LINUX
processus et outils de communication

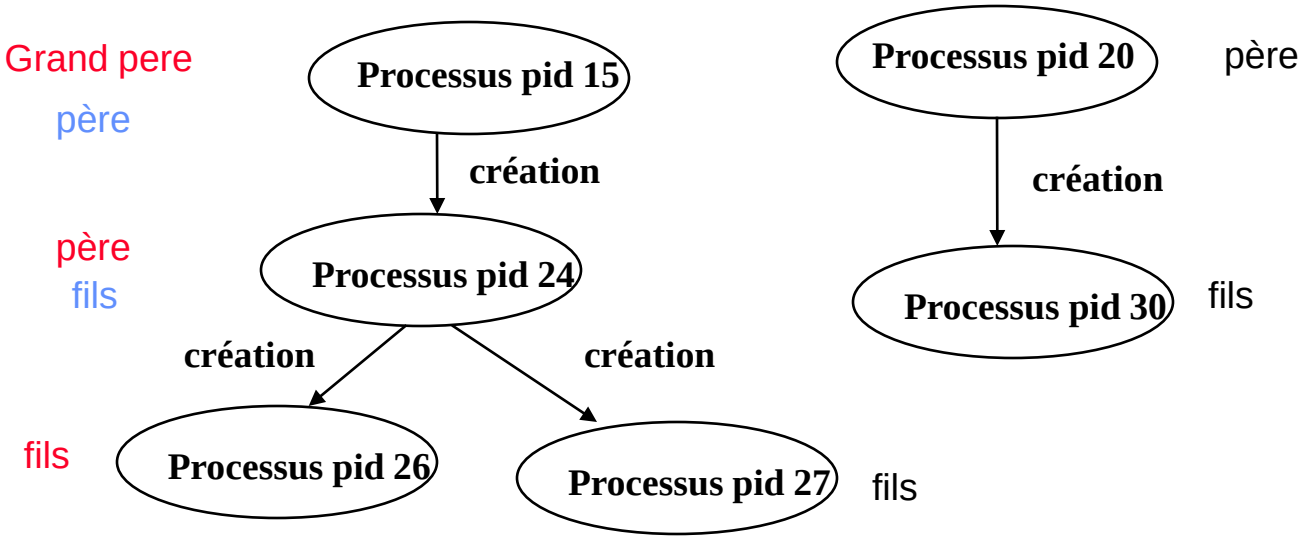
Processus
Processus LINUX

Caractéristiques Générales

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

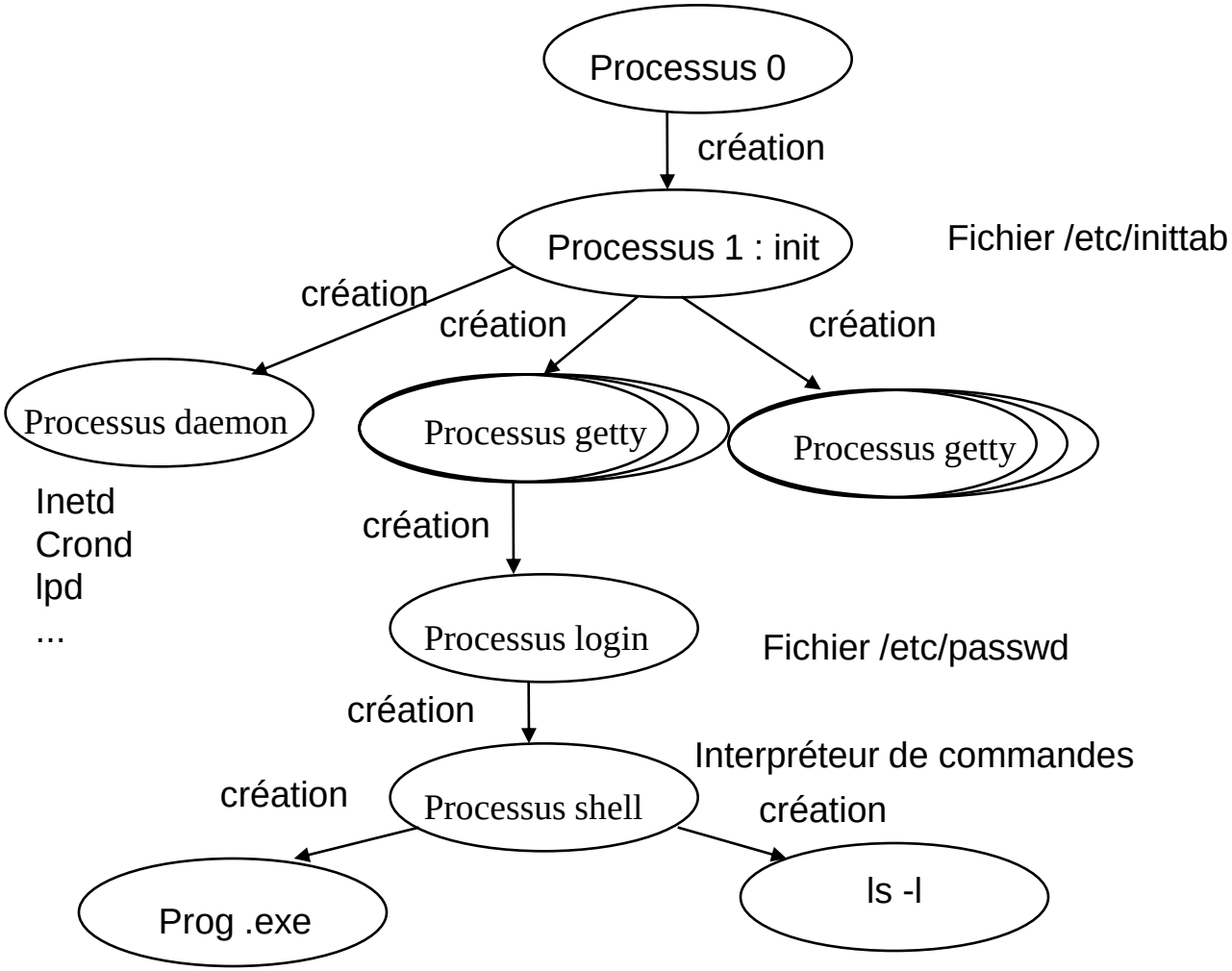
Processus Linux

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

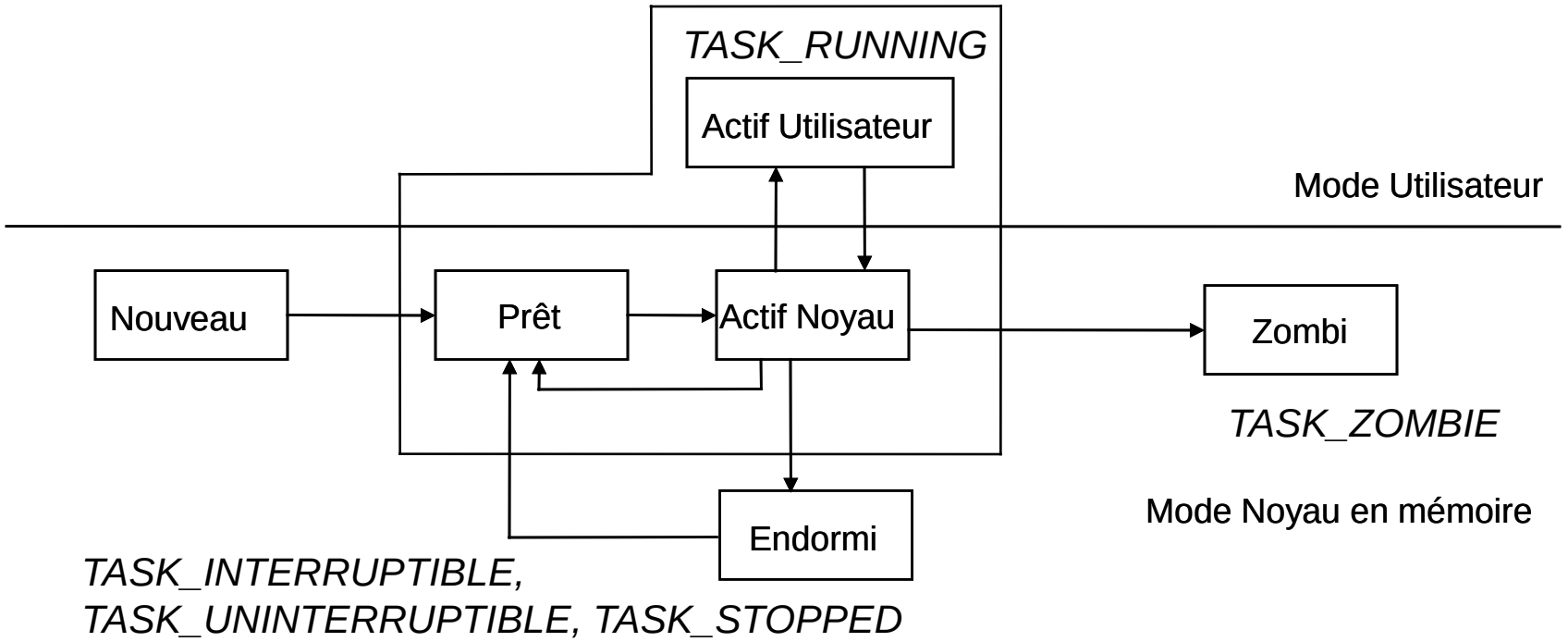


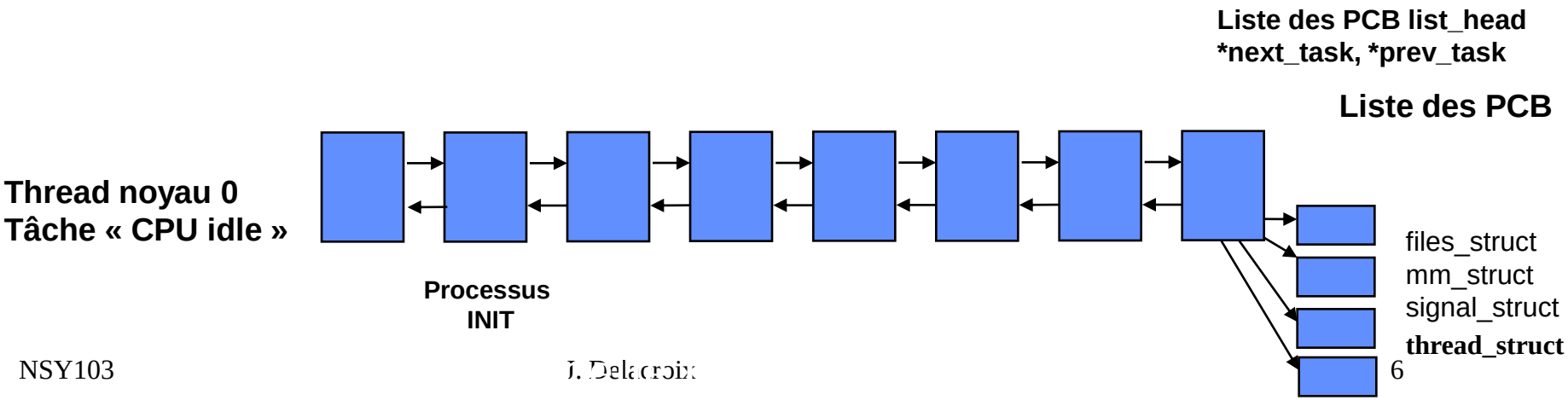
Processus Linux

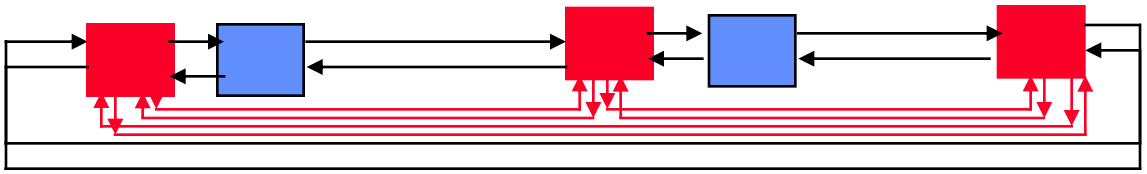
- **Tout le système Linux repose sur ce concept arborescent**



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)
```

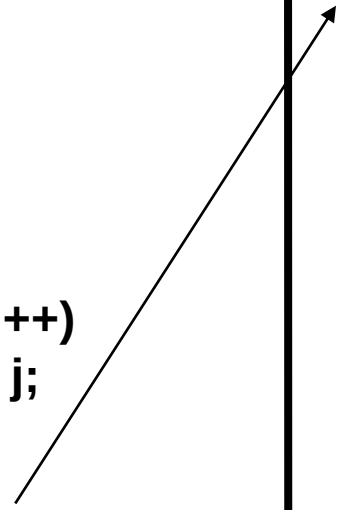
  - **La primitive fork() permet la création dynamique d ’un nouveau processus qui s ’exécute de manière concurrente avec le processus qui l ’a créé.**
  - **Tout processus Unix/Linux hormis le processus 0 est crée à l ’aide de cette primitive.**
  - **Le processus créateur (le père) par un appel à la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

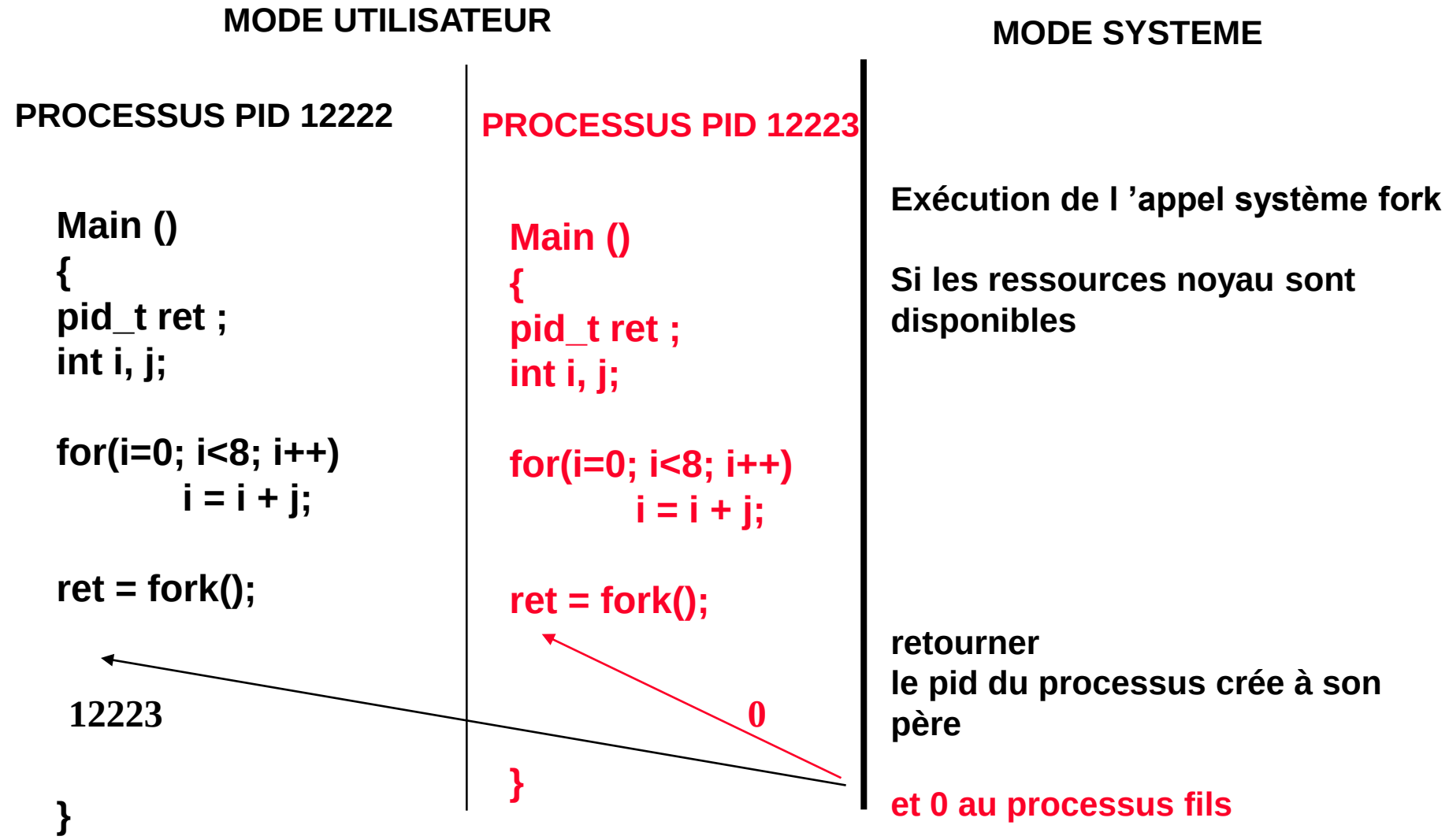
- allouer une entrée de la table des processus au nouveau processus

- allouer un pid unique au nouveau processus

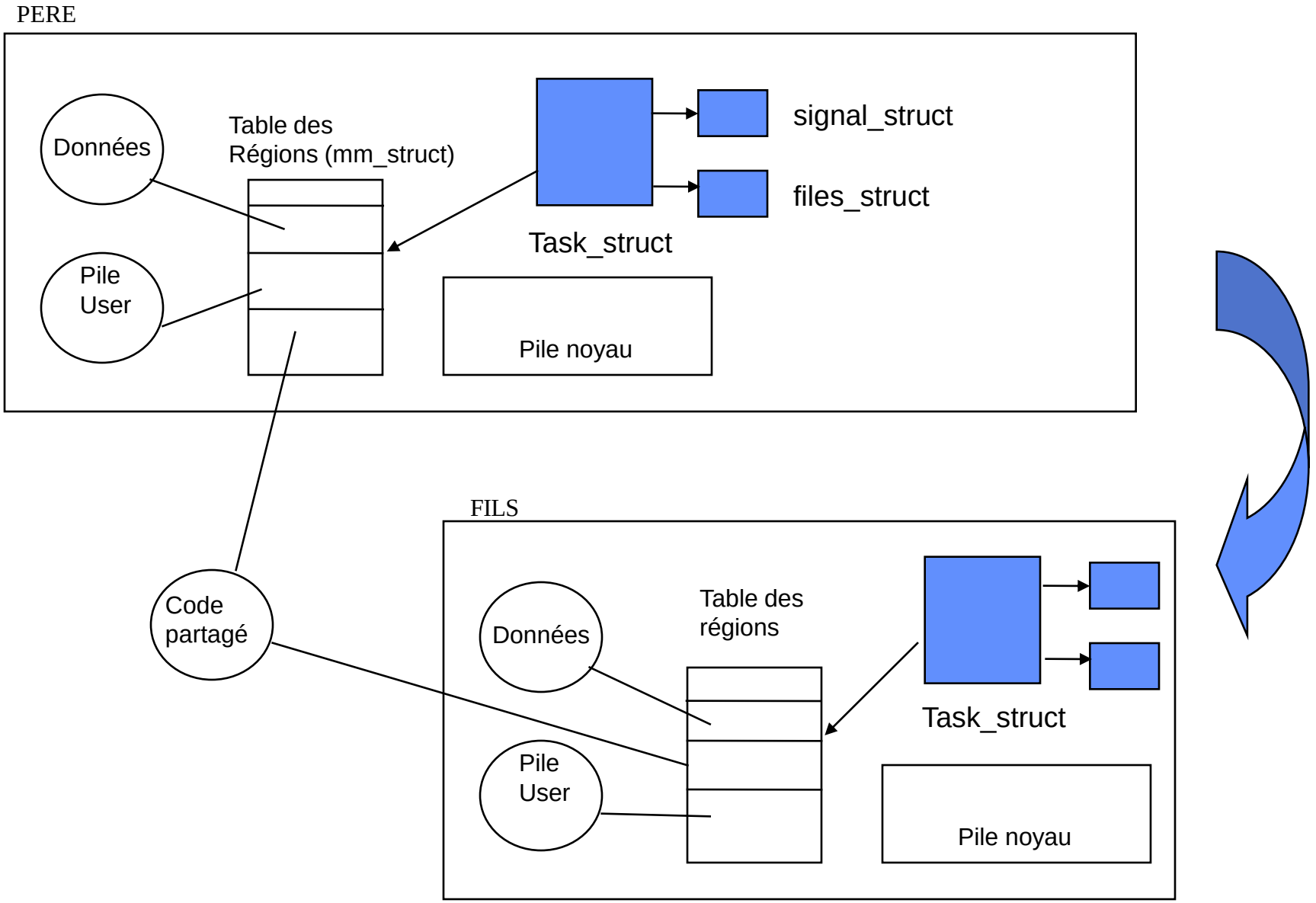
- dupliquer le contexte du processus parent (code, données, pile)

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

Primitive de création de processus



Primitive de création de processus



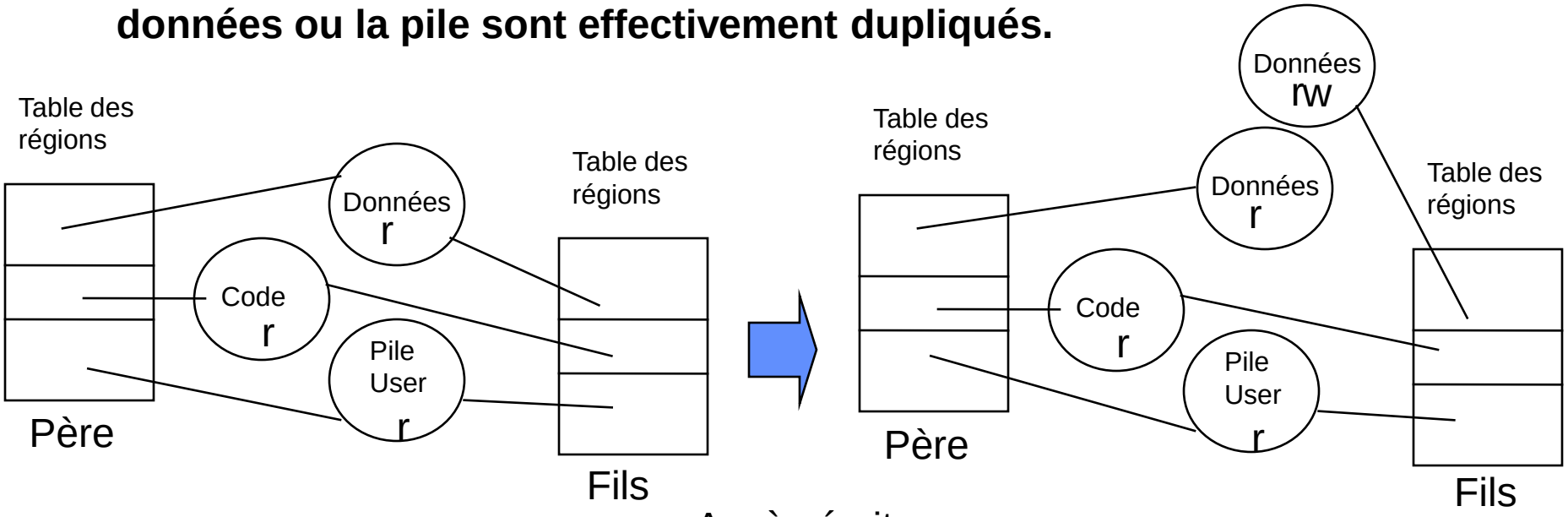
Primitive de création de processus

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

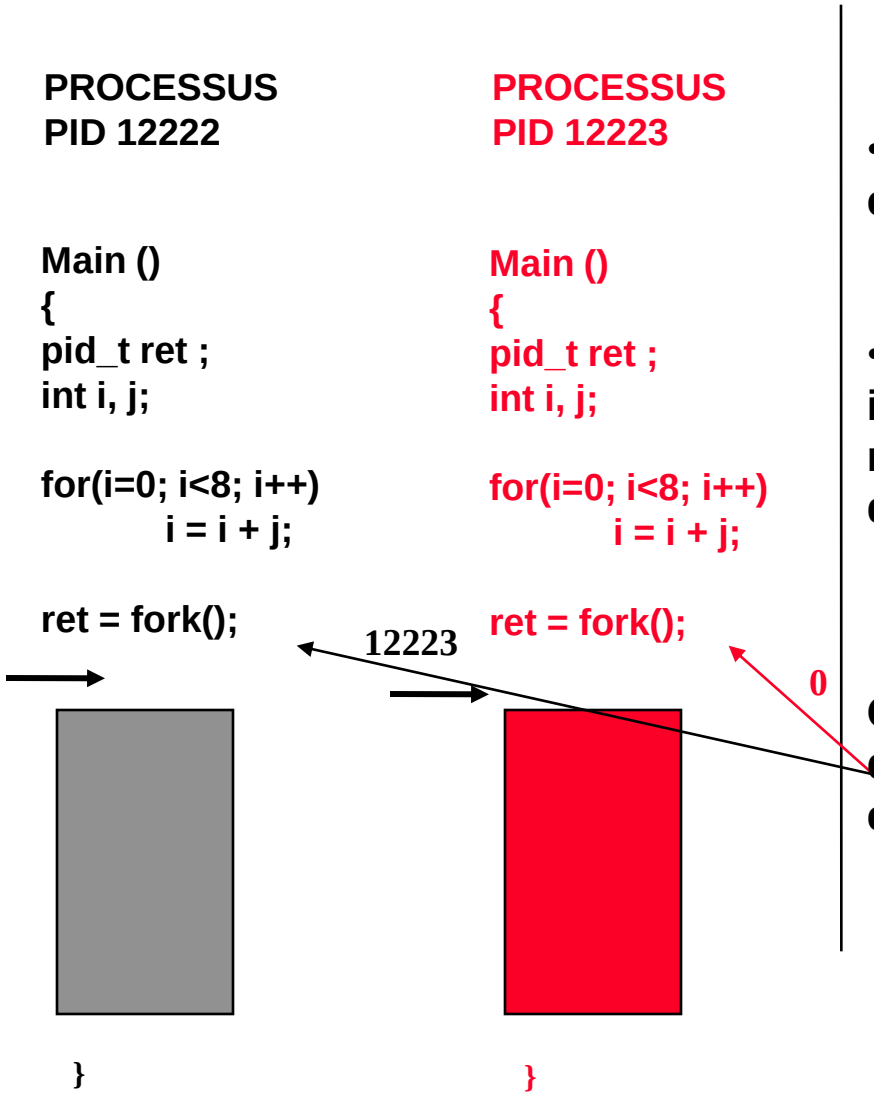
Lors de sa création, le fils partage avec son père:

- 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



- 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)

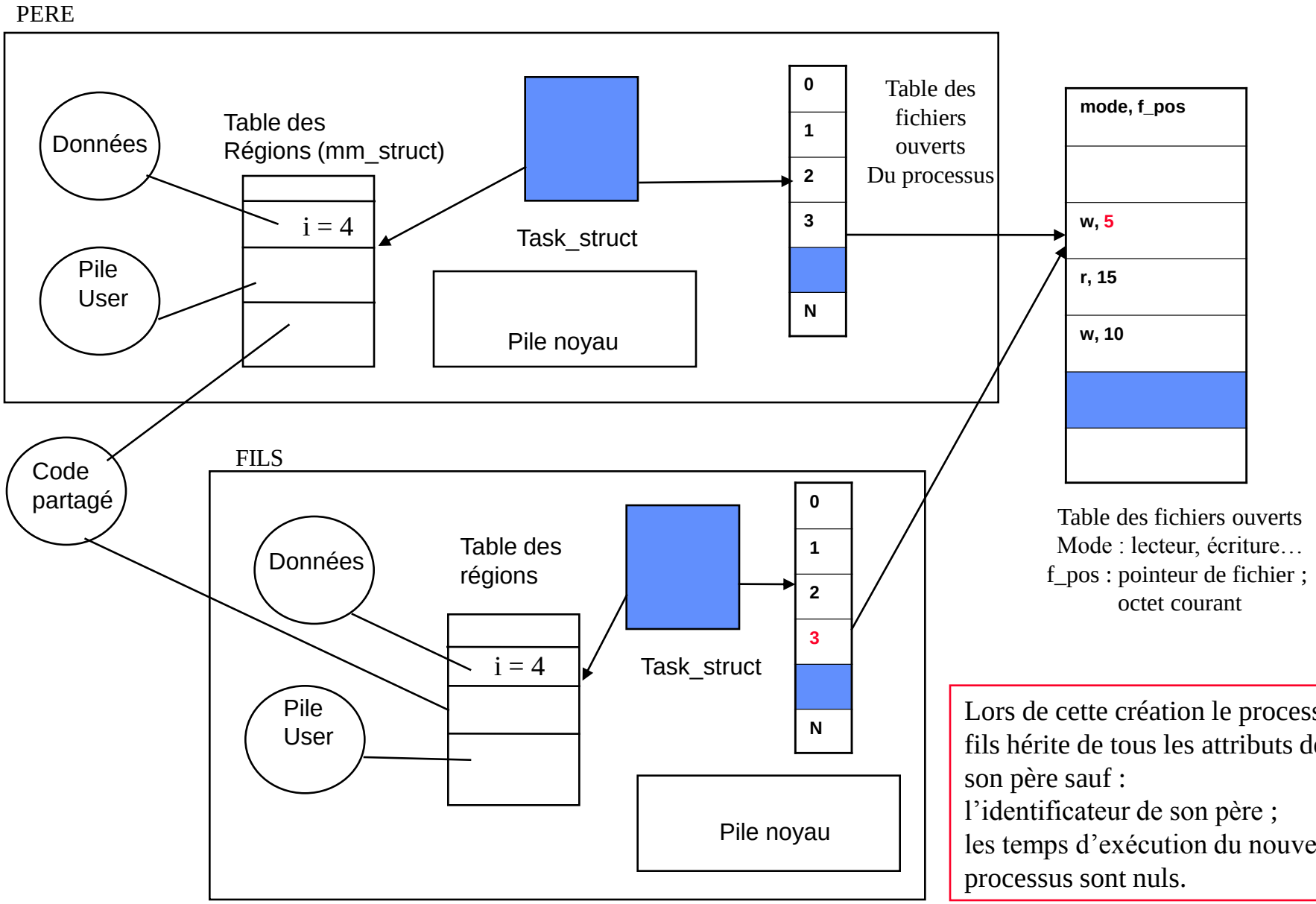


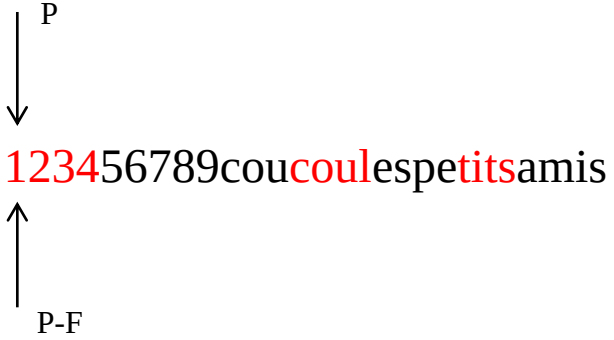
Table des fichiers ouverts
Mode : lecteur, écriture…
f_pos : pointeur de fichier ;
octet courant

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)

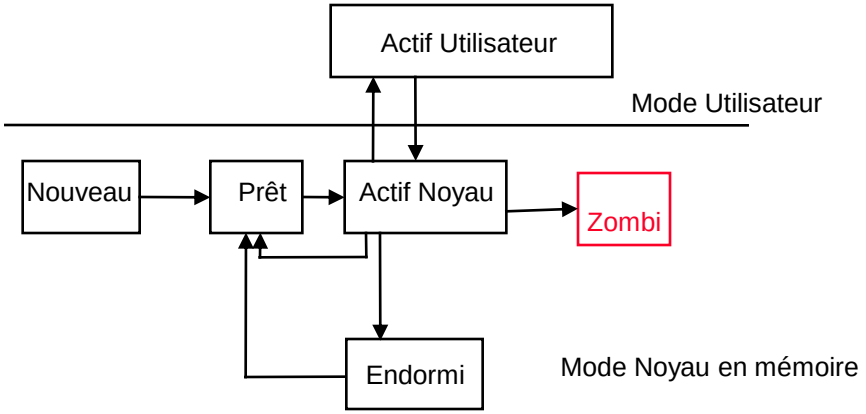
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


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.

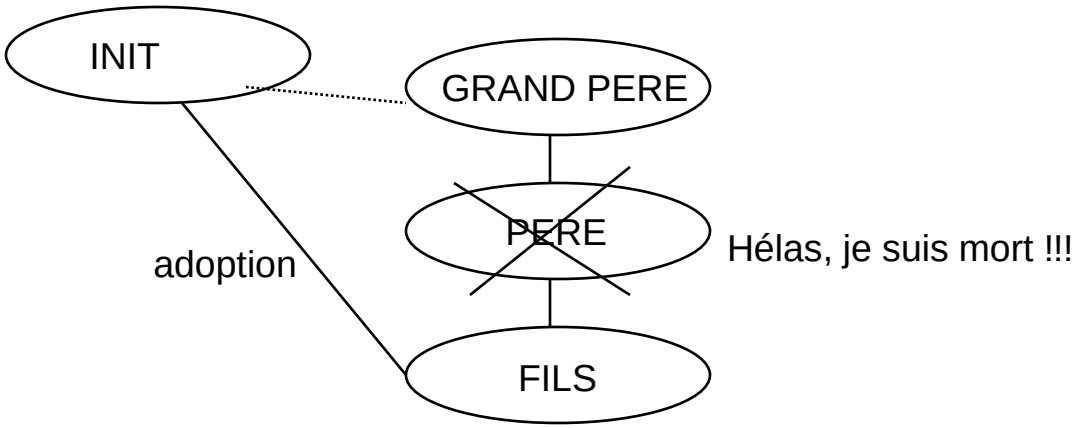
- **La communication entre le fils zombie et le père s'effectue par le biais d'un signal transmis du fils vers le père (signal SIGCHLD ou mort du fils)**

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;

pid= fork();

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

}
```

executable *paramètres*

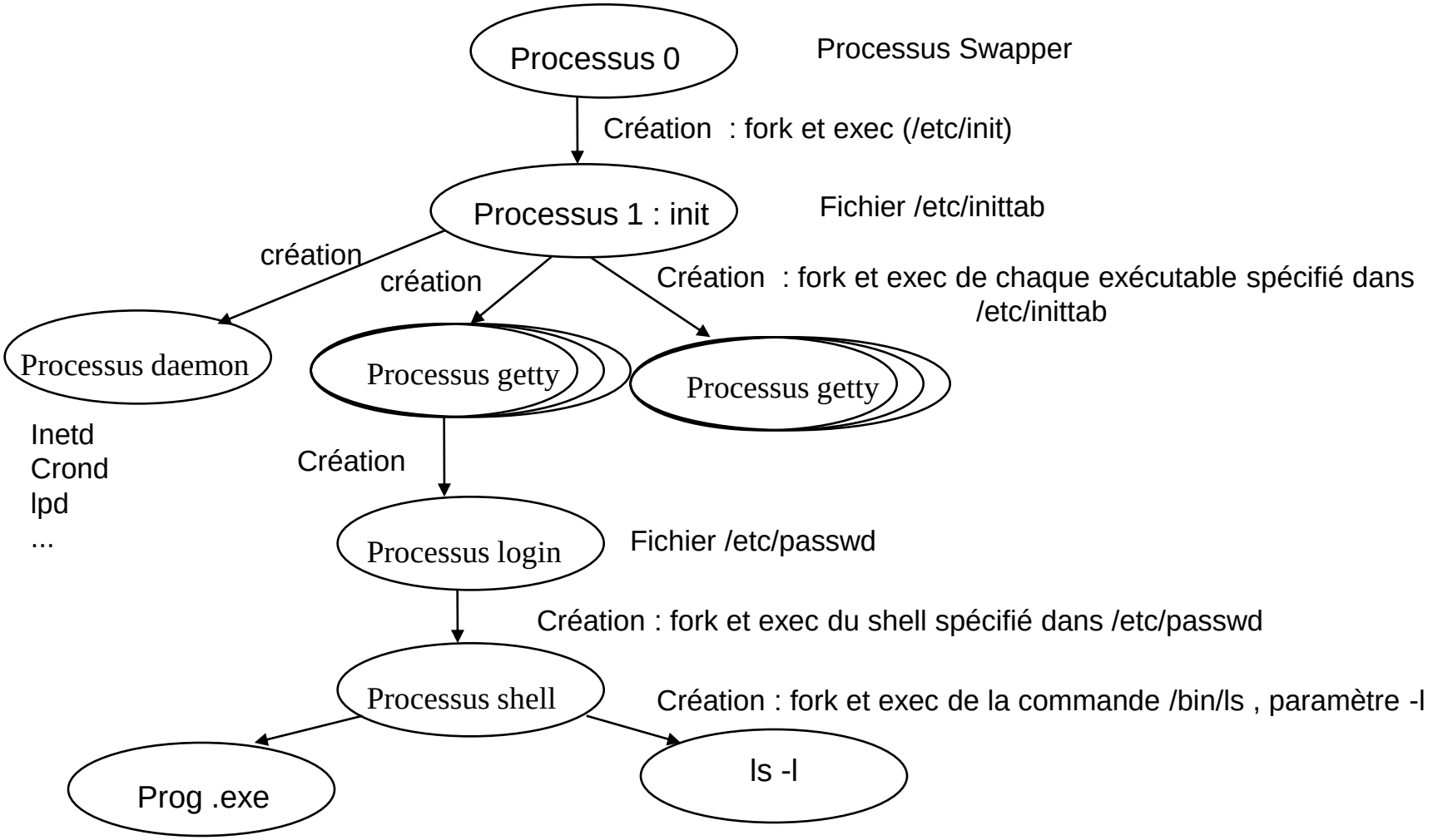
Création d'un processus fils par duplication du code et données du père

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)

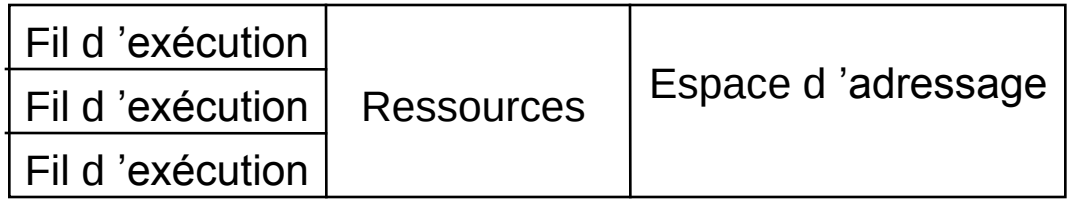
Notion de processus léger

- **Définition**

  - **Extension du modèle traditionnel de processus**

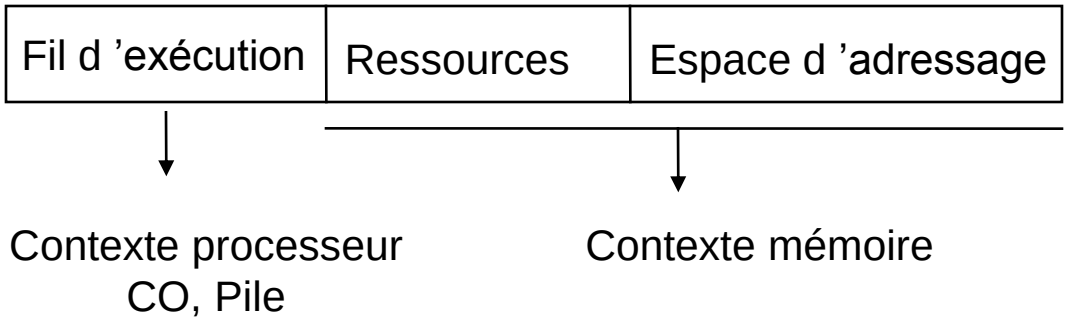
  - **Un thread ou processus léger est un fil d ’exécution au sein d ’un processus. On peut avoir plusieurs fils d ’exécution au sein du processus.**

Processus classique

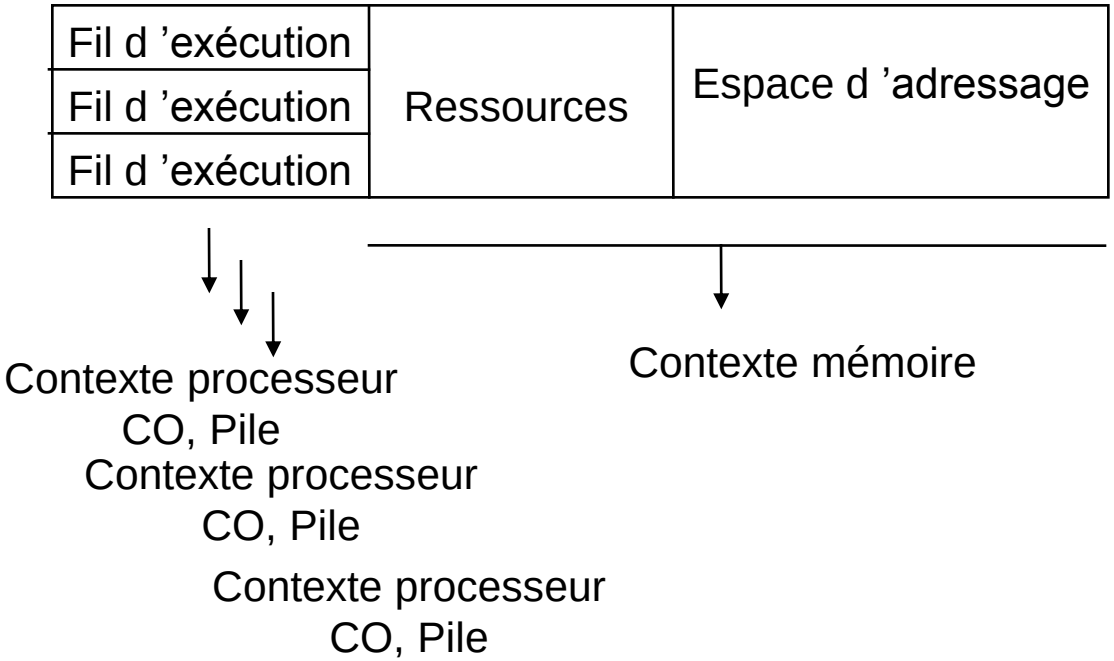


Processus à threads

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**

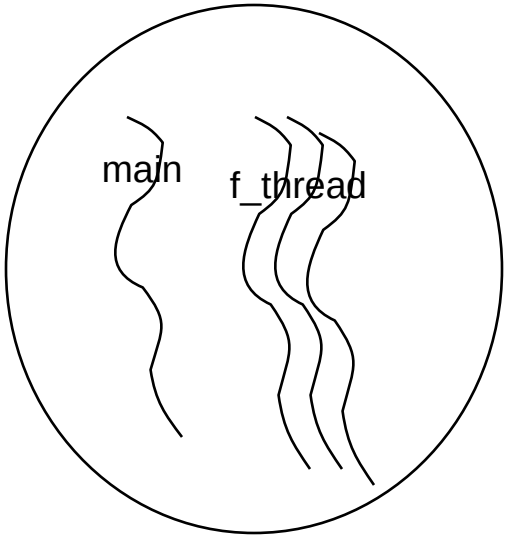
int pthread_join (pthread_t thread, void **value_ptr);

  - **terminaison de threads**

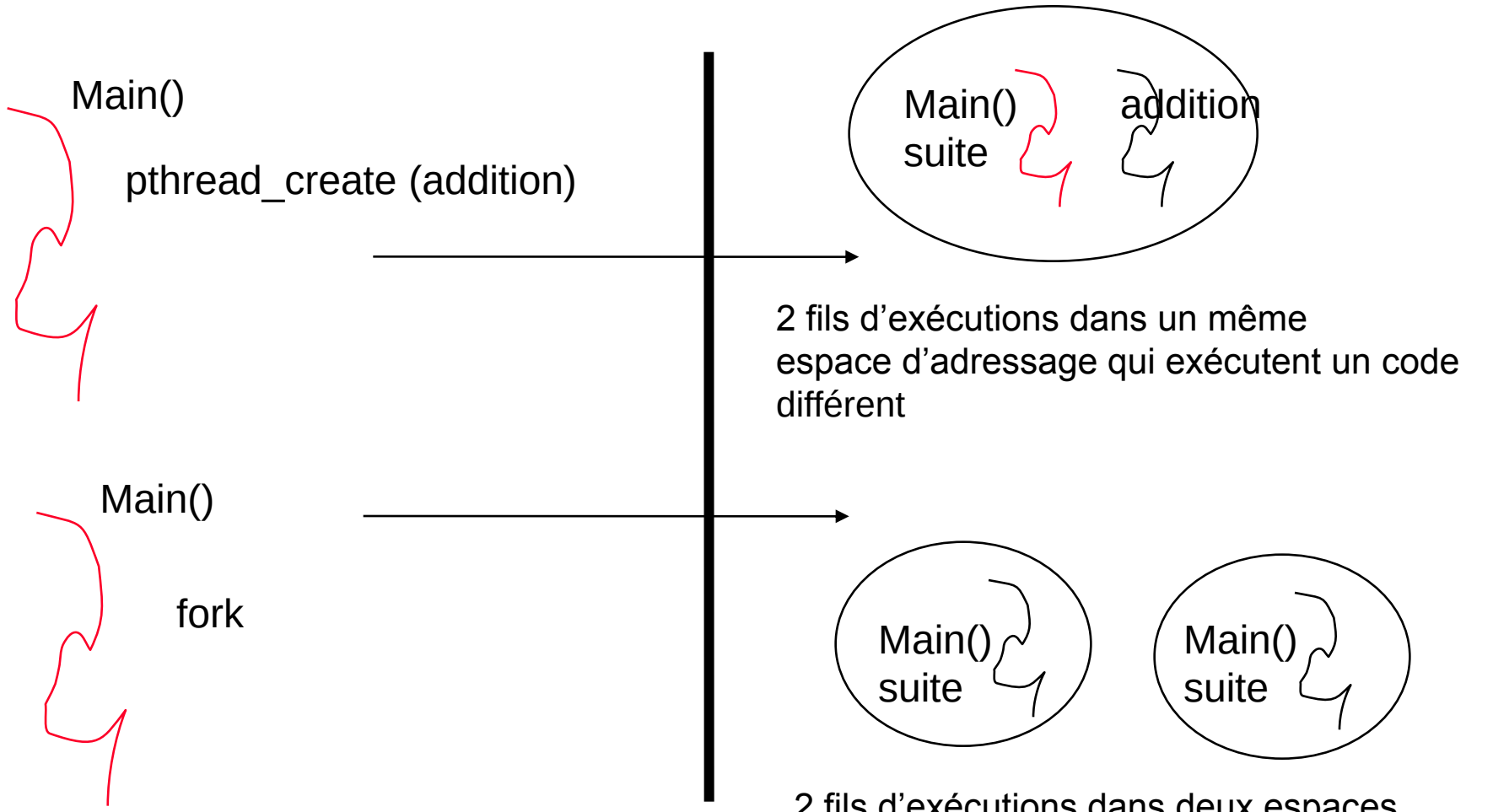
int pthread_exit (void **value_ptr);

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();
}
```

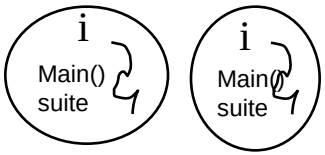


Notion de processus léger



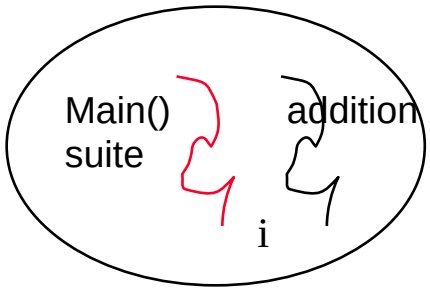
Notion de processus léger

```
#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

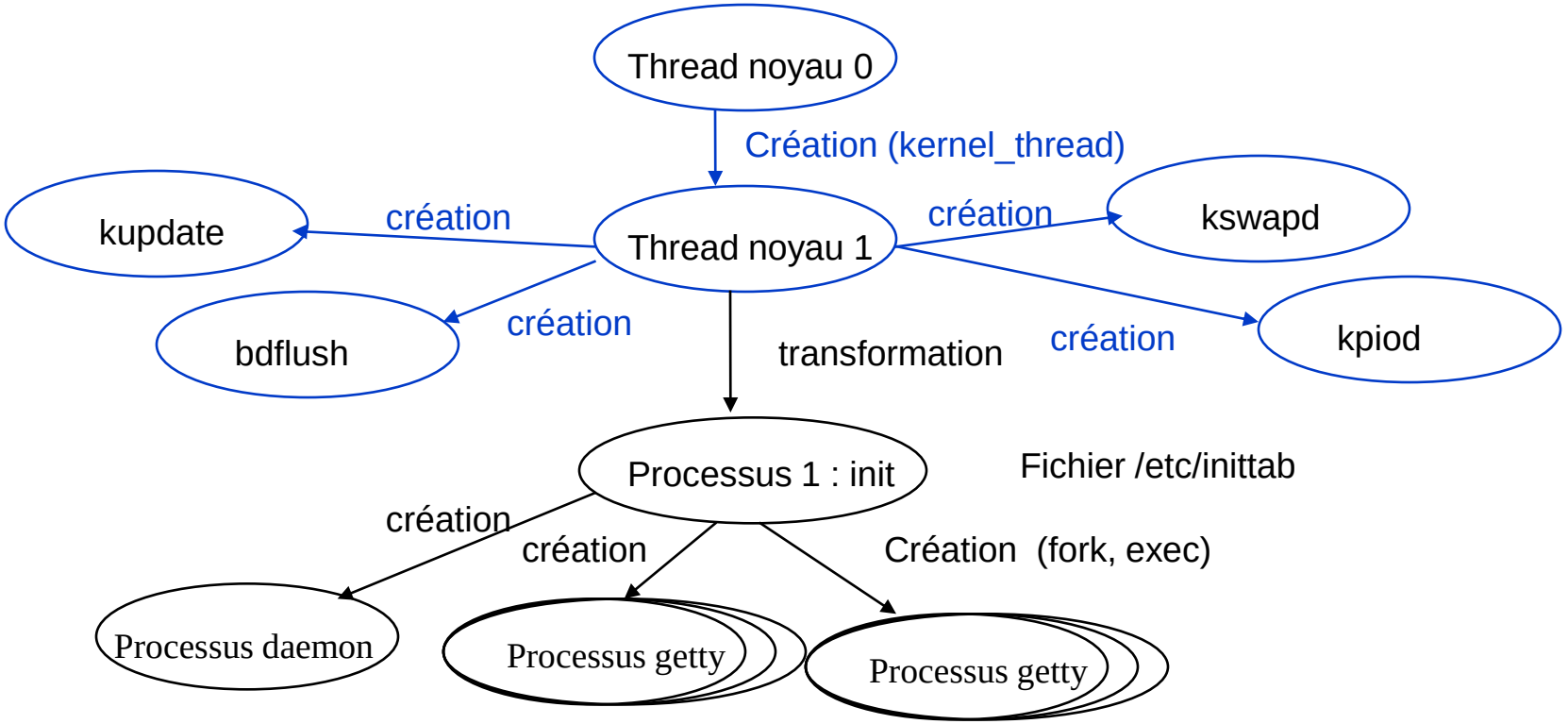
Communication : allégée
les fils d ’exécution d ’un même processus partagent les données, mais attention à la synchronisation

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.



Outils de communication entre processus Linux

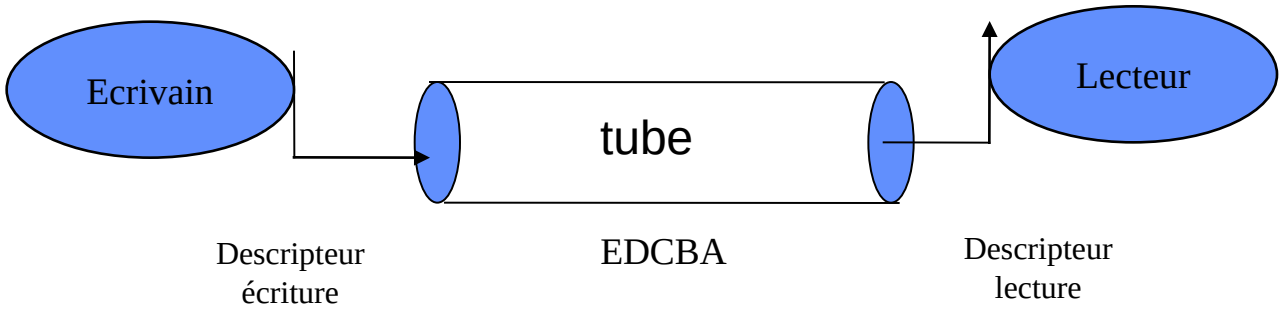
Deux familles

- **Les outils gérés avec le SGF : les tubes et les sockets**
- **Les outils IPC gérés dans des tables du systèmes et repérés par une clef : les MSQ, les segments de mémoire partagée, les sémaphores**

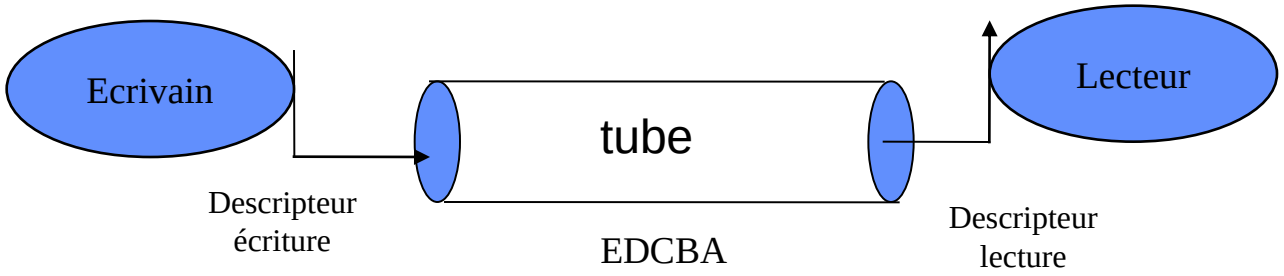
Outils de communication entre processus Linux

LES TUBES

- **Outils gérés avec le SGF, accessibles via des descripteurs**
- **Communication unidirectionnelle entre processus, en mode FIFO**
- **Tubes anonymes : entre processus d'une même famille**
- **Tubes nommés : entre n'importe quel processus**



Les tubes : les tubes anonymes



LES TUBES Anonymes

- **Fichier sans nom, seulement accessible par les deux descripteurs**
- **Seuls les processus ayant connaissance par héritage des descripteurs peuvent utiliser le tube → processus créateur et ses descendants**

Les tubes : les tubes anonymes
CREATION

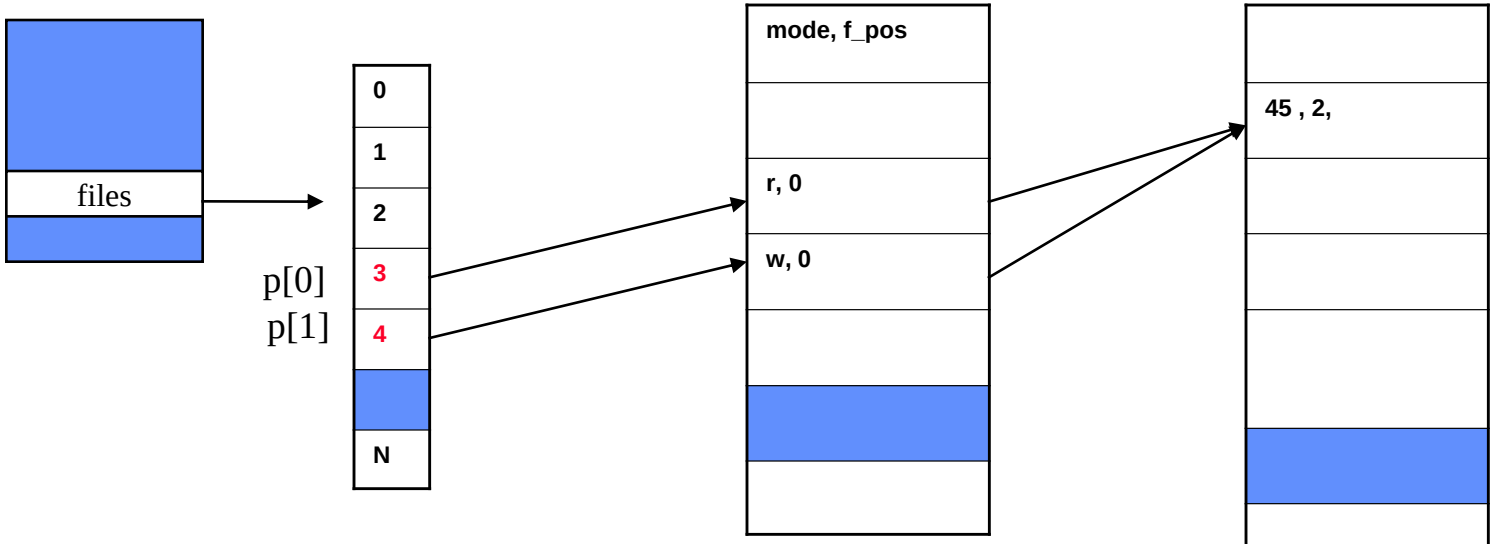
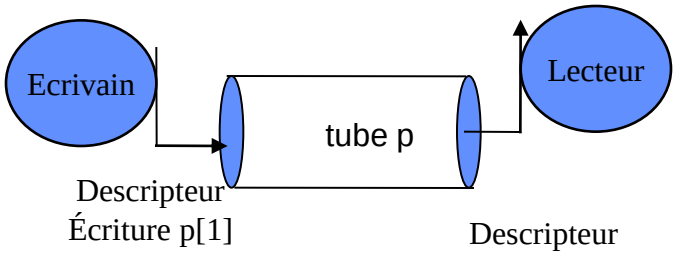


Table des fichiers ouverts
Du processus

Table des fichiers ouverts
Mode : lecteur, écriture…

f_pos : pointeur de fichier ; octet courant

Table des inodes mémoires
f_count : nombre de référence
au même fichier




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

Les tubes : les tubes anonymes

Exemple de communication entre processus par l'intermédiaire d'un tube. Un processus fils écrit à destination de son père la chaîne de caractères « bonjour ».

1. Le processus père ouvre un tube en utilisant la fonction pipe().

2. Le processus père crée un fils en utilisant la fonction fork().

3. Les descripteurs en lecture et écriture du tube sont utilisables par les 2 processus. Chacun des deux processus ferme le descripteur qui lui est inutile : ainsi, le processus père ferme le descripteur en écriture et le processus fils ferme le descripteur en lecture.

4. Le fils envoie un message à son père.

5. Le père lit le message.

```
/*****************************************************************/
/*    Communication unidirectionnelle entre un père et son fils    */
/*****************************************************************/
main () {
int p[2], pid_t retour;
char chaine[7];
pipe(p);
retour = fork();
if (retour == 0)
{ /* le fils écrit dans le tube */
      close (p[0]); /* pas de lecture sur le tube */
      write (p[1], "bonjour", 7);
      close (p[1]); exit(0); }
else
{ /* le père lit le tube */
      close (p[1]); /* pas d'écriture sur le tube */
      read(p[0], chaine, 7);
      close (p[0]);
      wait;
}
```

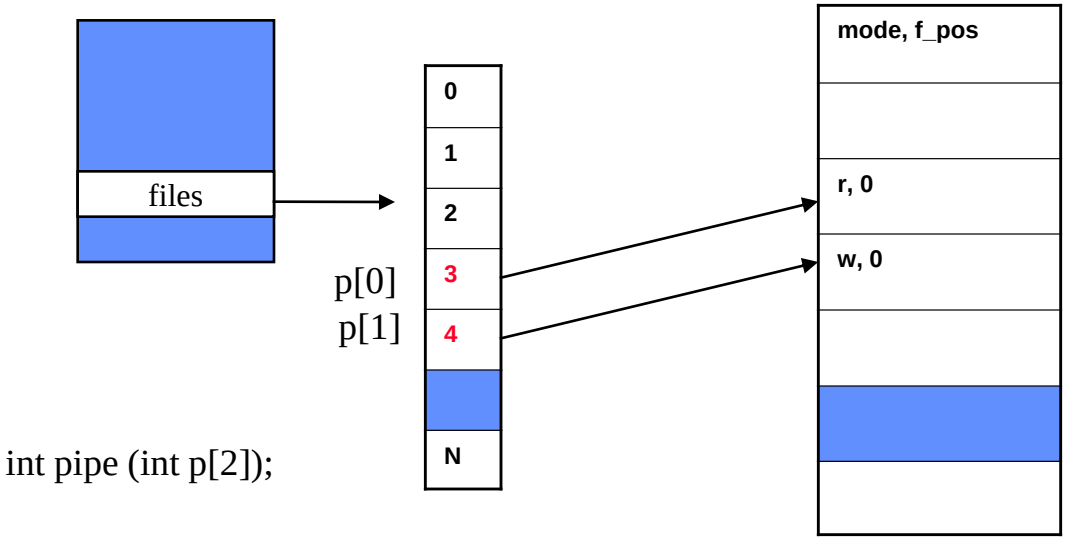
Les tubes : les tubes anonymes

Exemple de communication entre processus par l'intermédiaire d'un tube. Un processus fils écrit à destination de son père la chaîne de caractères « bonjour ».

1. Le processus père ouvre un tube en utilisant la fonction pipe().

2. Le processus père crée un fils en utilisant la fonction fork().

3. Les descripteurs en lecture et écriture du tube sont utilisables par les 2 processus. Chacun des deux processus ferme le descripteur qui lui est inutile : ainsi, le processus père ferme le descripteur en écriture et le processus fils ferme le descripteur en lecture.

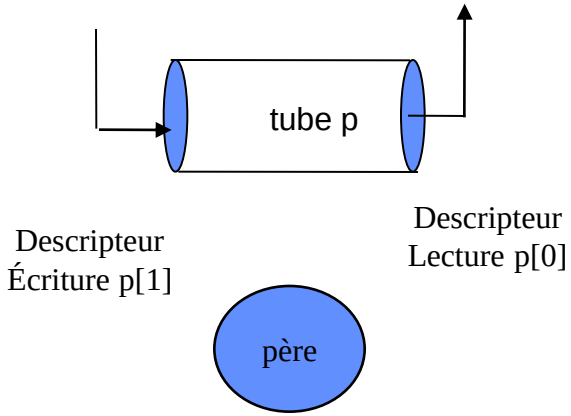


int pipe (int p[2]);

Table des fichiers ouverts
Du processus

Table des fichiers ouverts
Mode : lecteur, écriture…

f_pos : pointeur de fichier ; octet courant



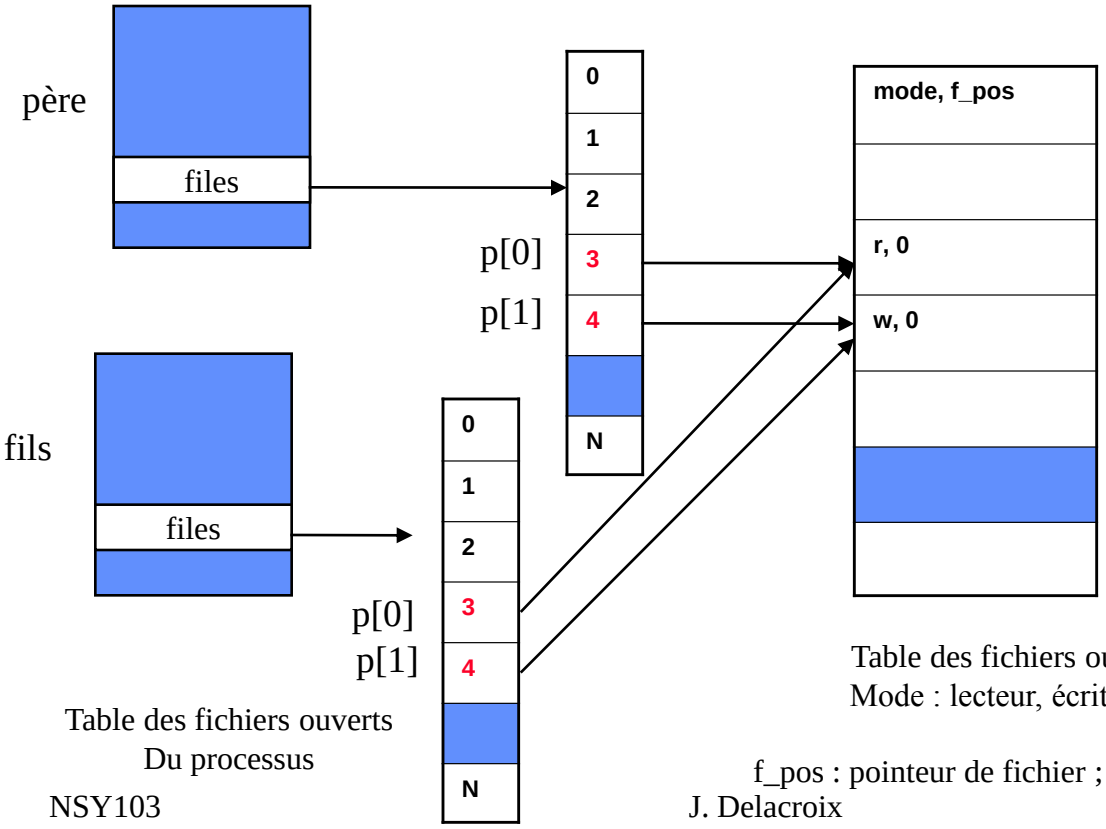
Les tubes : les tubes anonymes

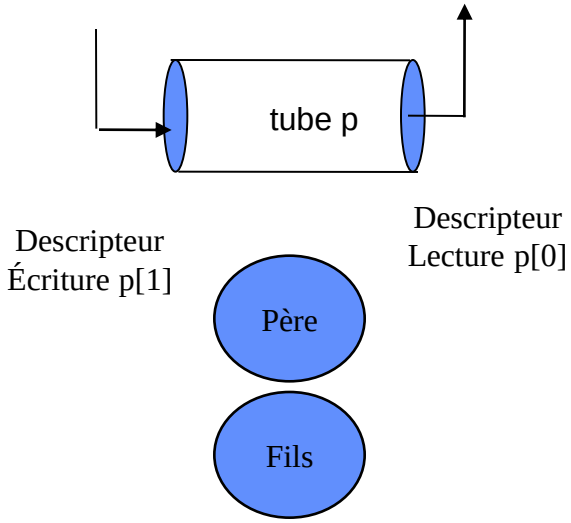
Exemple de communication entre processus par l'intermédiaire d'un tube. Un processus fils écrit à destination de son père la chaîne de caractères « bonjour ».

1. Le processus père ouvre un tube en utilisant la fonction pipe().

2. Le processus père crée un fils en utilisant la fonction fork().

3. Les descripteurs en lecture et écriture du tube sont utilisables par les 2 processus. Chacun des deux processus ferme le descripteur qui lui est inutile : ainsi, le processus père ferme le descripteur en écriture et le processus fils ferme le descripteur en lecture.





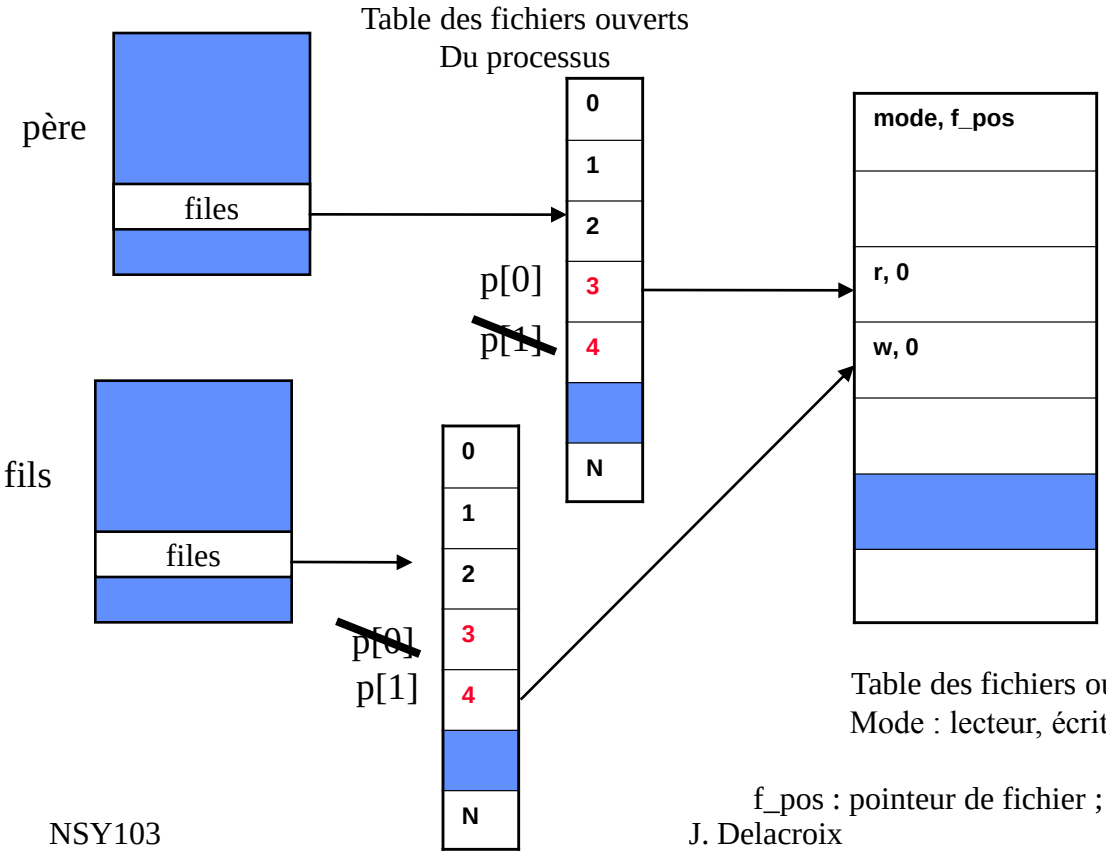
Les tubes : les tubes anonymes

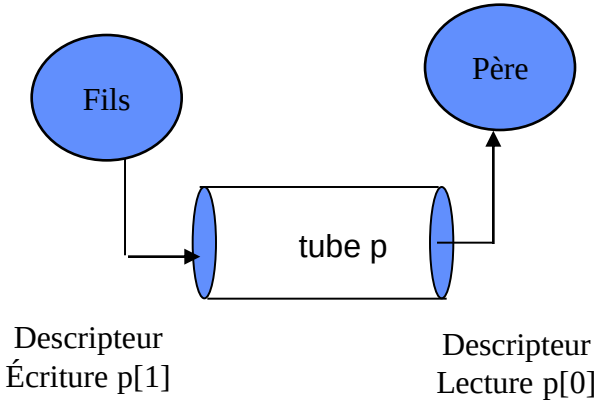
Exemple de communication entre processus par l'intermédiaire d'un tube. Un processus fils écrit à destination de son père la chaîne de caractères « bonjour ».

1. Le processus père ouvre un tube en utilisant la fonction pipe().

2. Le processus père crée un fils en utilisant la fonction fork().

3. Les descripteurs en lecture et écriture du tube sont utilisables par les 2 processus. Chacun des deux processus ferme le descripteur qui lui est inutile : ainsi, le processus père ferme le descripteur en écriture et le processus fils ferme le descripteur en lecture.





Les tubes : les tubes anonymes
PRIMITIVES

Fermeture de descripteur

Un processus ferme un descripteur de tube fd en utilisant la primitive close() :

int close(int fd);

Le nombre de descripteurs ouverts en lecture détermine le nombre de lecteurs existants pour le tube.

Le nombre de descripteurs ouverts en écriture détermine le nombre d'écrivains existants pour le tube.

Un descripteur fermé ne permet plus d'accéder au tube et ne peut pas être régénéré.

Les tubes : les tubes anonymes
PRIMITIVES

Lecture

La lecture s'effectue par le biais de la primitive read()

int read(int desc[0], char *buf, int nb);

La primitive permet la lecture de nb caractères depuis le tube desc, qui sont placés dans le tampon buf. Elle retourne en résultat le nombre de caractères réellement lus.

L'opération de lecture répond à la sémantique suivante :

- **si le tube n'est pas vide et contient taille caractères, la primitive extrait du tube min(taille, nb) caractères qui sont lus et placés à l'adresse buf ;**
- **si le tube est vide et que le nombre d'écrivains est non nul, la lecture est bloquante. Le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide ;**
- **si le tube est vide et que le nombre d'écrivains est nul, la fin de fichier est atteinte. Le nombre de caractères rendu est nul.**

Les tubes : les tubes anonymes
PRIMITIVES

Ecriture

L'écriture dans un tube anonyme s'effectue par le biais de la primitive write()

int write(int desc[1], char *buf, int nb);

La primitive permet l'écriture de nb caractères placés dans le tampon buf dans le tube desc. Elle retourne en résultat le nombre de caractères réellement écrits.

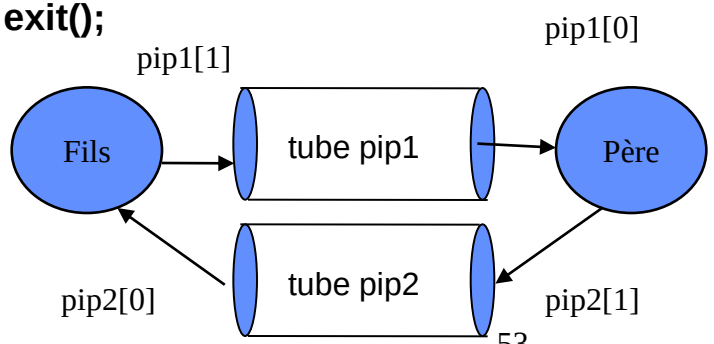
L'opération d'écriture répond à la sémantique suivante :

- **si le nombre de lecteurs dans le tube est nul, alors une erreur est générée et le signal SIGPIPE est délivré au processus écrivain, et le processus se termine. L'interpréteur de commandes shell affiche par défaut le message « Broken pipe » ;**

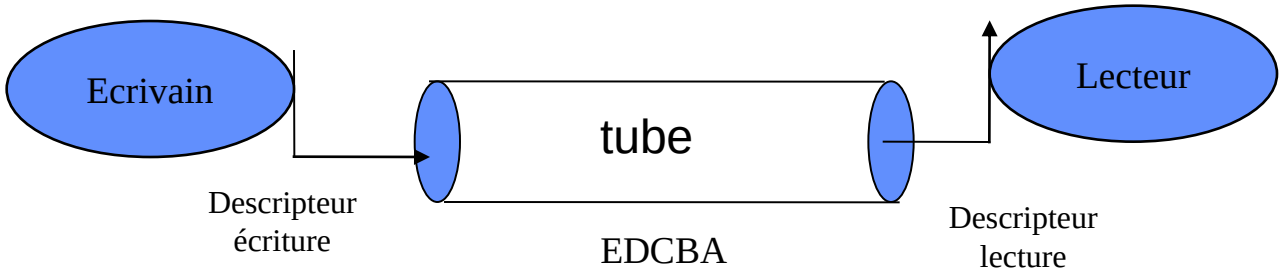
- **si le nombre de lecteurs dans le tube est non nul, l'opération d'écriture est bloquante jusqu'à ce que les nb caractères aient effectivement été écrits dans le tube.**

Les tubes : les tubes anonymes
Communication bidirectionnelle

```
/*****************************************************************/
/*    Communication bidirectionnelle entre un père et son fils    */
/*****************************************************************/
#include <stdio.h>
int pip1[2];       /*  descripteurs pipe 1  */
int pip2[2];       /*  descripteurs pipe 2  */
main()
{
int idfils;
char rep[7], mesg[5];
/* ouverture tubes */
 (pipe(pip1))
 (pipe(pip2))
/* création processus */
idfils=fork()
if(idfils) {
        /*le premier tube sert dans le sens père vers fils
          il est fermé en lecture */
       close(pip1[0]);
       /*le second tube sert dans le sens fils vers père
          il est fermé en écriture*/
       close(pip2[1]);
       /* on envoie un message au fils par le tube 1*/
         write(pip1[1],"hello",5)
       /* on attend la réponse du fils par le tube 2
          */
         iread(pip2[0],rep,7);
         printf("message du fils: %s\n",rep);
         wait; }
else   {
         /*fermeture du tube 1 en écriture */
         close(pip1[1]);
         /* fermeture du tube 2 en lecture */
         close(pip2[0]);
         /* attente d'un message du père */
         read(pip1[0],mesg,5)
         printf("la chaine reçue par le fils est:
       %s\n",mesg);
         /* envoi d'un message au père */
         write(pip2[1],"bonjour",7)
         exit();
       }
}
```



Les tubes : les tubes nommés



LES TUBES Nommés

- **Fichier avec nom, seulement accessible par les deux descripteurs**
- **Tous les processus ayant connaissance des descripteurs peuvent utiliser le tube → pas d'obligation de filiation**

Les tubes : les tubes nommés
PRIMITIVES

Création et ouverture

Un tube nommé est créé par l'intermédiaire de la fonction mkfifo()

int mkfifo (const char *nom, mode_t mode);

Le paramètre nom correspond au chemin d'accès dans l'arborescence de fichiers pour le tube.

Le paramètre mode correspond aux droits d'accès associés au tube

L'ouverture d'un tube nommé s'effectue en utilisant la primitive open()

int open (const char *nom, int mode_ouverture);

La primitive renvoie un seul descripteur correspond au mode d'ouverture spécifié (lecture seule, écriture seule, lecture/écriture).

La primitive open() appliquée au tube nommé est bloquante.

- **demande d'ouverture en lecture est bloquante tant qu'il n'existe pas d'écrivain sur le tube.**
- **demande d'ouverture en écriture est bloquante tant qu'il n'existe pas de lecteur sur le tube.**

Ce mécanisme permet à deux processus de se synchroniser et d'établir un rendez-vous en un point particulier de leur exécution.

Les tubes : les tubes nommés
PRIMITIVES

Fermeture de descripteur

Un processus ferme un descripteur de tube fd en utilisant la primitive close() :

```
int close(int fd);
```

Lecture

La lecture s'effectue par le biais de la primitive read()

```
int read (int desc, char *buf, int nb);
```

Ecriture

L'écriture s'effectue par le biais de la primitive write()

```
int write (int desc, char *buf, int nb);
```

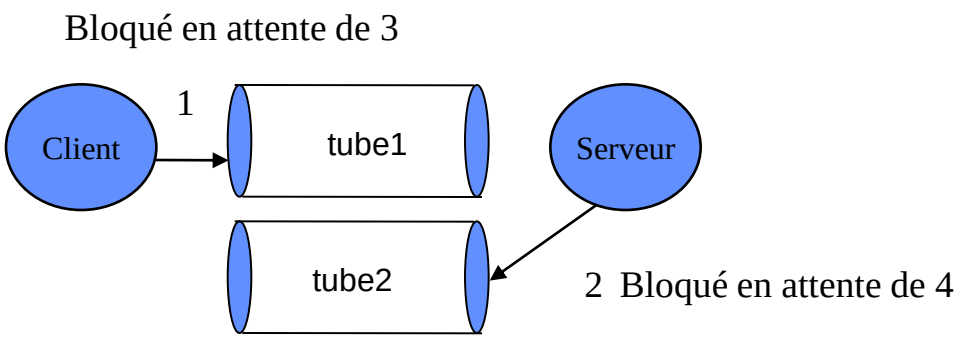
Les tubes : les tubes nommés

```
/************************************************/
/* Processus lecteur sur le tube nommé  */
/************************************************/
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
main ()
{
char zone[11];
int tub;
/* ouverture du tube */
tub = open ("fictub", O_RDONLY);
/* lecture dans le tube */
read (tub, zone, 10);
printf ("processus lecteur du tube fictub: j'ai
      lu %s", zone);
/* fermeture du tube */
close(tub);
}
```

```
/************************************************/
/*   Processus écrivain sur le tube nommé */
/************************************************/
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
main ()
{
mode_t mode;
int tub;
mode = S_IRUSR | S_IWUSR;
/*création du tube nommé */
mkfifo ("fictub", mode);
/* ouverture du tube */
tub = open ("fictub", O_WRONLY);
/* écriture dans le tube */
write (tub, "0123456789", 10);
/* fermeture du tube */
close(tub);
```

Les tubes : les tubes nommés. Exemple d'interblocage

```
   CLIENT
   /* ouverture du tube tube1 en écriture  */
1  tub1 = open ("tube1", O_WRONLY);  -- en attente de l'ouverture en lecture de tube1
   /* ouverture du tube tube2 en lecture */
4  tub2 = open ("tube2", O_RDONLY);

   SERVEUR
2  /*ouverture du tube 2 en écriture */
   tub2 = open ("tube2", O_WRONLY);  -- en attente de l'ouverture en lecture de tube2
3  /* ouverture du tube 1 en lecture */
   tub1 = open ("tube1", O_RDONLY);
```



Notion d'interblocage

On programme comme suit les deux processus

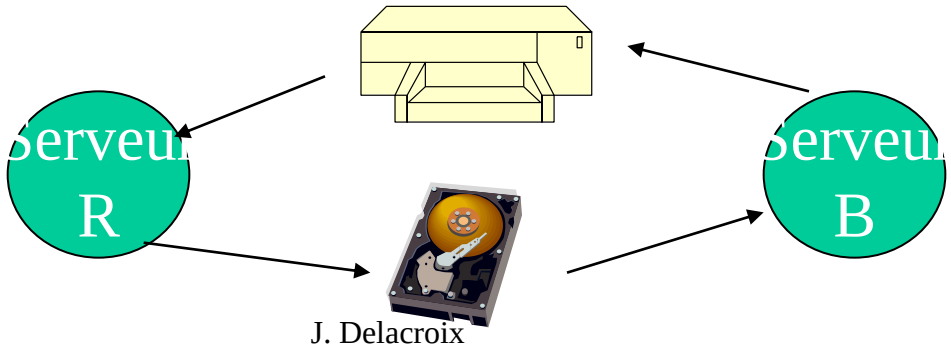
Processus Réservation client	Processus Billet
Effectuer réservation client Verrouiller (Imprimante) Verrouiller (Fichier Commande) Imprimer (facture) Ecrire (commande, Fichier Commande) Deverrouiller (Imprimante) Deverrouiller (Fichier Commande)	Toutes les P unités de temps Faire Verrouiller (Fichier Commande) Verrouiller (Imprimante) Tant que (commandes) Si paiement mettre état payé Imprimer (billet) Fsi Fin tant que Deverrouiller (Fichier Commande) Deverrouiller (Imprimante)

Notion d'interblocage

Processus Réservation client (RC)	Processus Billet (B)
Effectuer réservation client Verrouiller (Imprimante) (imprimante libre allouée à RC) Verrouiller (Fichier Commande) (RC Bloqué)	 REVEIL BILLET Verrouiller (Fichier Commande) (fichier commande libre, alloué à B) Verrouiller (Imprimante) (B bloqué)
Imprimer (facture) Ecrire (commande, Fichier Commande) Deverrouiller (Imprimante) Deverrouiller (Fichier Commande)	Tant que (commandes) Si paiement mettre état payé Imprimer (billet) Fsi Fin tant que Deverrouiller (Fichier Commande) Deverrouiller (Imprimante)

Notion d'interblocage

Processus Réservation client (RC)	Processus Billet (B)
Effectuer réservation client Verrouiller (Imprimante) (imprimante libre allouée à RC) Verrouiller (Fichier Commande) (RC Bloqué)	 REVEIL BILLET Verrouiller (Fichier Commande) (fichier commande libre, alloué à B Verrouiller (Imprimante) (B bloqué)



Notion d'interblocage

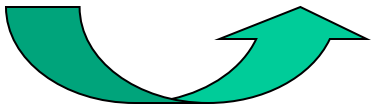
Les deux processus demandent l'accès aux ressources dans un ordre différent.

→ imposer un ordre de demande des ressources

PR
Verrouiller (Imprimante)
Verrouiller (Fichier Commande)
…
Deverrouiller (Imprimante)
Deverrouiller (Fichier Commande)

B
Verrouiller (Fichier Commande)
Verrouiller (Imprimante)
…
Deverrouiller (Imprimante)
Deverrouiller (Fichier Commande)

B
Verrouiller (Imprimante)
Verrouiller (Fichier Commande)
…
Deverrouiller (Imprimante)
Deverrouiller (Fichier Commande)



Notion d'interblocage

Processus Réservation client (RC)	Processus Billet (B)
Effectuer réservation client Verrouiller (Imprimante) (imprimante libre allouée à RC)	REVEIL BILLET Verrouiller (Imprimante) (B bloqué)
Verrouiller (Fichier Commande) (Fichier libre alloué à RC)	
Imprimer (facture) Ecrire (commande, Fichier Commande)	BILLET débloqué, acquiert Fichier Commande et s'exécute
Deverrouiller (Imprimante) Deverrouiller (Fichier Commande)	

LINUX : processus et outils de communication
Partie 2

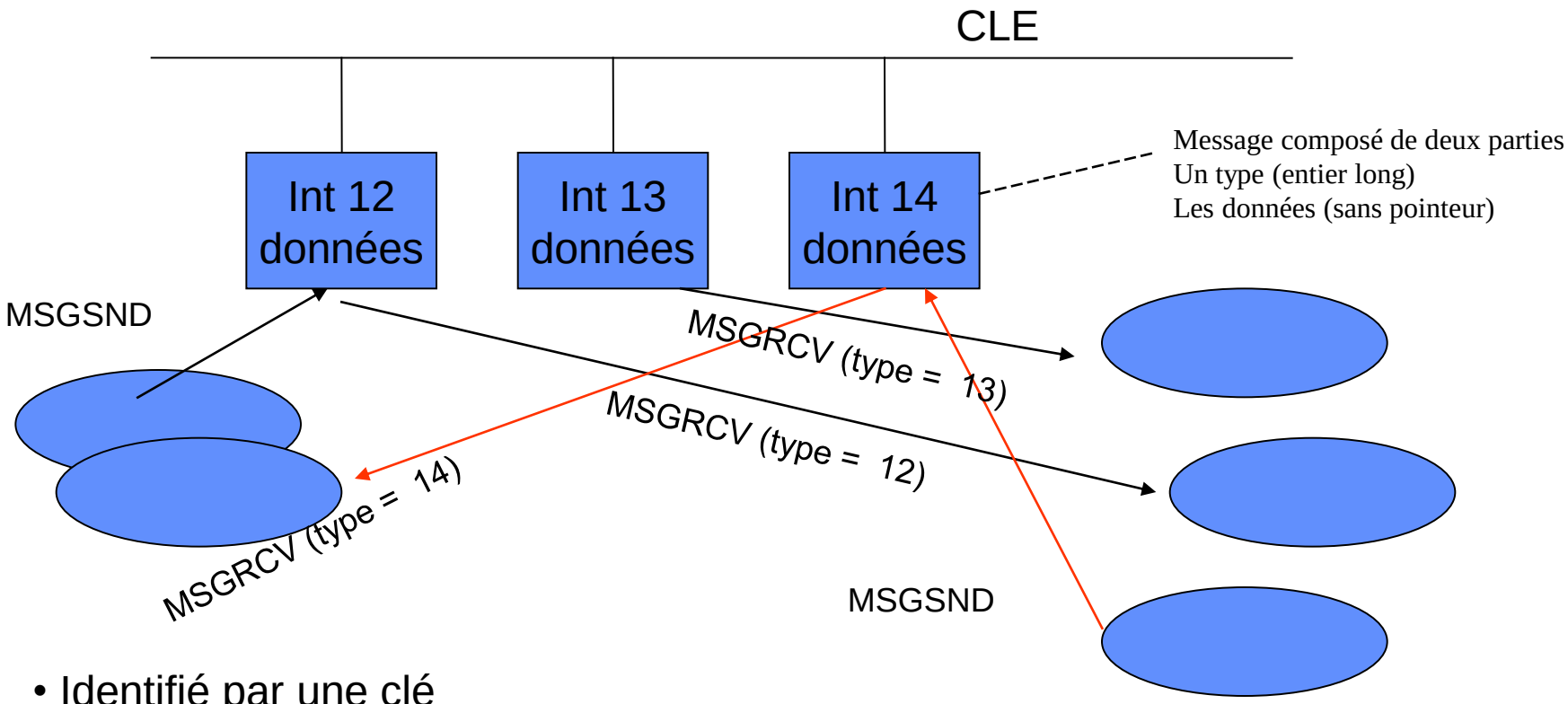
Outils IPC (*Inter Process Communication*)

- **Outils de communication n'appartenant pas au système de gestion de fichiers.**
  - **les files de messages ou MSQ (*Messages Queues*) ;**
  - **les régions de mémoire partagée ;**
  - **les sémaphores.**

Outils IPC
Caractéristiques communes

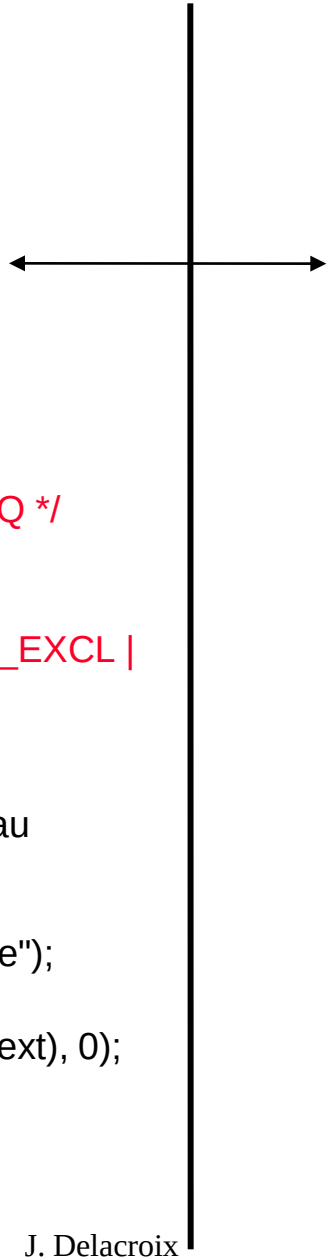
- **Gérés dans des tables du système, une par outils.**
- **identifié de manière unique par un identifiant externe appelé la clé (qui a le même rôle que le chemin d'accès d'un fichier) et par un identifiant interne (qui joue le rôle de descripteur).**
- **Accessible à tout processus connaissant l'identifiant interne de cet outil, soit par héritage ou par une demande explicite au système au cours de laquelle le processus fournit l'identifiant externe de l'outil IPC.**
- **La clé est une valeur numérique de type key_t. Les processus désirant utiliser un même outil IPC pour communiquer doivent se mettre d'accord sur la valeur de la clé référençant l'outil. Ceci peut être fait de deux manières :**
  - **la valeur de la clé est figée dans le code de chacun des processus ;**
  - **la valeur de la clé est calculée par le système à partir d'une référence commune à tous les processus. Cette référence est composée de deux parties, un nom de fichier et un entier. Le calcul de la valeur de la clé à partir cette référence est effectuée par la fonction ftok(), dont le prototype est :**
- **#include <sys/ipc.h>**
- **key_t ftok (const char *ref, int numero);**

Outils IPC
Les files de message ou MSQ



- Identifié par une clé
- Entre processus quelconque connaissant le clé
- Bidirectionnel
- Multiplexage

Exemple Files de messages



PROCESSUS A

```
#include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define cle 17 /* indentifiant externe */
struct msgbuf_exo {long mtype;
                   char mtext[20]; }
struct msgbuf_exo msgp;
main ()  {
int msqid ; /* identifiant interne de la MSQ */

/* allocation de la msq */
msqid = msgget (cle, IPC_CREAT | IPC_EXCL |
      0666);

/* ecriture message dans la msq */
msgp.mtype  =12;   on associe un type au
      message

strcpy(msgp.mtext, "ceci est un message");

msgsnd (msqid, &msgp, strlen(msgp.mtext), 0);
}
```

PROCESSUS B

```
#include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define cle 17/* indentifiant externe */
struct msgbuf_exo {long mtype;
                   char mtext[20]; }
struct msgbuf_exo msgp;
main ()  {
int msqid ; /* identifiant interne de la MSQ */

/* recuperation de la msq */
msqid = msgget (cle, 0);

/* lecture message dans la msq de type 12*/
msgrcv (msqid, &msgp, 19, 12, 0);

/* destruction de la msq */
msgctl (msqid, IPC_RMID, NULL);
}
```

Outils IPC
Les files de message ou MSQ

- **L'accès à une file de message s'effectue par l'intermédiaire de la primitive msgget(). Cette primitive permet :**
  - **la création d'une nouvelle file de messages ;**
  - **l'accès à une file de messages déjà existante.**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t cle, int option);
```

combinaison des constantes IPC_CREAT, IPC_EXCL et de droits d'accès

Clenv = ftok ("/home/delacroix/essai.c", 24)

msqid = msgget (cleenv, IPC_CREAT | IPC_EXCL | 0666); : création d'un file nouvelle de cle « clenv » avec des accès en lecture/écriture pour tous

msqid = msgget (cleenv, IPC_CREAT | 0660); : création ou accès à une file existante de cle « clenv » avec des accès en lecture/écriture pour le propriétaire et le groupe du propriétaire

msqid = msgget (cleenv, 0); : accès à une file existante de cle « clenv »

Exemple Files de messages

PROCESSUS A

```
#include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define cle 17 /* indentifiant externe */
struct msgbuf_exo {long mtype;
                   char mtext[20]; }
struct msgbuf_exo msgp;
main ()  {
int msqid ; /* identifiant interne de la MSQ */

/* allocation de la msq */
msqid = msgget (cle, IPC_CREAT | IPC_EXCL |
      0666);

/* ecriture message dans la msq */
msgp.mtype  =12;   on associe un type au
      message

strcpy(msgp.mtext, "ceci est un message");

msgsnd (msqid, &msgp, strlen(msgp.mtext), 0);
}
```

PROCESSUS B

```
#include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define cle 17/* indentifiant externe */
struct msgbuf_exo {long mtype;
                   char mtext[20]; }
struct msgbuf_exo msgp;
main ()  {
int msqid ; /* identifiant interne de la MSQ */

/* recuperation de la msq */
msqid = msgget (cle, 0);

/* lecture message dans la msq de type 12*/
msgrcv (msqid, &msgp, 19, 12, 0);

/* destruction de la msq */
msgctl (msqid, IPC_RMID, NULL);
}
```

Outils IPC
Les files de message ou MSQ

- **La communication au travers d'une file de messages peut être bidirectionnelle.**

- **Chaque message comporte les données en elles-mêmes ainsi qu'un type qui permet de faire du multiplexage dans la file de messages et de désigner le destinataire d'un message.**

Format des messages
Un message est toujours composé de deux parties :

```
struct message {
        long mtype;
        int n1;
        char[4];
        float fl1; };
```

la première partie constitue le type du message. C'est un entier long positif ;

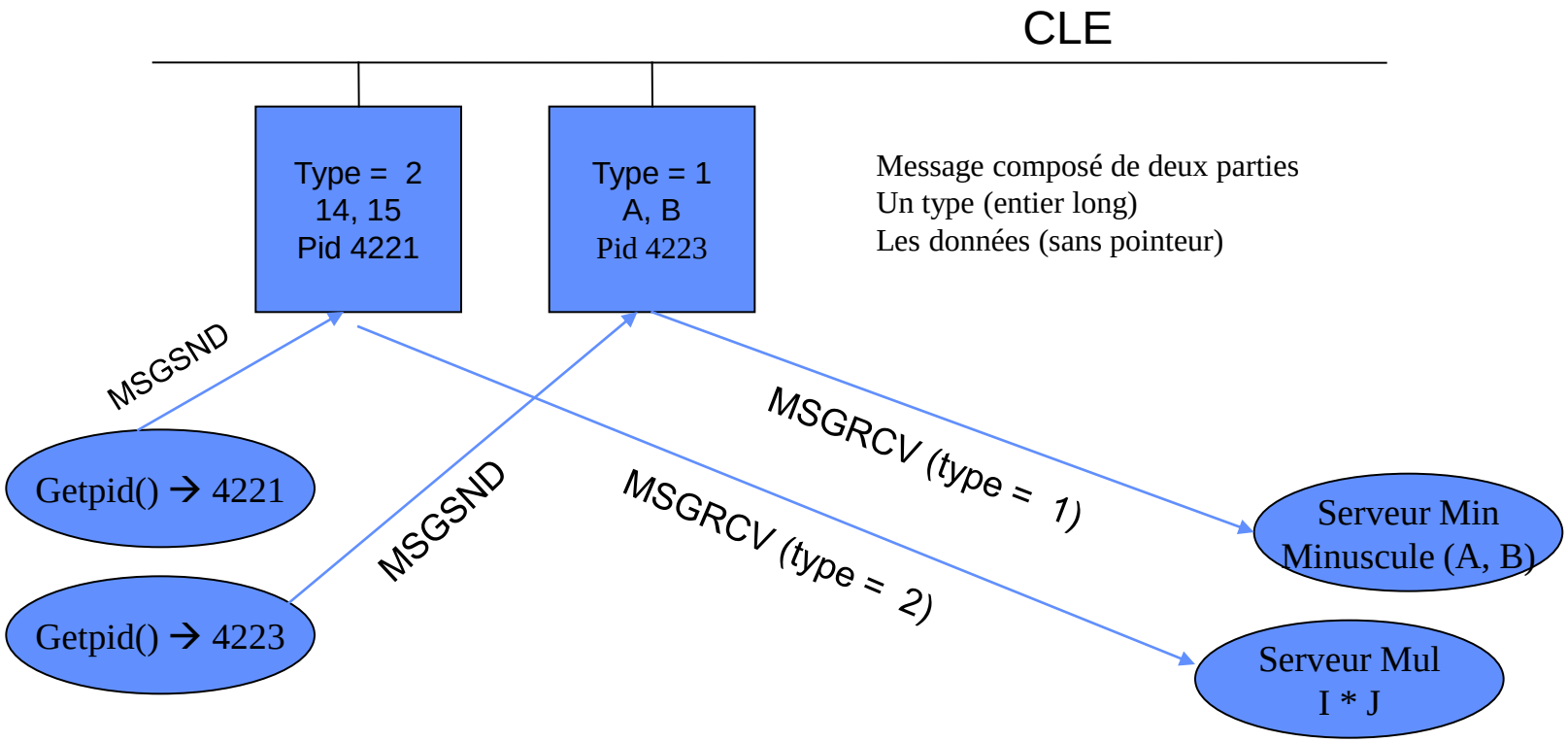
la seconde partie est composée des données proprement dites.

- **L'envoi et la réception d'un message s'effectue via les primitives :**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int idint, const void *msg, int longueur, int option);
int msgrcv (int idint, const void *msg, int longueur, long letype, int option);
```

Identifiant clé — message — taille du message en données — type à prélever

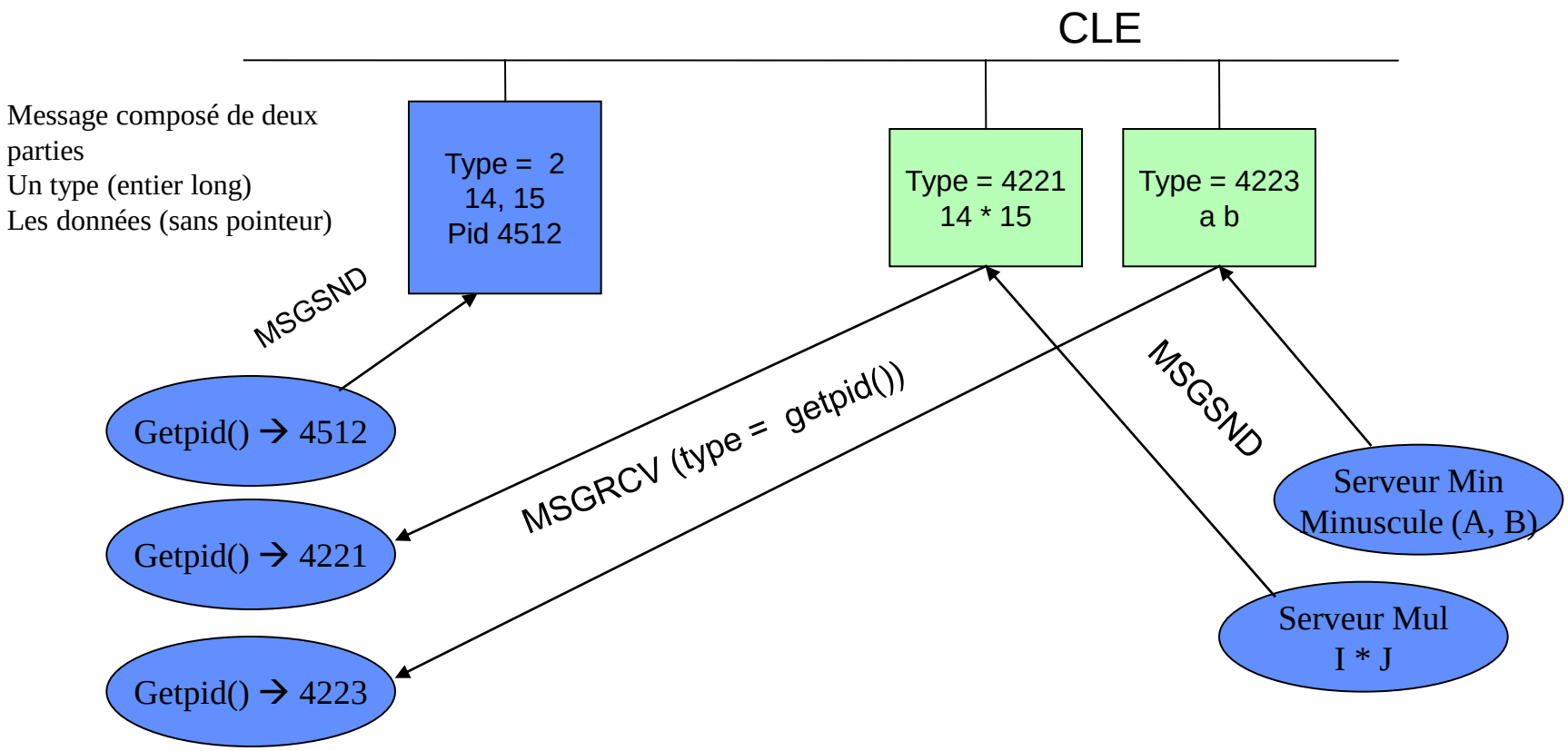
Outils IPC
Les files de message ou MSQ



Le type dans le message permet de désigner le destinataire. La primitive MSGRCV spécifie quel type de message prélever.

Côté client : type à utiliser pour l'identifier de façon unique : son pid
Côté serveur : de petits numéros qui ne peuvent pas correspondre à des pid de clients.

Outils IPC
Les files de message ou MSQ



Le type dans le message permet de désigner le destinataire. La primitive MSGRCV spécifie quel type de message prélever.

Côté client : type à utiliser pour l'identifier de façon unique : son pid
Côté serveur : de petits numéros qui ne peuvent pas correspondre à des pid de clients.

Outils IPC
Les Sémaphores

- **Famille des IPCs, un ensemble de sémaphores est identifié par une clef.**

- **Les opérations P, V et ATT (attente qu'une valeur de sémaphore soit nulle) s'effectuent sur un tableau de sémaphores, atomiquement**

  ↳ **L'ensemble des opérations est réalisée avant que le processus puisse poursuivre son exécution..**

Outils IPC
Les Sémaphores

- **Famille des IPCs, un ensemble de sémaphores est identifié par une clef.**

- **Les opérations P, V et ATT (attente qu'une valeur de sémaphore soit nulle) s'effectuent sur un tableau de sémaphores, atomiquement**

  ↳ **L'ensemble des opérations est réalisée avant que le processus puisse poursuivre son exécution..**

Outils IPC
Les Sémaphores

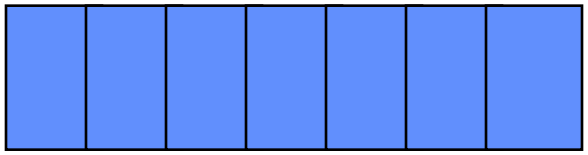
- **Création et recherche d'un ensemble de sémaphore**

int semget (key_t cle, int nsems, int semflg);

Création ou recherche d'un ensemble de n sémaphores identifiés par clé.

Retourne un identifiant interne de type entier

Semflg = constantes IPC_EXCL, IPC_CREAT, 0



nsems

```
Struct semaphore {
atomic_t count ; /* compteur */
int sleepers; /* nombre de processus endormis */
Wait_queue_head_t wait; /* file d'attente */ }
```

Outils IPC
Les Sémaphores

- **Opérations sur les sémaphores**

int semop (int semid, struct sembuf *ops, unsigned nsops);

Réalisation d'un ensemble d'opérations (nsops) décrite chacune dans une structure sembuf sur l'ensemble de sémaphore semid.

```
struct sembuf {
   unsigned short sem_num;/* numéro du sémaphore dans le tableau
    */
   short sem_op; /* opération à réaliser sur le sémaphore */
   short sem_flg; /* options */
    };
```

– si sem_op est négatif, l'opération à réaliser est une opération P;
– si sem_op est positif, l'opération à réaliser est une opération V;
– si sem_op est nul, l'opération à réaliser est une opération ATT.

Outils IPC
Les Sémaphores

- **Initialiser un sémaphore**

int semctl (int semid, int semnum, int cmd, union semun arg);

semctl (semid, 0, SETVAL, 3) initialisation à la valeur 3 du sémaphore 0 dans l'ensemble désigné par l'identifiant semid.

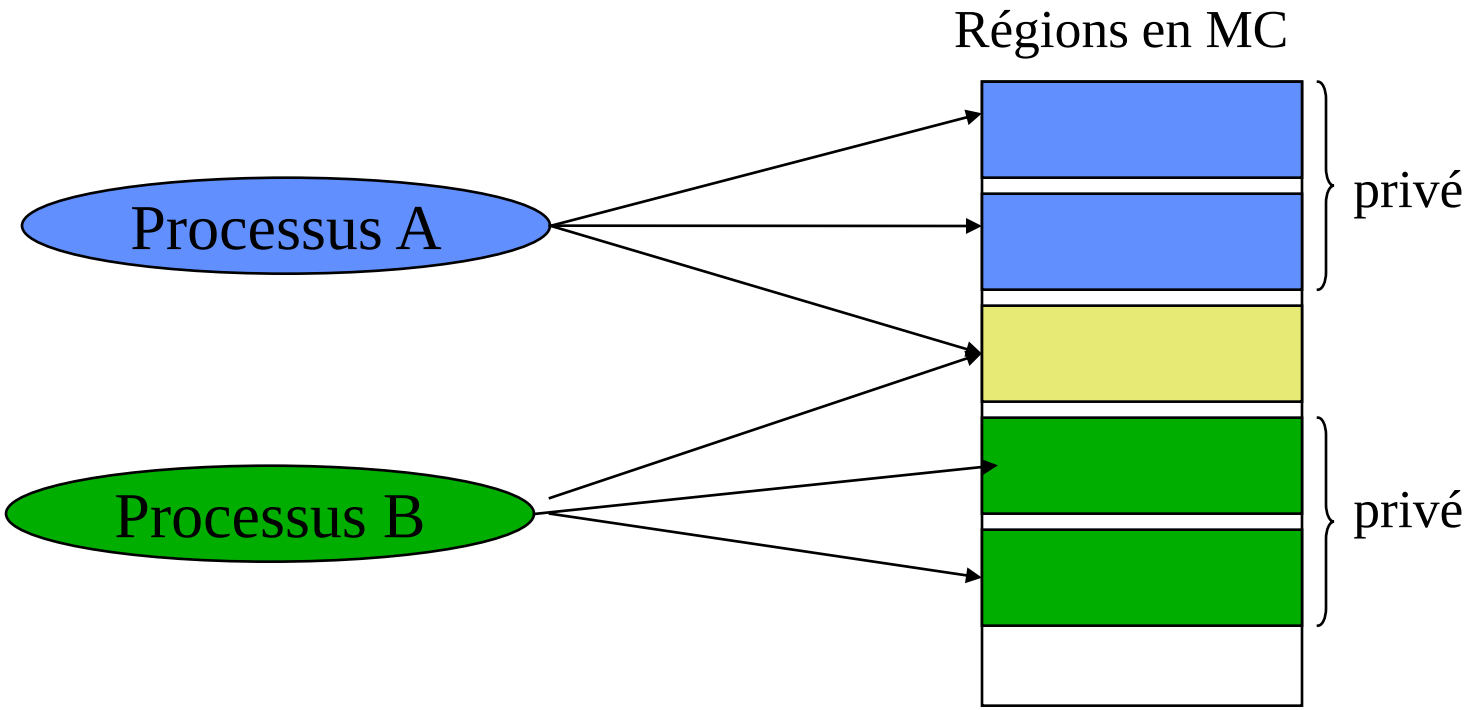
- **Détruire un ensemble de sémaphores**

int semctl (int semid, 0, IPC_RMID, 0);

```
#include <stdio.h>
#include <pthread.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int i, nb_place;
int semid;
struct sembuf operation;

void reservation()
{
/* opération P */
operation.sem_num = 0;
operation.sem_op = -1;
operation.sem_flg = 0;
semop (semid, &operation, 1);
nb_place = nb_place - 1;
/* opération V */
operation.sem_num = 0;
operation.sem_op =  1;
operation.sem_flg = 0;
semop (semid, &operation, 1);
}
```

```
main()
{ pthread_t num_thread[3];


/* création d'un sémaphore initialisé à
la valeur 1 */
semid = semget (12, 1,
                IPC_CREAT|IPC_EXCL|0600);
semctl (semid, 0, SETVAL, 1);


for(i=0; i<3; i++) {
pthread_create(&num_thread[i], NULL,
(void *(*)())reservation, NULL);
pthread_join(num_thread, NULL);
semctl (semid, 0, IPC_RMID, 0)
}
```

Outils IPC
les régions partagées

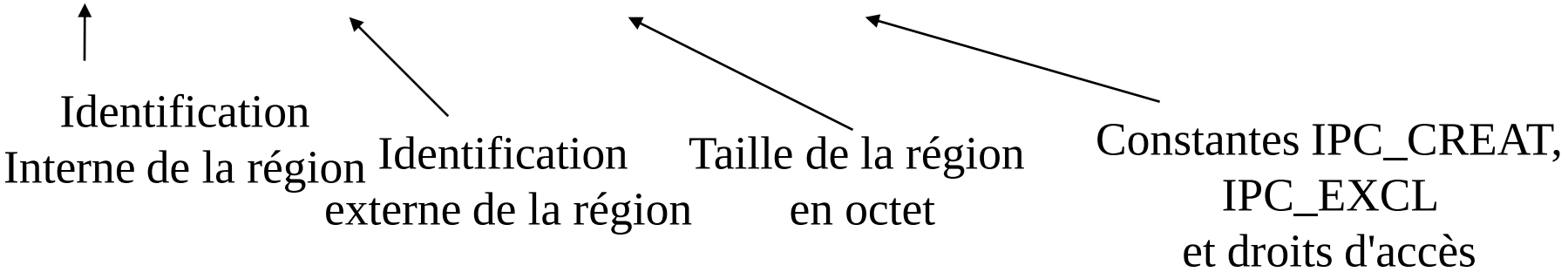


- **Région de mémoire pouvant être partagée entre plusieurs processus**
- **Un processus doit attacher le région à son espace d'adressage avant de pouvoir l'utiliser**
- **Outil IPC repéré par une clé unique**
- **L'accès aux données présentes dans la région peut requérir une synchronisation (outil sémaphore)**

Outils IPC
les régions partagées

- **Création ou accès à une région de mémoire partagée**

int shmget (key_t cle, int taille, int option)



- **Attachement d'une région de mémoire partagée**

void *shmat (int shmid, const void *shmadd, int option)

- **Détachement d'une région de mémoire partagée**

void *shmdt (void *shmadd)

```
/*****************************************************************/
/*              processus créateur du segment et écrivain        */
/*****************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE         256

main()
{
int shmid;
char *mem ;

/* création du segment de mémoire partagée avec la clé CLE */
shmid=shmget((key_t)CLE,1000,0750 |IPC_CREAT | IPC_EXCL);

/* attachement */
if((mem=shmat(shmid,NULL,0))==(char *)-1)
{
   perror("shmat");
   exit(2);
}

/* écriture dans le segment */
strcpy(mem,"voici une écriture dans le segment");
exit(0);
}
```

```
/*************************************************************/
/*              processus destructeur du segment et lecteur          */
/*************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE         256

main()
{
/* récupération du segment de mémoire */
shmid=shmget((key_t)CLE,0,0);

/* attachement */
mem=shmat(shmid,NULL,0);

/* lecture dans le segment */
printf("lu: %s\n",mem);

/* détachement du processus */
shmdt(mem);

/* destruction du segment */
shmctl(shmid,IPC_RMID,NULL);

exit(0)
}
```