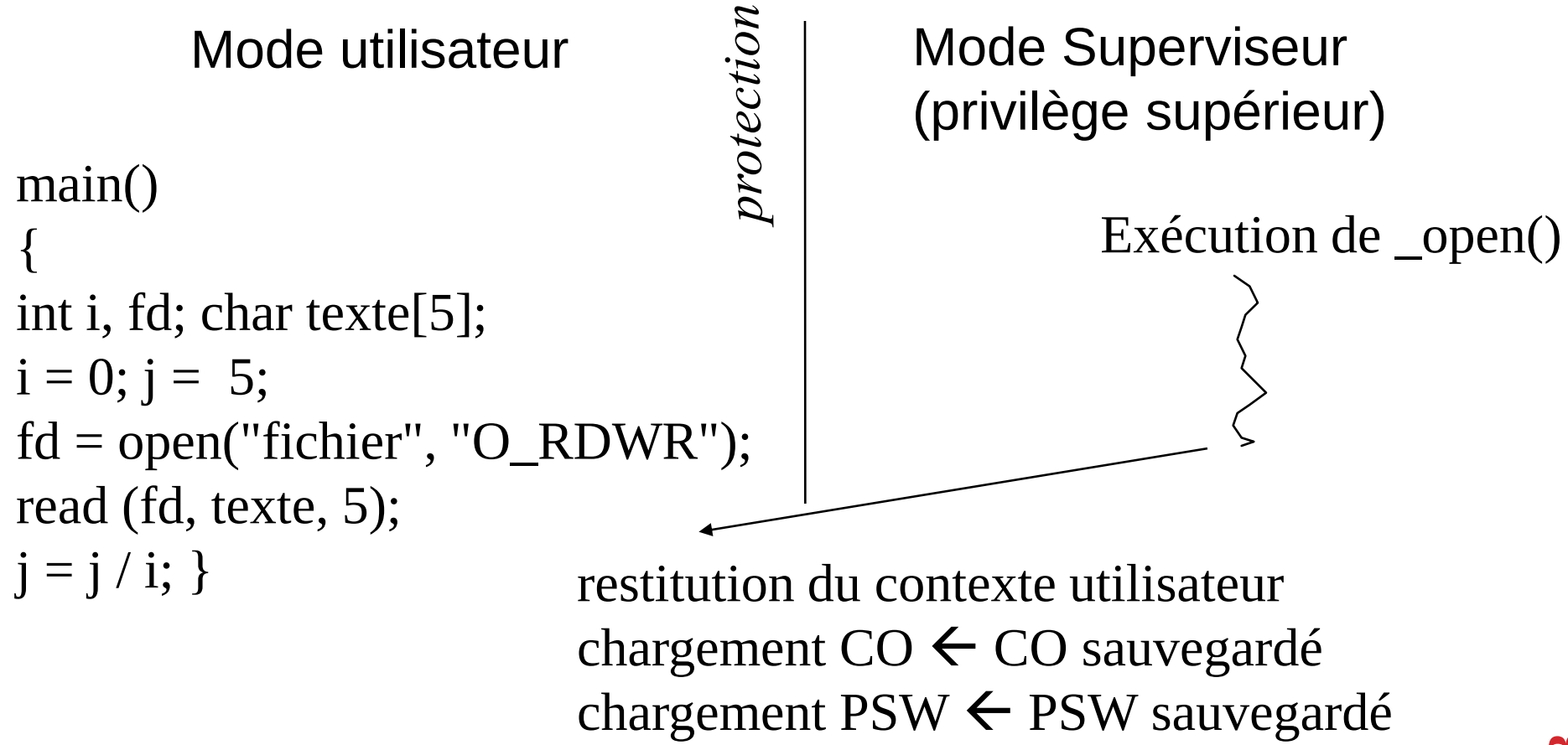
NOTIONS DE BASE : modes d'exécutions

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



NOTIONS DE BASE : modes d'exécutions

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



NOTIONS DE BASE : commutations de contexte

Mode utilisateur

Mode Superviseur

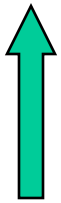
protection

```
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



IT HORLOGE
MATERIEL

Trappe = interruption synchrone
IT = interruption asynchrone

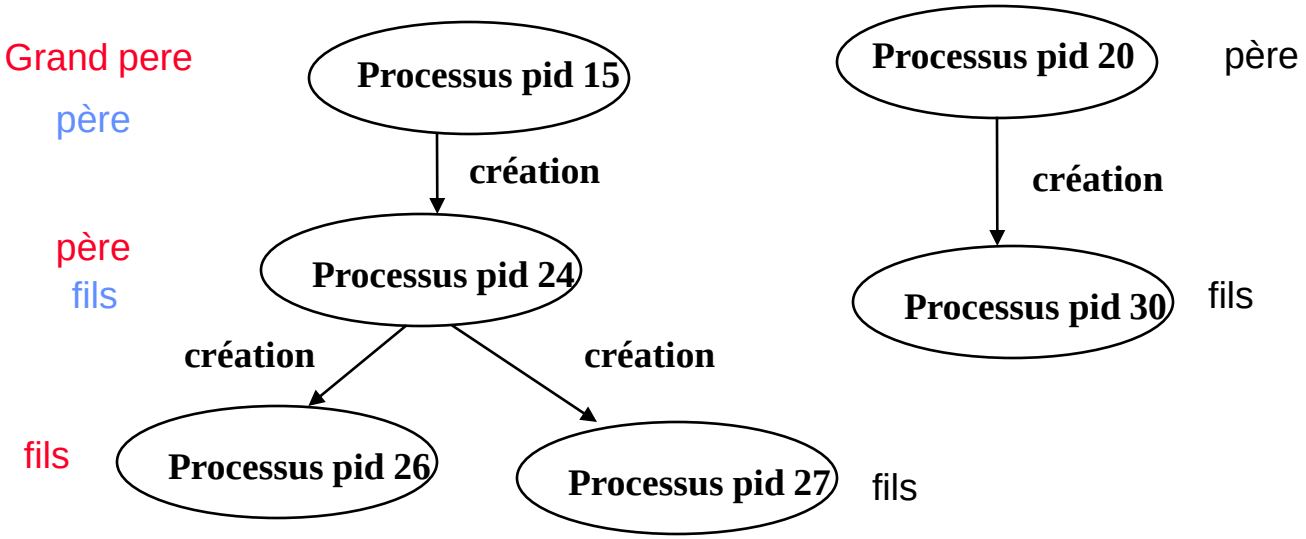
Processus
Processus LINUX

Caractéristiques Générales

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

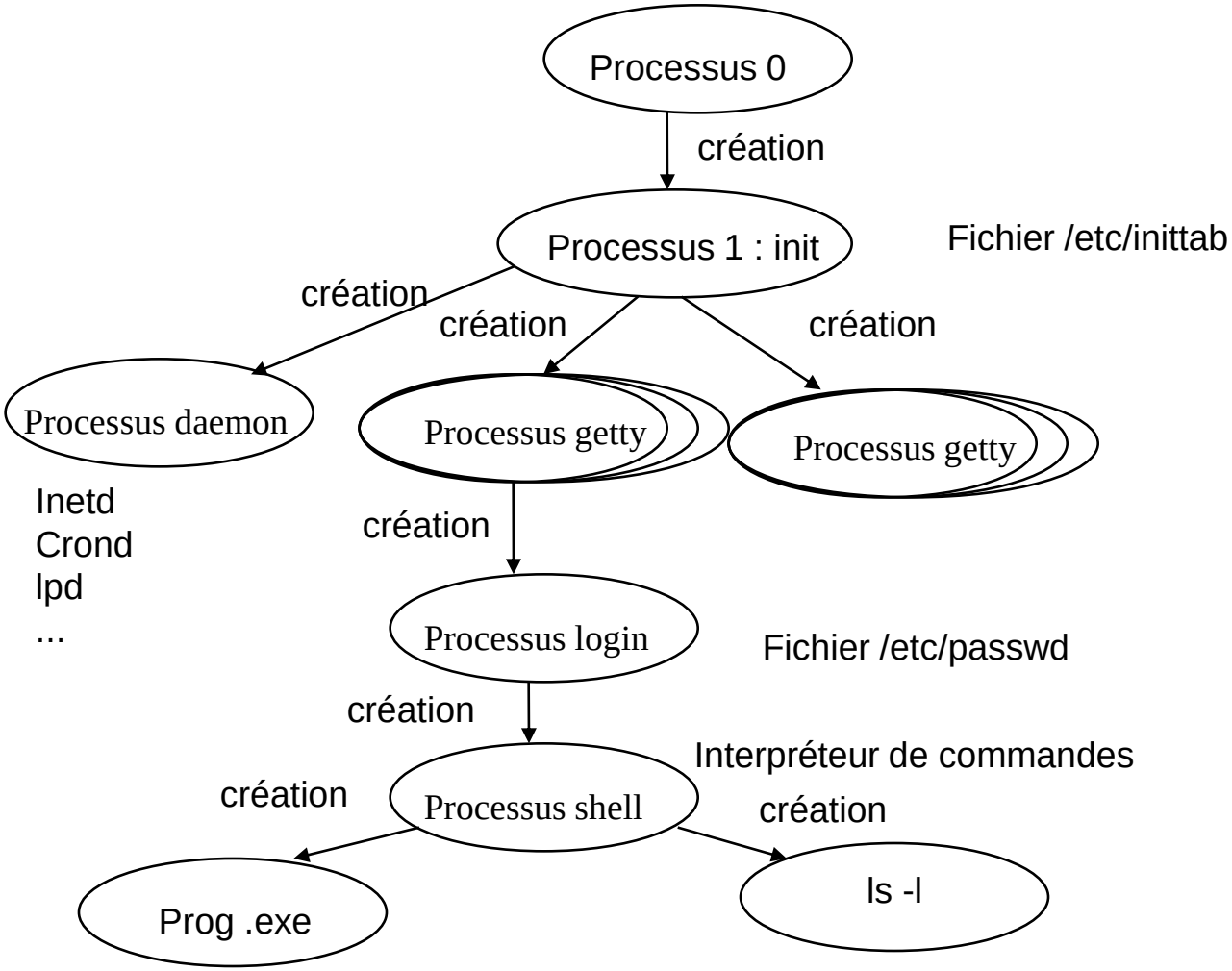
Processus Linux

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

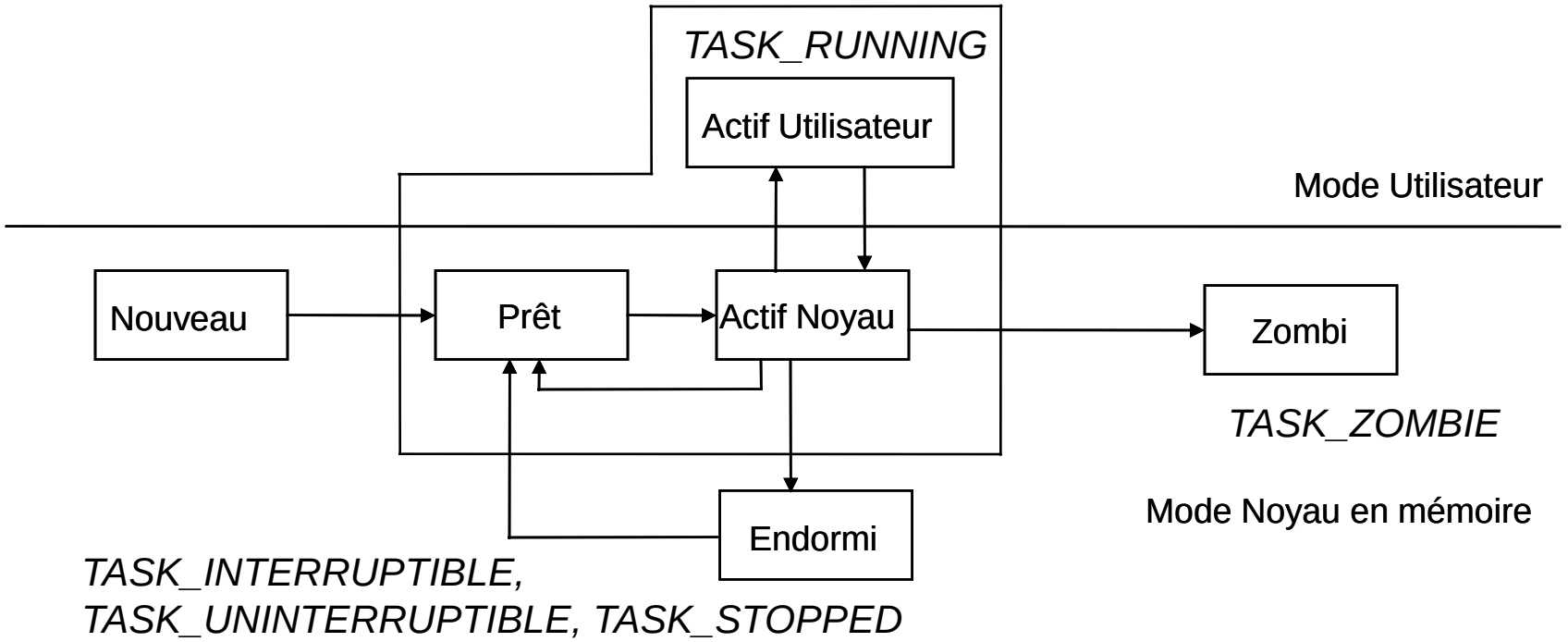


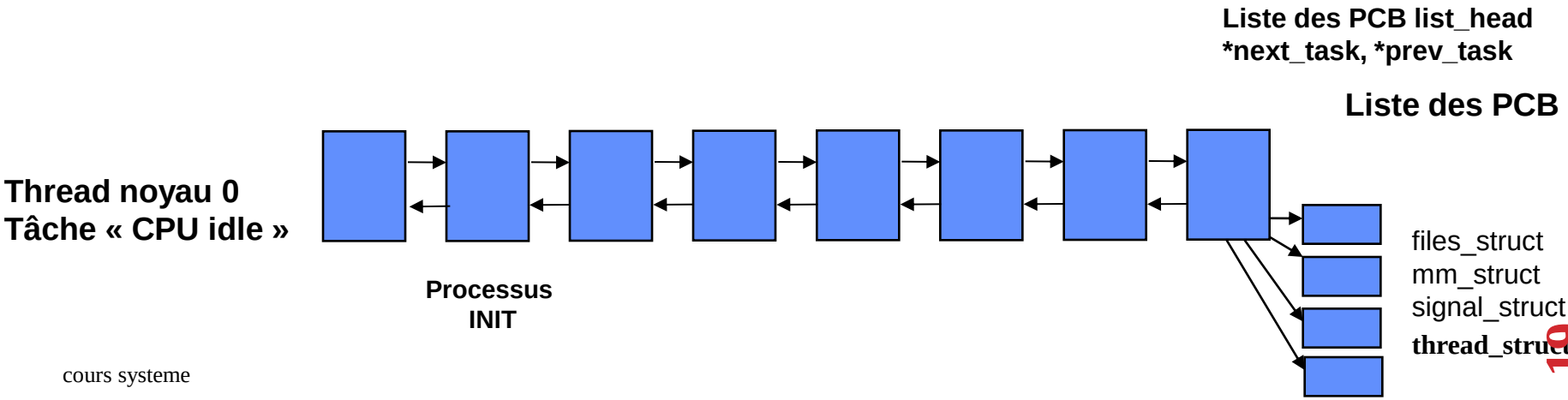
Processus Linux

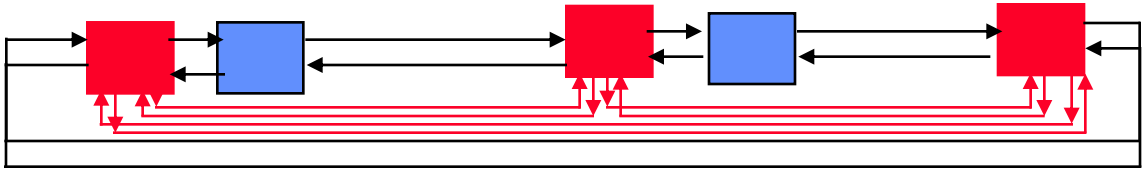
- **Tout le système Linux repose sur ce concept arborescent**



Processus Linux







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

Liste des PCB list_head *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**
  pid_t getpid(void)
  retourne le pid du processus appelant
  pid_t getppid(void)
  retourne le pid du père du processus appelant

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

```
S  PID PPID PRI TIME
S  581 579   60    bash
S  592 581   61    essai
S  593 592   61    essai
R  599 580   73    ps
```

Arrêter un processus : kill

La commande `kill` permet d'envoyer un signal à un processus.

Un signal est un moyen de communication entre processus; il permet de spécifier à un processus qu'un évènement est arrivé. Chaque signal est identifié par un nom et un numéro

Le processus réagit au signal reçu (par exemple en s'arrêtant)

```
kill – numerosignal  pid
```

SIGKILL 9 Force le processus à se terminer.

SIGTERM 15 signal par défaut. Termine le processus en « douceur ».

```
linux-9bxb:~ # ./essaibis&
[1] 4052
linux-9bxb:~ # ps
  PID TTY          TIME CMD
 3424 pts/1    00:00:00 bash
 4052 pts/1    00:00:00 essaibis
 4055 pts/1    00:00:00 ps
linux-9bxb:~ # kill 4052
[1]+  Terminated              ./essaibis
linux-9bxb:~ # ps
  PID TTY          TIME CMD
 3424 pts/1    00:00:00 bash
 4060 pts/1    00:00:00 ps
linux-9bxb:~ #
```

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

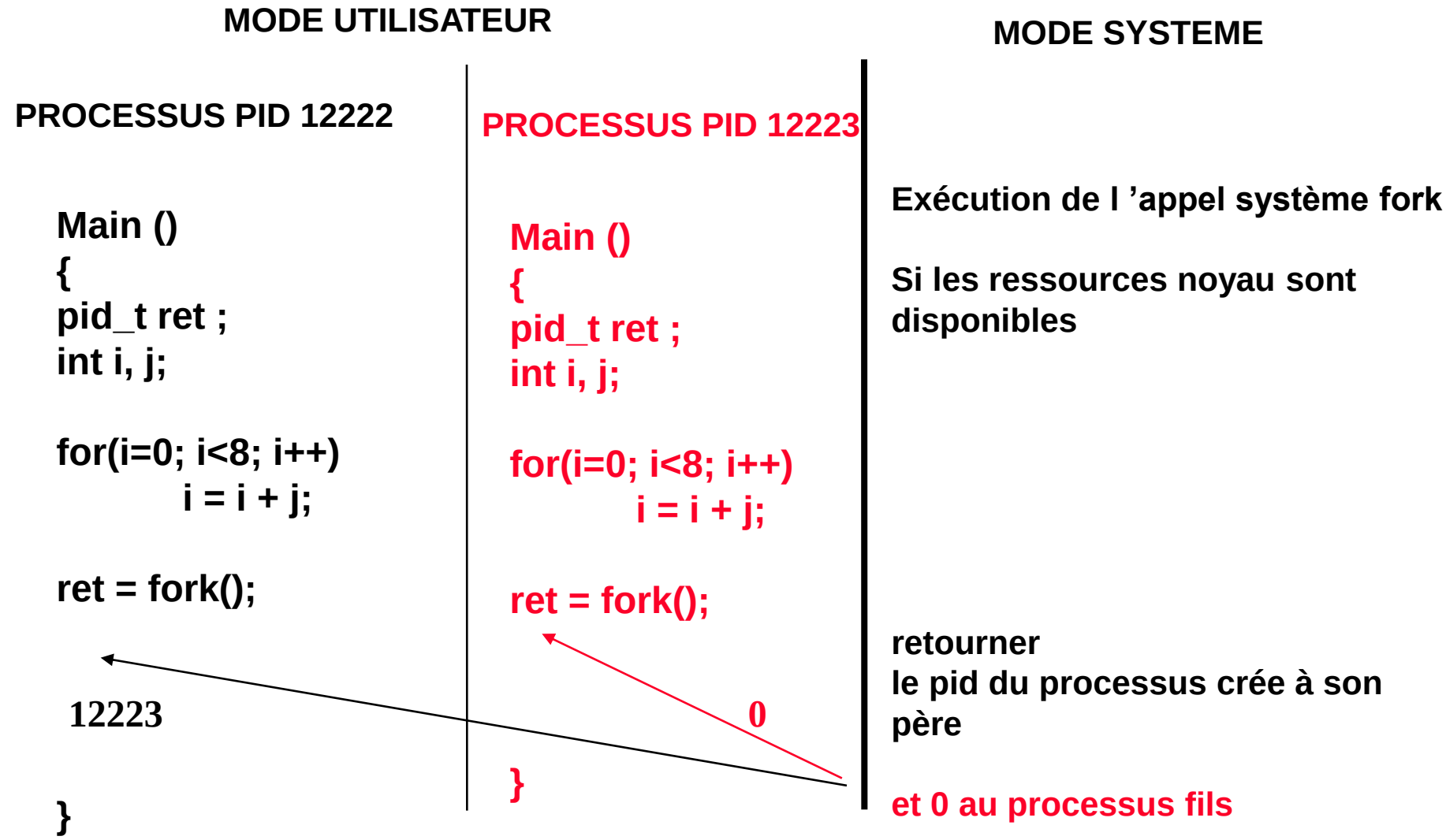
- allouer une entrée de la table des processus au nouveau processus

- allouer un pid unique au nouveau processus

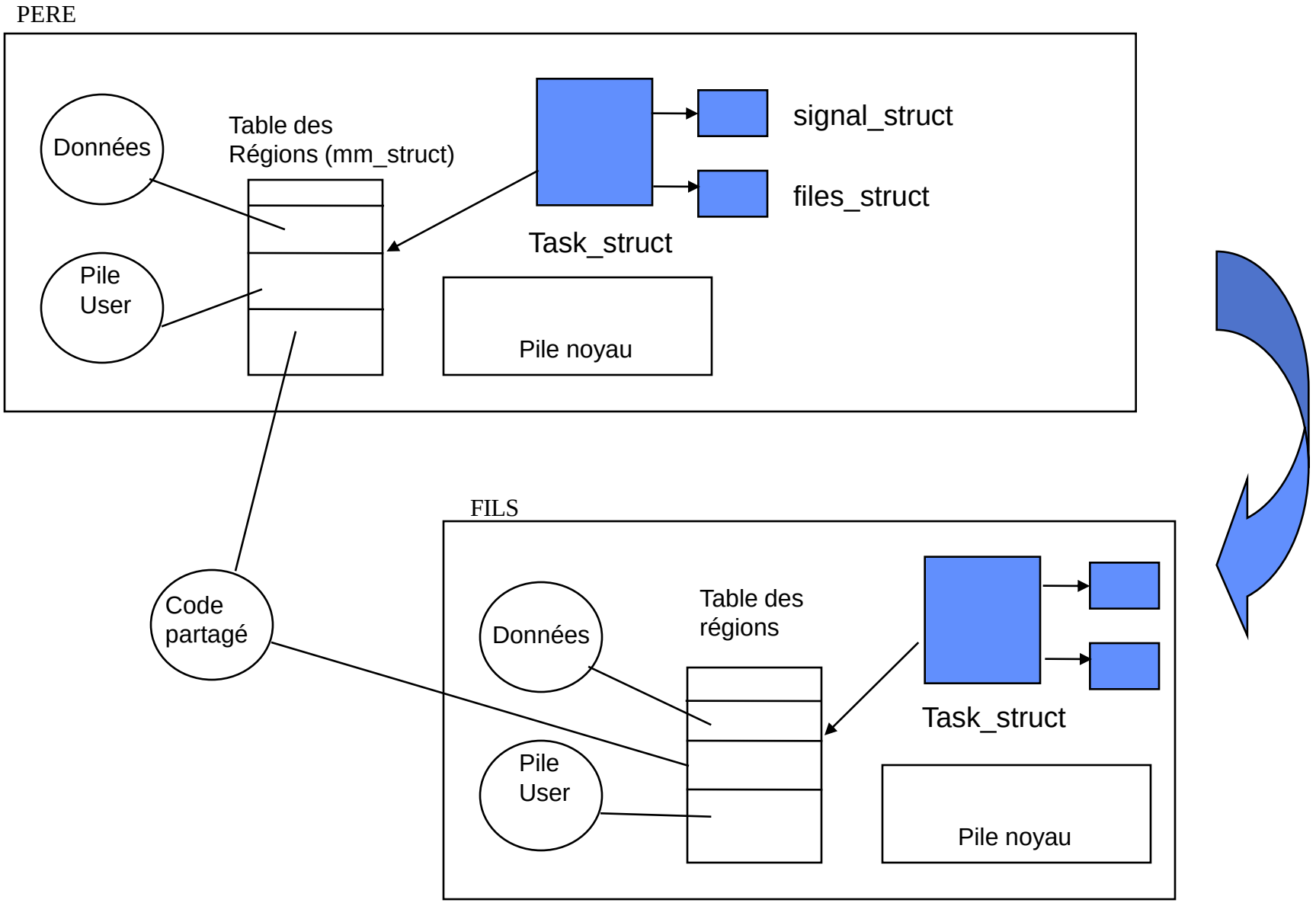
- dupliquer le contexte du processus parent (code, données, pile)

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

Primitive de création de processus



Primitive de création de processus



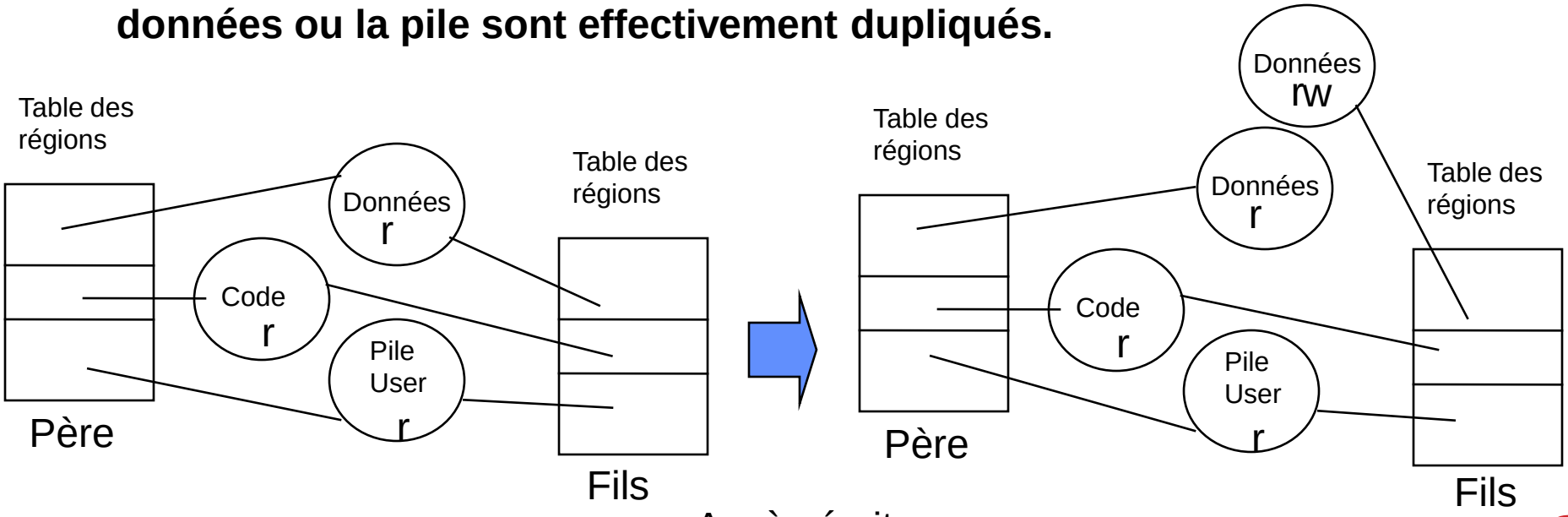
Primitive de création de processus

- **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.



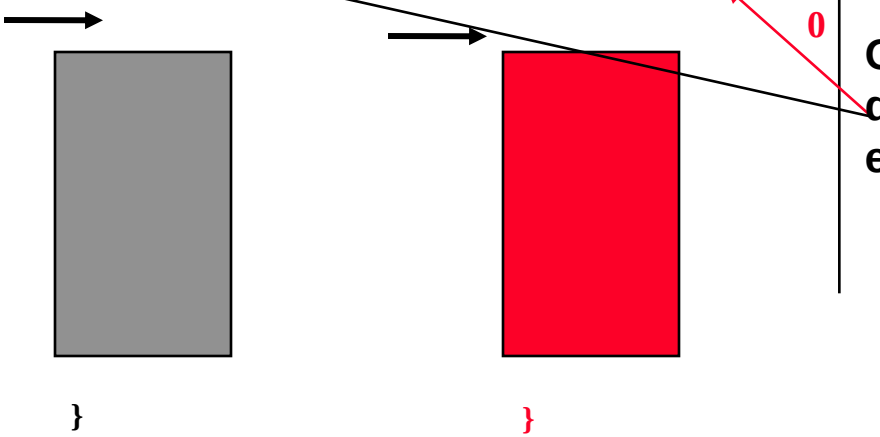
Primitive de création de processus

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();
```



- 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

Génétique des processus Linux (1)

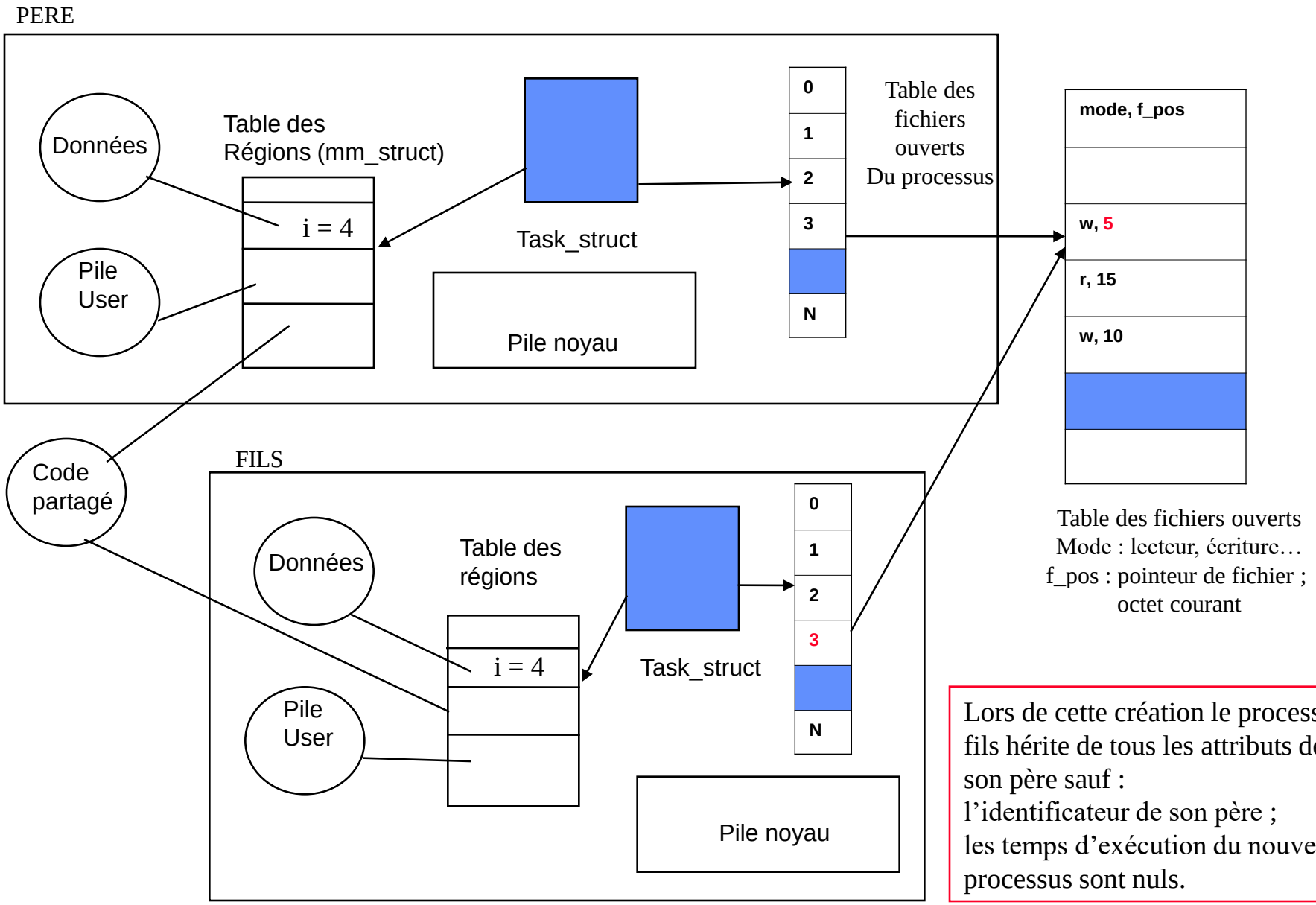


Table des fichiers ouverts
Mode : lecteur, écriture…
f_pos : pointeur de fichier ;
octet courant

Lors de cette création le processus fils hérite de tous les attributs de son père sauf :
l'identificateur de son père ;
les temps d'exécution du nouveau processus sont nuls.

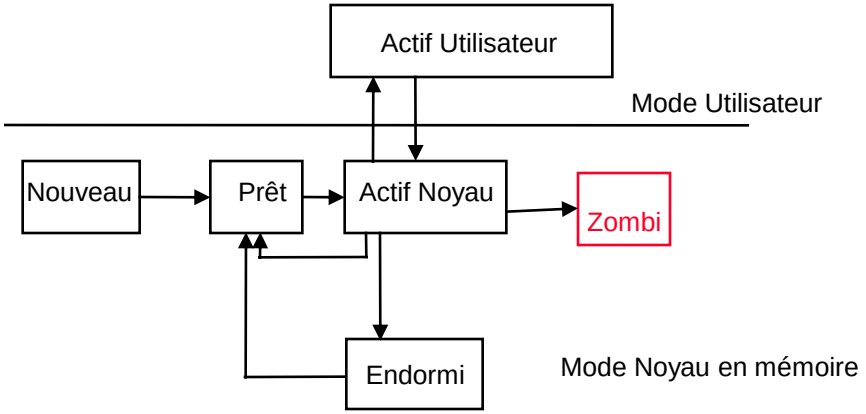
Génétique des processus Linux (1)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


main()
{
int pid, fp, i, j;
char ch[4];

fp = open("fichier", O_RDWR);
pid = fork();
if (pid==0)
        {
        for(i=0; i<4; i++)
        { read(fp, ch, 4);
        printf("hello fils; %s\n ", ch);
        sleep(4);
         }
        }
else
        {
        for(j=0; j<3; j++)
        {read(fp, ch, 4);
        printf("hello pere; %s\n ", ch);
        sleep(5);}
        wait();
        close(fp);
        }


}
```

```
volcan:~/MPS $ essai
hello pere; 1234
hello fils; 5678
 hello fils; 9cou
 hello pere; coul'
 hello fils; espe
 hello pere; tits
 hello fils; amis
volcan:~/MPS $
```

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.

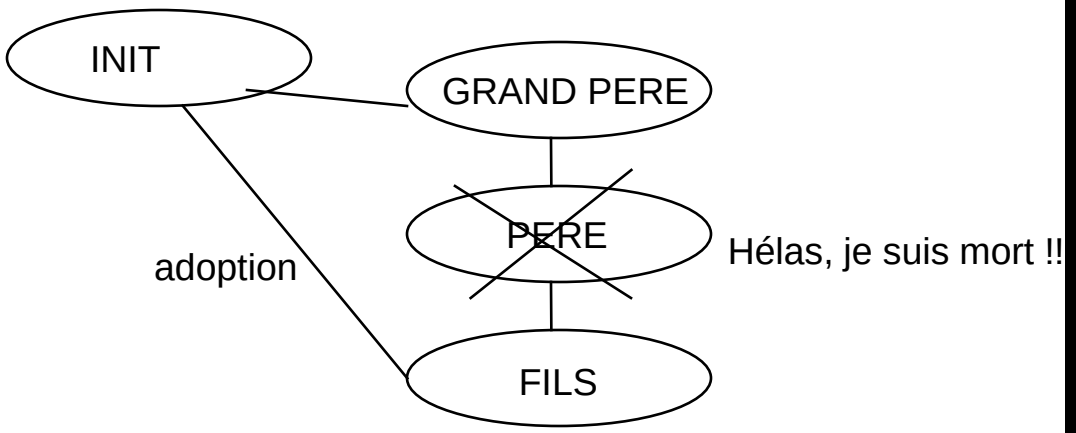
- **La communication entre le fils zombie et le père s'effectue par le biais d'un signal transmis du fils vers le père (signal SIGCHLD ou mort du fils)**

Génétique des processus Linux

Decès et adoption

- **1. Un processus fils défunt reste zombie jusqu'à ce que son contexte soit totalement détruit.**

- **2. 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).**



Génétique des processus Linux

Cas 1

```
linux-9bxb:~ # ./essai &
[1] 3997
linux-9bxb:~ # ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  3432  3368  0  80   0 -  3309 wait   pts/1    00:00:00 bash
0 S     0  3997  3432  0  80   0 -  1022 -      pts/1    00:00:00 essai
1 S     0  4000  3997  0  80   0 -  1022 -      pts/1    00:00:00 essai
0 R     0  4001  3432  0  80   0 -  3932 -      pts/1    00:00:00 ps
linux-9bxb:~ # kill 4000
linux-9bxb:~ # ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  3432  3368  0  80   0 -  3309 wait   pts/1    00:00:00 bash
0 S     0  3997  3432  0  80   0 -  1022 -      pts/1    00:00:00 essai
1 Z     0  4000  3997  0  80   0 -     0 exit   pts/1    00:00:00 essai <defunct>
0 R     0  4006  3432  0  80   0 -  3932 -      pts/1    00:00:00 ps
```

Génétique des processus Linux

Cas 2

```
linux-9bxb:~ # ./essai &
[2] 3957
linux-9bxb:~ # ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  3432  3368  0  80   0 -  3309 wait   pts/1    00:00:00 bash
0 S     0  3948  3432  0  80   0 -  1021 -      pts/1    00:00:00 essaibis
0 S     0  3957  3432  0  80   0 -  1022 -      pts/1    00:00:00 essai
1 S     0  3960  3957  0  80   0 -  1022 -      pts/1    00:00:00 essai
0 R     0  3961  3432  0  80   0 -  3932 -      pts/1    00:00:00 ps
linux-9bxb:~ # kill 3957
[2]+  Terminated              ./essai
linux-9bxb:~ # ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  3432  3368  0  80   0 -  3309 wait   pts/1    00:00:00 bash
0 S     0  3948  3432  0  80   0 -  1021 -      pts/1    00:00:00 essaibis
1 S     0  3960     1  0  80   0 -  1022 -      pts/1    00:00:00 essai
0 R     0  3967  3432  0  80   0 -  3932 -      pts/1    00:00:00 ps
```

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;
```

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

```
pid= fork();

if (pid == 0)
{
    printf(" je suis le fils ");
    execl("/home/calcul","calcul","3","4", NULL);
```

executable *paramètres*

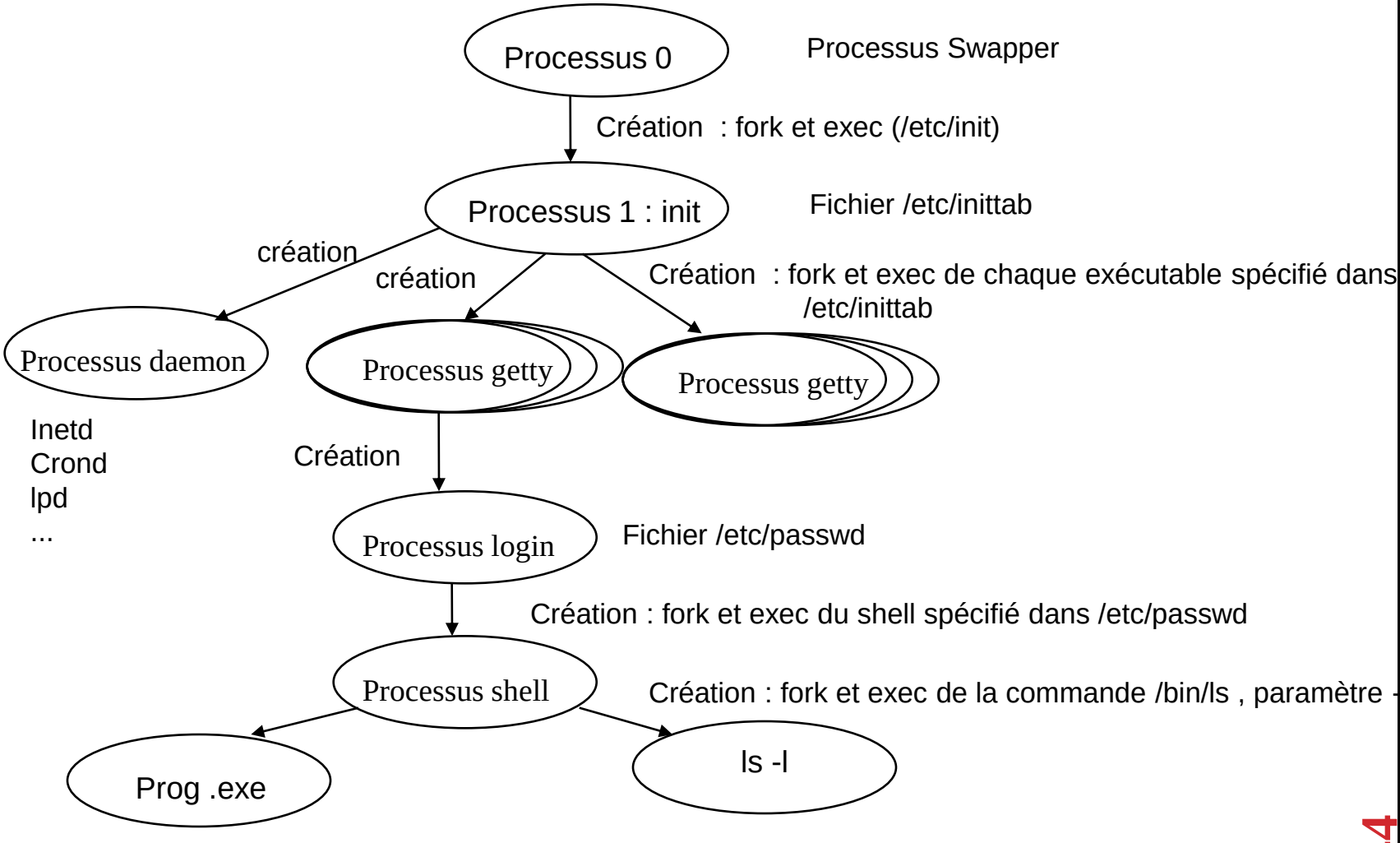
Le processus fils recouvre le code et les données hérités du père par ceux du programme calcul.
Le père transmet des données de son environnement vers son fils par les paramètres de l'exec

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

}
```

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)

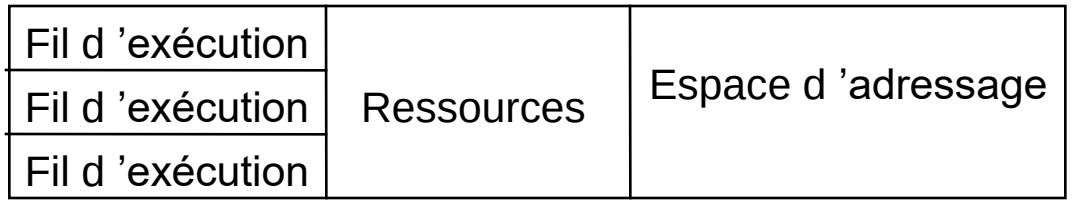
Notion de processus léger

- **Définition**

  - **Extension du modèle traditionnel de processus**

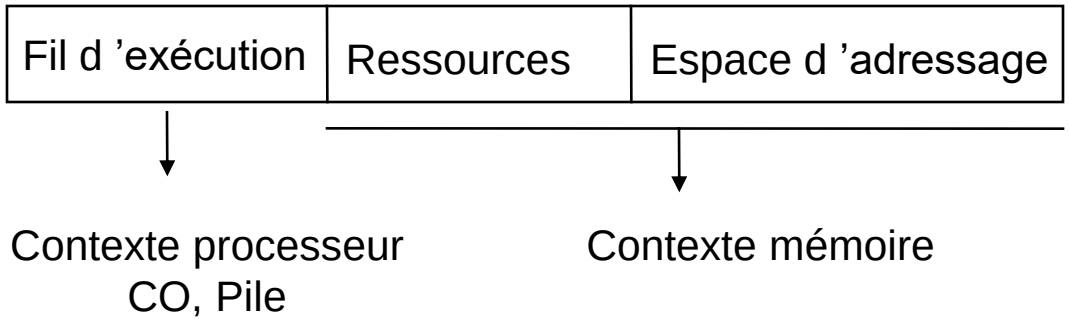
  - **Un thread ou processus léger est un fil d ’exécution au sein d ’un processus. On peut avoir plusieurs fils d ’exécution au sein du processus.**

Processus classique

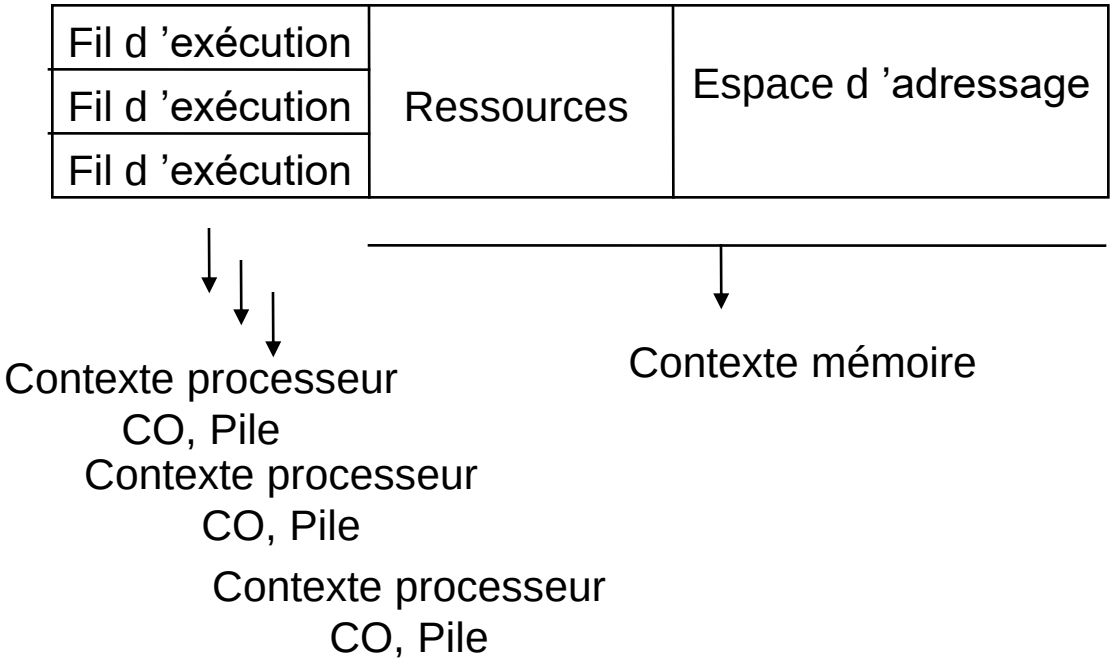


Processus à threads

Notion de processus léger



Processus classique



Processus à threads

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**

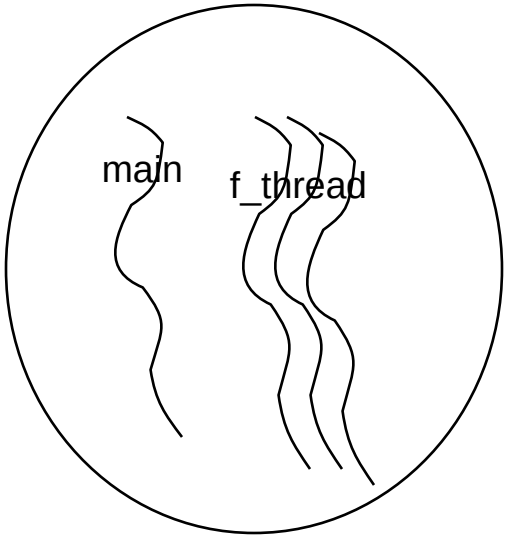
int pthread_join (pthread_t thread, void **value_ptr);

  - **terminaison de threads**

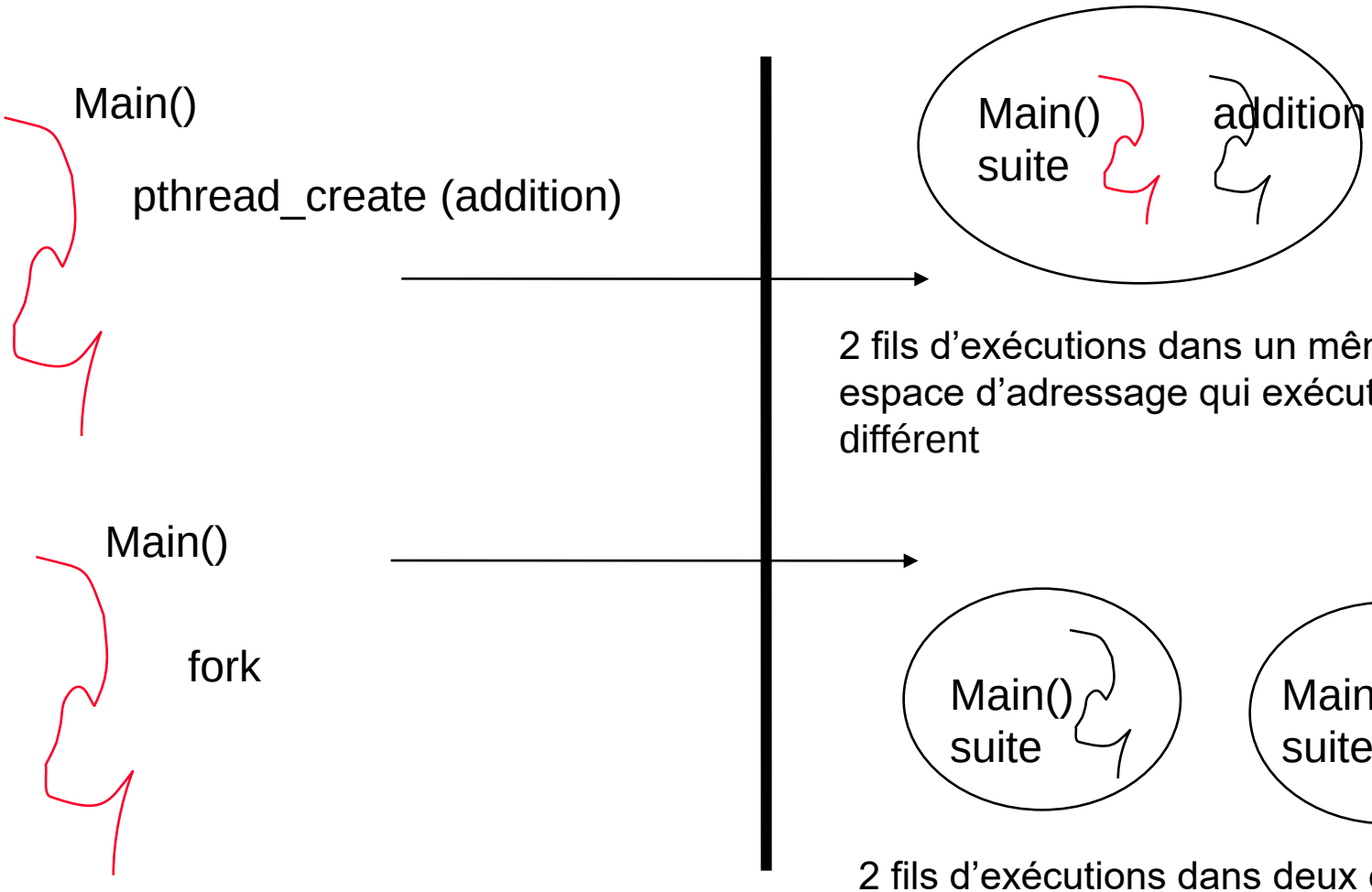
int pthread_exit (void **value_ptr);

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();
}
```

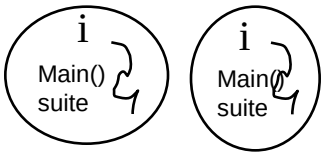


Notion de processus léger



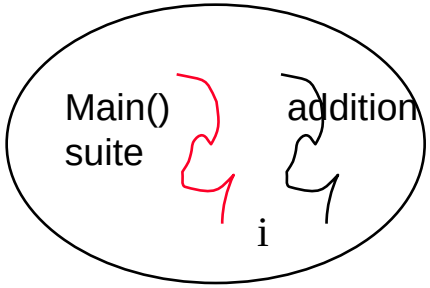
Notion de processus léger

```
#include <stdio.h>
int i;
main()
{
int pid;
i = 0;
pid = fork();
if (pid == 0)
{ i = i + 10;
printf ("hello,  fils %d\n", i);
i = i + 20;
printf ("hello, fils %d\n", i); }
else
{ i = i + 1000;
printf ("hello, père %d\n", i);
i = i + 2000;
printf ("hello, père %d\n", i);
wait();}}
```



TRACES D'EXECUTION
hello, père 1000
hello, fils 10
hello, fils 30
hello, père 3000

```
#include <stdio.h>
#include <pthread.h>
int i;
void addition()
{
i = i + 10;
printf ("hello, thread fils %d\n", i);
i = i + 20;
printf ("hello, thread fils %d\n", i);
}
main()
{ pthread_t num_thread;
i = 0;
if (pthread_create(&num_thread, NULL, (void *(*)())addition, NULL) == -1)
    perror ("pb pthread_create\n");
i = i + 1000;
printf ("hello, thread principal %d\n", i);
i = i + 2000;
printf ("hello, thread principal %d\n", i);
pthread_join(num_thread, NULL);}
```



TRACES D'EXECUTION
hello, thread principal 1000
hello, thread fils 1010
hello, thread principal 3010
hello, thread fils 3030

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

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.