UNIVERSITÉ PARIS SUD

IUT D'ORSAY

M3101 · Principes des systèmes d'exploitation

Les processus

IUT d'Orsay · DUT Informatique · 2015 / 2016

Les processus

Processus est un programme en cours d'exécution.

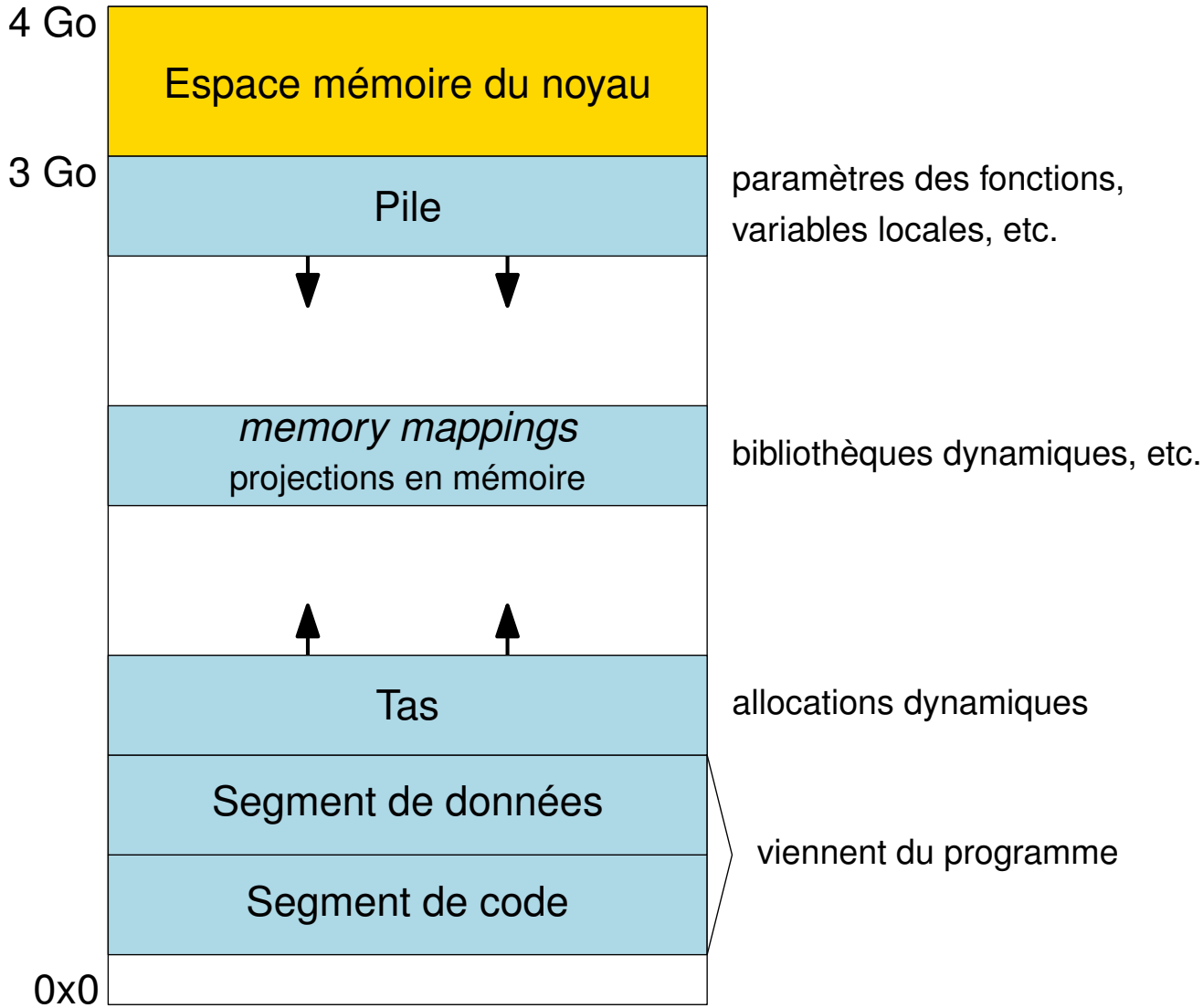
Le programme fournit

- code segment — les instructions à exécuter
- data segment — les données définies dans les sources

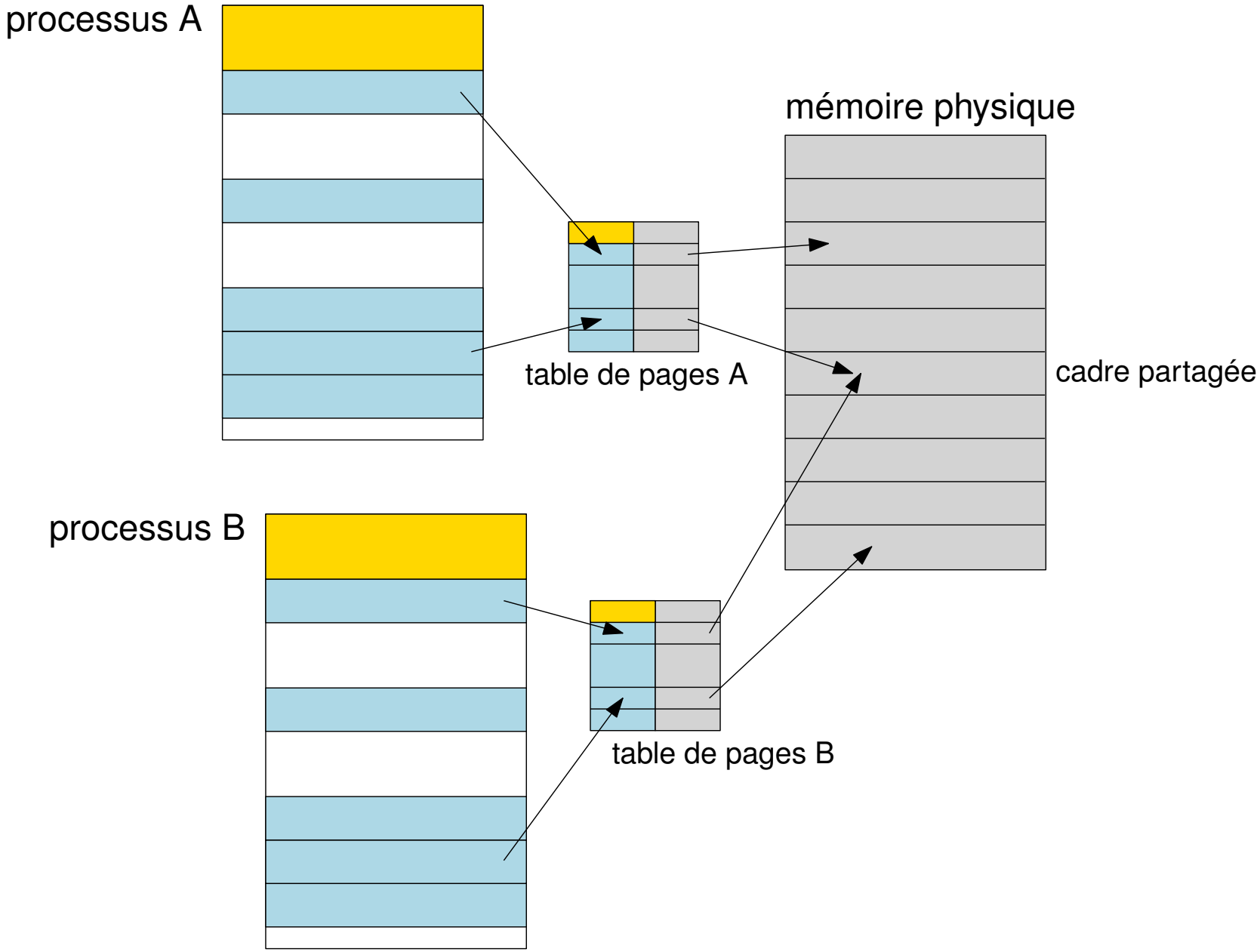
L'état d'un processus inclut

- la mémoire — *code*, *data*, *heap* (tas), *stack* (pile)
- le contexte d'exécution — les registres, le compteur ordinal, le pointeur sur le sommet de pile, etc.
- *Process Control Bloc* (PCB) — l'état vis-à-vis du SE

L'espace d'adressage d'un processus



Mémoire virtuelle et physique



Process Control Block (PCB)

Une entrée dans la table de processus maintenue par le noyau :

- Identification : *process ID* (PID), *parent PID* (PPID), *user ID* (UID), *effective UID* (EUID), etc.
- État : actif, prêt, en attente, arrêté, terminé
- Contexte d'exécution (quand inactif)
- Mémoire : les adresses des segments, la table de pages
- Ressources : les fichiers ouverts, le terminal de rattachement
- Ordonnancement : les priorités, les horloges et les alarmes
- Relations : les processus fils, le groupe, la session
- Signaux : bloqués, suspendus, les gestionnaires, etc.
- Comptabilité : temps, mémoire, entrées-sorties, etc.
- ...

Dans les sources du noyau Linux : `struct task_struct`.

Les primitives système

L'interface entre les programmes utilisateur et le noyau.

Donnent l'accès à toutes les fonctionnalités du système :

fichiers, réseau, matériel, IPC, gestion de processus...

Sont exécutées en mode noyau (ring 0 = « omnipotent »)

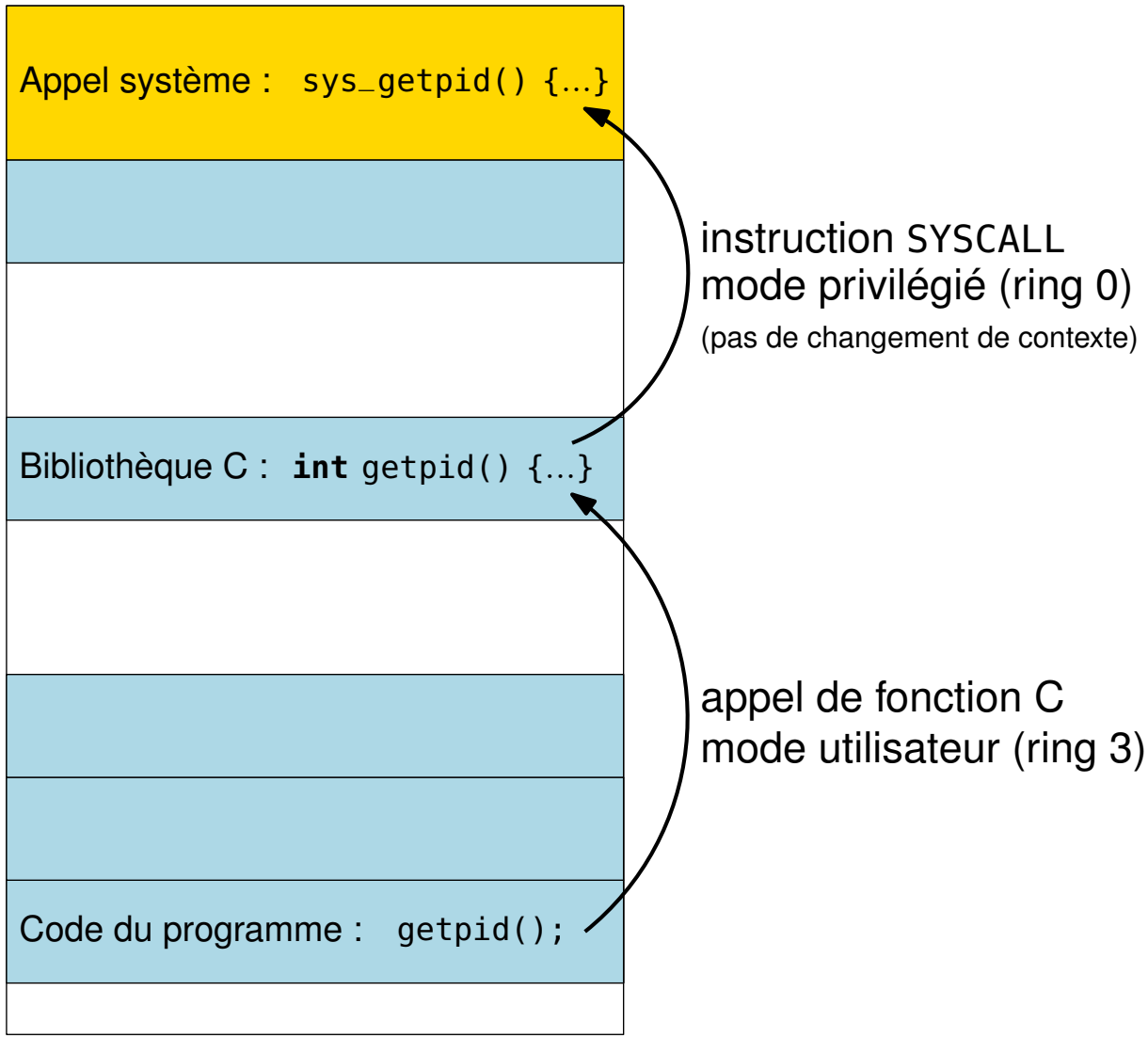
$\Rightarrow$ contrôle de privilèges est nécessaire

Sont exécutées dans le contexte du processus appelant

$\Rightarrow$ pas de sauvegarde de registres, pas de *TLB flush*

La liste des primitives est figée à l'intérieur du noyau.

Appel d'une primitive



Primitives sur les processus

```c
#include <unistd.h>
#include <stdio.h>
int main() {
  printf("Mon PID est %d\n", getpid());
  printf("Le PID de mon père est %d\n", getppid());
  sleep(15); // pour garder le processus vivant
  return 0; }
```

Dans deux terminaux :

```
> ./example_getpid
Le PID est 3013
Le PID du père est 2918
>
```

```
> ps -u bob -o pid,ppid,state,cmd
  PID  PPID S CMD
 2918  1114 S bash
 3013  2918 S ./example_getpid
>
```

Le processus forment un arbre, avec `init` (PID 1) dans la racine.

Création de processus

```
pid_t fork(void);
```

Un nouveau processus est né :

- un nouveau PID
- une nouvelle entrée dans la table des processus

Hérite les attributs du processus appelant :

- le contenu de mémoire (COW = *copy-on-write*)
  $\Rightarrow$ le même code et les mêmes données
- le propriétaire (UID, EUID, SUID, GID, EGID, SGID)
- les variables d'environnement
- les fichiers ouverts
- les gestionnaires de signaux
- etc.

La primitive `fork()`

```c
#include <unistd.h>
#include <stdio.h>
int main() {
  printf("Start: %d\n", getpid());
  fork(); // un processus entre, deux processus sortent
  printf("End: %d\n", getpid());
  return 0; }
```

Dans deux terminaux :

```
> ./example_fork
Start: 5088
End: 5088
End: 5090
>
```

```
> ./example_fork
Start: 5094
End: 5096
End: 5094
>
```

L'ordre d'exécution entre père et fils n'est pas prévisible.

La primitive `fork()`

Si le code est le même, comment différencier les deux processus ?

La valeur de retour de `fork()` fait la différence :

```
pid_t id = fork();
```

- `id` > 0 $\Rightarrow$ on est dans le père, `id` vaut le PID du fils
- `id` $= 0$ $\Rightarrow$ on est dans le fils
- `id` $= -1$ $\Rightarrow$ `fork()` a échoué, le fils n'a pas été créé

La primitive `fork()`

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main() {
  pid_t id;
  printf("Start: %d\n", getpid());
  id = fork(); // un processus entre, deux processus sortent
  if (id == -1) { perror("fork() failure"); exit(1); }
  if (id == 0) {
    printf("Child: PID %d, parent %d\n", getpid(), getppid());
  } else {
    printf("Parent: PID %d, child %d\n", getpid(), id); }
  return 0; }
```

```
> ./example_fork_2
Start: 6621
Parent: PID 6621, child 6623
Child: PID 6623, parent 6621
```

La primitive `fork()`

Si le contenu de mémoire est le même, peut-on le partager ?

```
1  int main() {
2    int compteur = 0;
3    pid_t id = fork();
4    if (id == 0) {
5      printf("Child: PID %d, parent %d\n", getpid(), getppid());
6      compteur += 1;
7    } else {
8      printf("Parent: PID %d, child %d\n", getpid(), id);
9      compteur += 2;
10   }
11   printf("End: PID %d, compteur = %d\n", getpid(), compteur);
12   return 0; }
```

exécuté par le père : 1 2 3↑ 3↓ 4 8 9 11 12

exécuté par le fils : 3↓ 4 5 6 11 12

La primitive `fork()`

Si le contenu de mémoire est le même, peut-on le partager ?

```
1  int main() {
2    int compteur = 0;
3    pid_t id = fork();
4    if (id == 0) {
5      printf("Child: PID %d, parent %d\n", getpid(), getppid());
6      compteur += 1;
7    } else {
8      printf("Parent: PID %d, child %d\n", getpid(), id);
9      compteur += 2;
10   }
11   printf("End: PID %d, compteur = %d\n", getpid(), compteur);
12   return 0; }
```

```
> ./example_fork_3
Parent: PID 7012, child 7014
End: PID 7012, compteur = 2
Child: PID 7014, parent 7012
End: PID 7014, compteur = 1
```

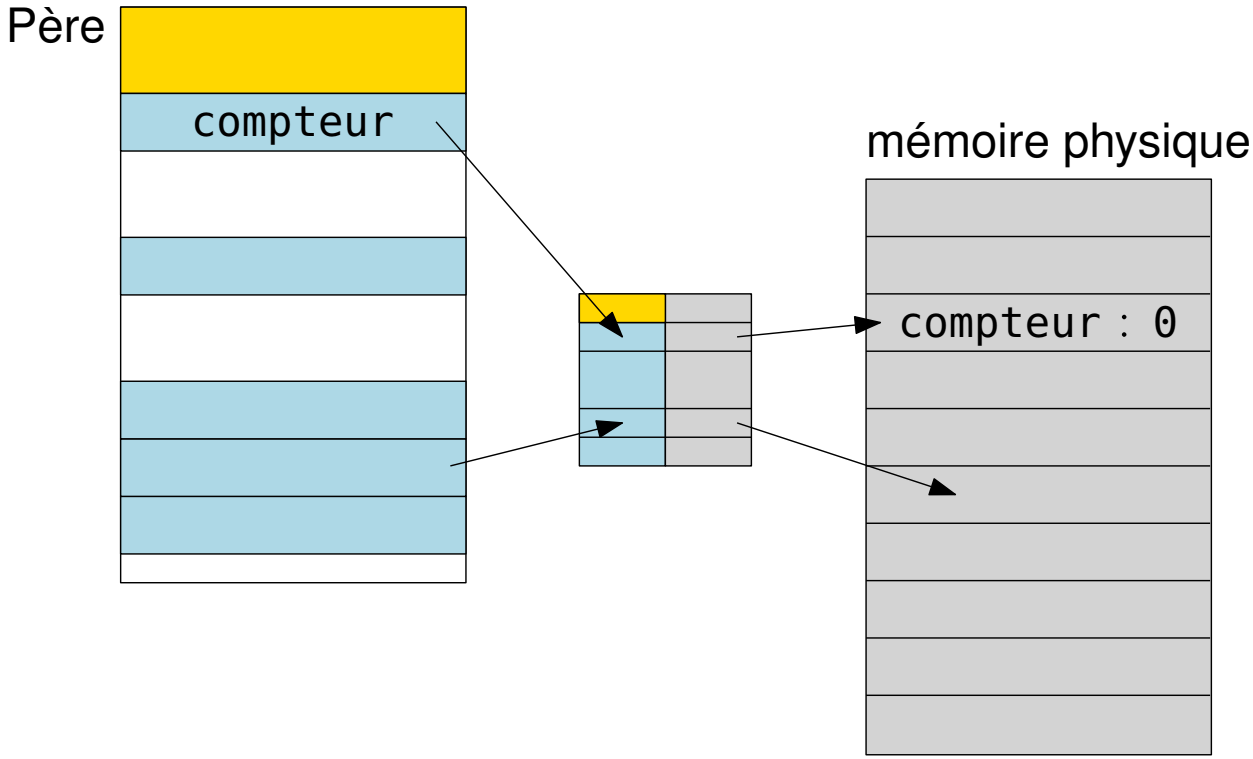
La primitive `fork()`

Si le contenu de mémoire est le même, peut-on le partager ?

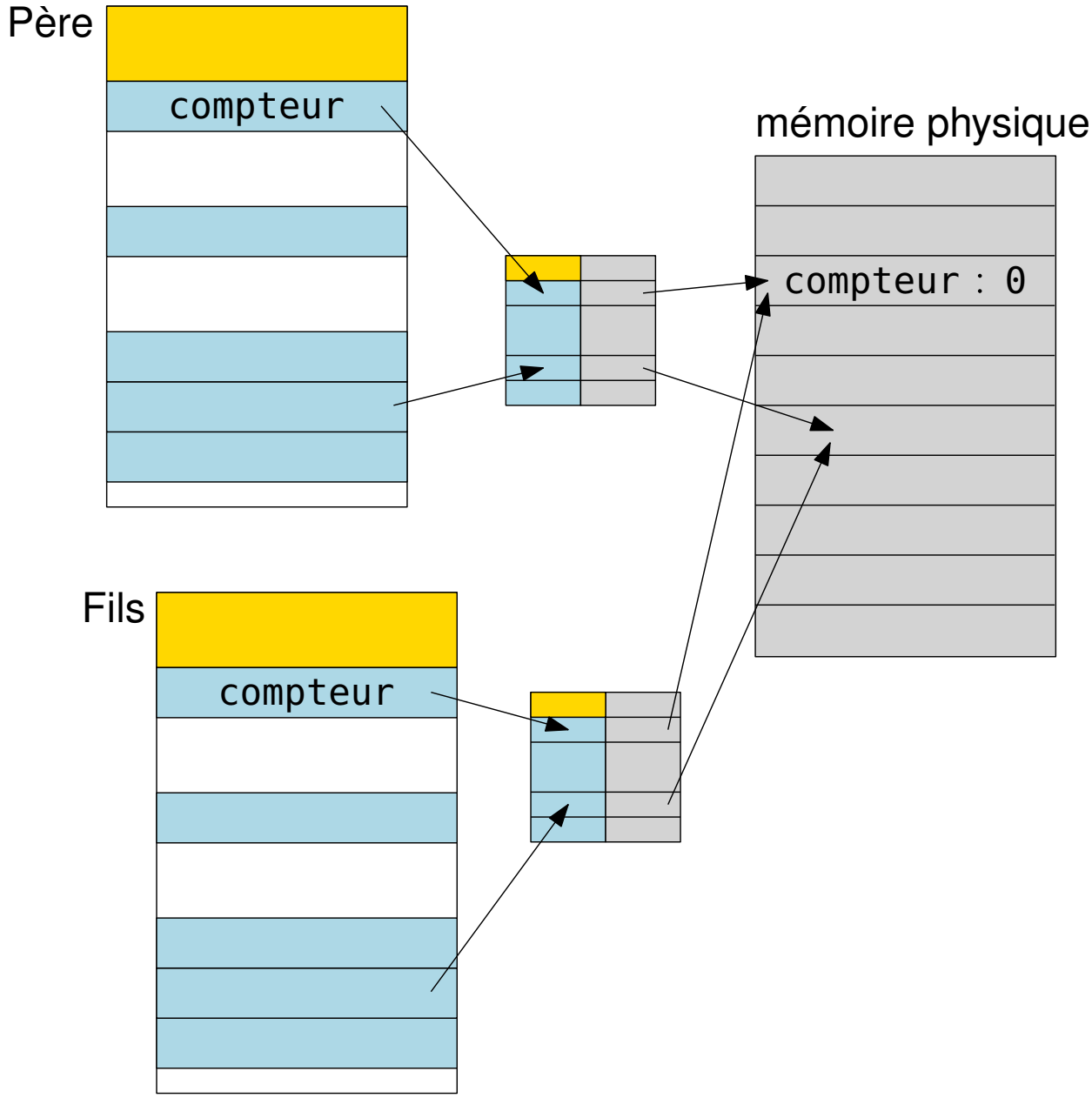
```
1  int main() {
2    int compteur = 0;
3    pid_t id = fork();
4    if (id == 0) {
5      printf("Child: PID %d, parent %d\n", getpid(), getppid());
6      compteur += 1;
7    } else {
8      printf("Parent: PID %d, child %d\n", getpid(), id);
9      compteur += 2;
10   }
11   printf("End: PID %d, compteur = %d\n", getpid(), compteur);
12   return 0; }
```

Les processus ne partagent pas la mémoire (cf. *threads*).

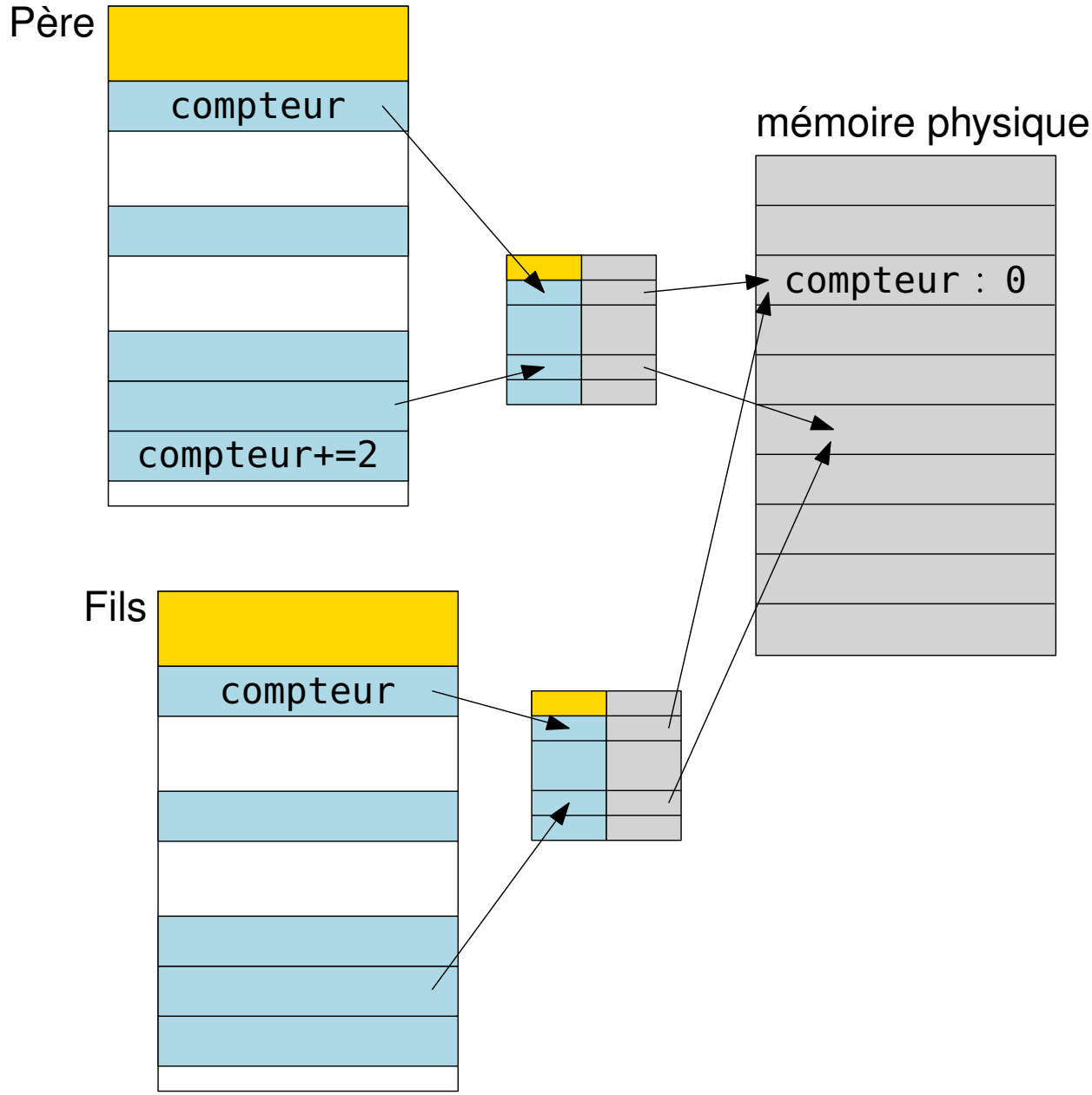
Copy-On-Write



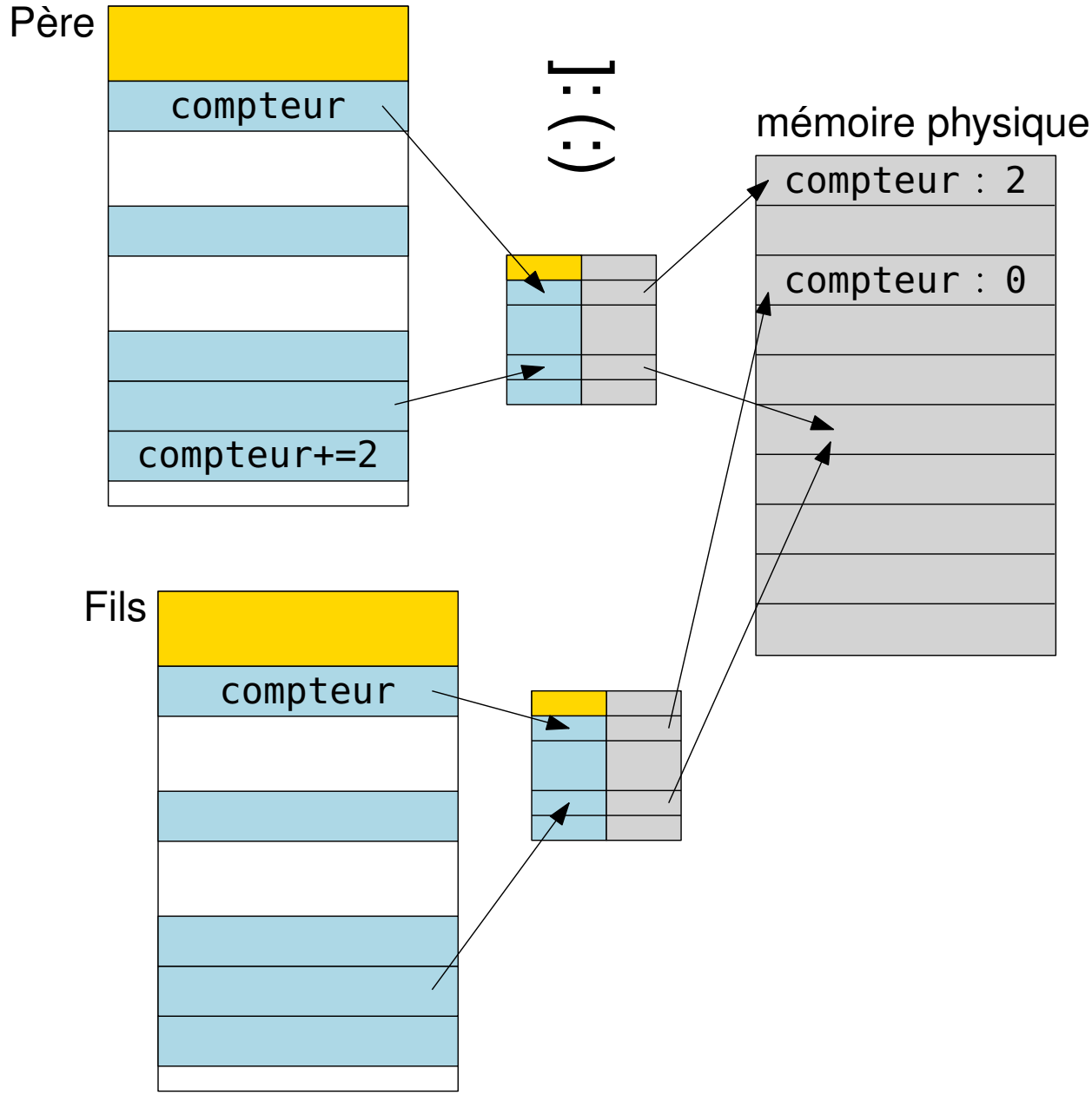
Copy-On-Write



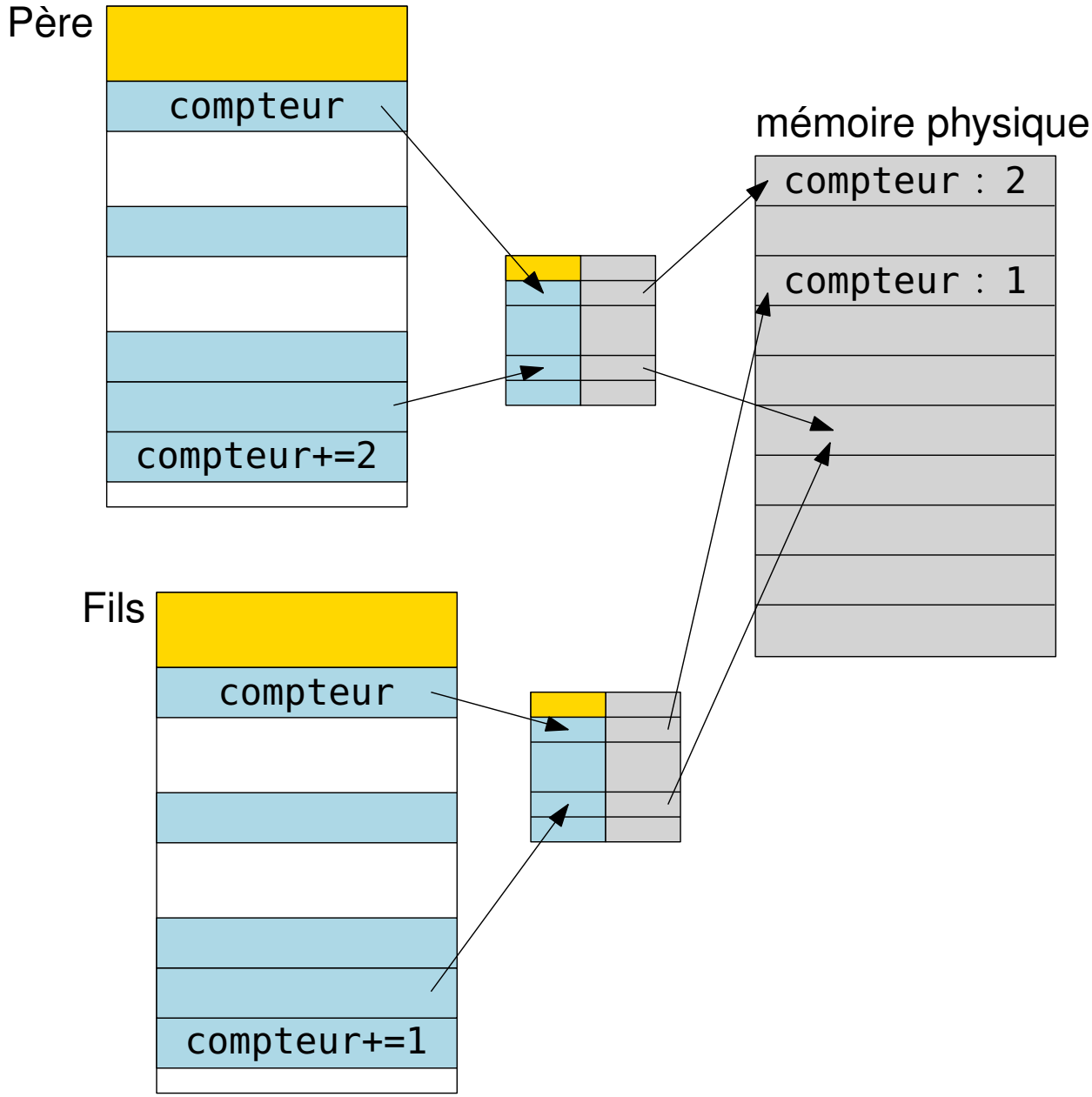
Copy-On-Write



Copy-On-Write



Copy-On-Write



Terminaison de processus

Terminaison normale (avec un code de retour) :

- fin de la fonction `main()` : `return 0;`
- primitive système `exit()` : `exit(1);`

Terminaison anormale (par un signal) :

- envoyé par l'utilisateur : `<Ctrl-C>`, commande `kill`
- envoyé par un autre processus : primitive `kill()`
- envoyé par le noyau : en cas d'erreur d'exécution

Chaque type de signal a un numéro associé.

Terminaison de processus

À la terminaison d'un processus :

- ses ressources sont libérés
- ses fichiers ouverts sont fermés
- ses enfants sont adoptés par `init`
- son père reçoit un signal `SIGCHLD`
- son état d'exécution devient « terminé » (= zombie)
- son entrée dans la table de processus n'est pas libérée

Le PID et le PCB d'un processus terminé ne sont pas libérés

avant que son père *ne ramasse les cendres*

La primitive `wait()`

```
pid_t wait(int * etat);
```

Attend la fin d'un processus fils (on ne choisit pas lequel).
Retourne immédiatement si un fils est mort avant l'appel.

Renvoie le PID du fils terminé dans la valeur de retour.

Met dans l'entier `*etat` le compte-rendu sur la terminaison :

- le type de la terminaison : normale ou anormale
- le code de retour si normale
- le numéro de signal si anormale

On peut appeler `wait(NULL)` si on n'en a pas besoin.

Libère l'entrée du défunt dans la table de processus.

Renvoie -1 s'il n'y a pas ou plus de processus fils.

La primitive `wait()`

```
1  int main() {
2    pid_t id = 0;
3    if (fork() == 0) {
4      printf("Child %d\n", getpid());
5      exit(0);
6    }
7    id = wait(NULL);
8    printf("Child %d is gone\n", id);
9    return 0; }
```

```
> ./example_wait
Child 7014
Child 7014 is gone
```

Ne dépend pas de l'ordre d'exécution entre père et fils.

Le compte-rendu de `wait()`

```
int status = 0;
…
wait(&status);
```

Les macros définies dans `sys/wait.h` :

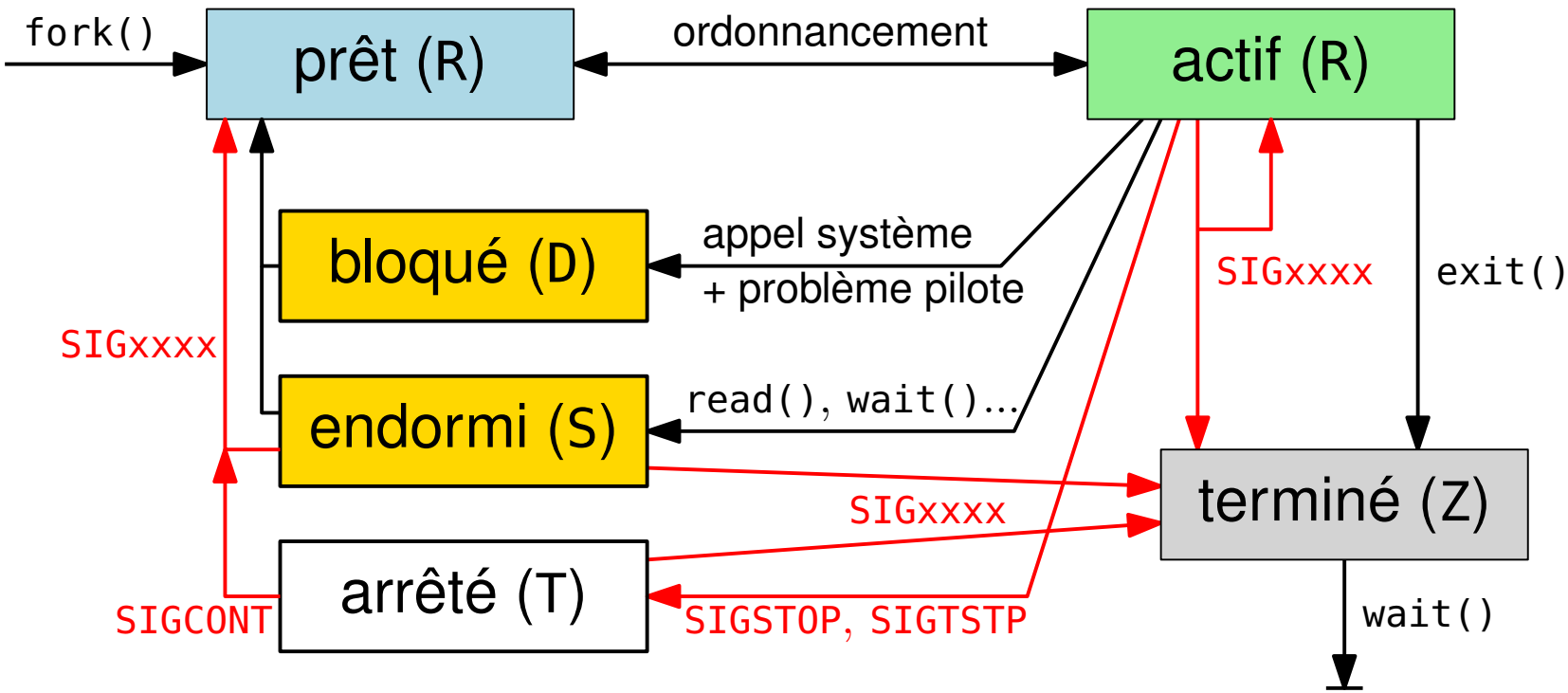
`WIFEXITED(status)`	1 si la terminaison est normale, sinon 0
`WIFSIGNALED(status)`	1 si la terminaison est anormale, sinon 0
`WEXITSTATUS(status)`	le code de retour du processus terminé si la terminaison est normale
`WTERMSIG(status)`	le numéro du signal qui a causé la fin si la terminaison est anormale

Le compte-rendu de `wait()`

```c
int main() {
  pid_t id = 0;
  int status = 0;
  int coderet = 0;
  if (fork() == 0) {
    coderet = random() % 256; // entre 0 et 255
    printf("Child %d, coderet %d\n", getpid(), coderet);
    exit(coderet);
  }
  id = wait(&status);
  printf("Child %d is gone\n", id);
  if (WIFEXITED(status)) // terminaison normale
    printf("Exit status %d\n", WEXITSTATUS(status));
  else if (WIFSIGNALED(status)) // terminaison anormale
    printf("Killed by signal %d\n", WTERMSIG(status));
  return 0; }
```

```
> ./example_wait_2
Child 17863, coderet 103
Child 17863 is gone
Exit status 103
```

La cycle de vie d'un processus



La lecture

1. Brian W. Kernighan and Rob Pike,
   The Practice of Programming, 1999,
   `http://www.cs.princeton.edu/~bwk/tpop.webpage/`
2. W. Richard Stevens, Stephen A. Rago,
   Advanced Programming in the UNIX Environment, 3e éd., 2013,
   `http://www.apuebook.com/index.html`
3. W. Richard Stevens et al.,
   UNIX Network Programming, vol. 1 et 2, 1998, 2003,
   `http://www.unixnetworkprogramming.com/`
4. Michael Kerrisk,
   The Linux Programming Interface, 2010,
   `http://man7.org/tlpi/`
5. LWN, `https://lwn.net/`
6. `man`