

Systemes d'exploitations II

Présentation #3 : Processus

30/10/2021

Ahmed Benmoussa



Centre universitaire d'Aflou

Objectif de cette presentation

- **Rappel sur le cours precedent.**
- **Etats d'un processus.**
- **Opérations sur un processus.**

Rappel: Concepts fondamentaux de SE

1. Thread

- Décrit un état du programme.
- Pointeur d'instruction (*Program Counter*), Registres, Pile, ...

2. Espace d'adresse (*address space*)

- Une plage d'adresse accessible pour le programme (lecture/écriture).

3. Processus: une instance d'un programme en execution

- Espace memoire protégé.
- Un ou plusieurs Threads.

4. Dual mode operation / Protection

- Seul le SE a accès a certaines ressources.
- Isoler les programmes les uns des autres, et proteger le SE des programmes.

Rappel: Processus

- **Processus: un environnement d'exécution avec des droits réduits**
 - Un espace d'adresse avec un ou plusieurs Threads
 - Possède un espace memoire
 - Possède descripteurs de fichiers
 - Encapsule un ou plusieurs Thread.
- Une application consiste d'un ou plusieurs processus.
- **Pourquoi processus?**
 - Se proteger les uns des autres
 - Et proteger le SE
 - Apporter une protection pour la memoire.

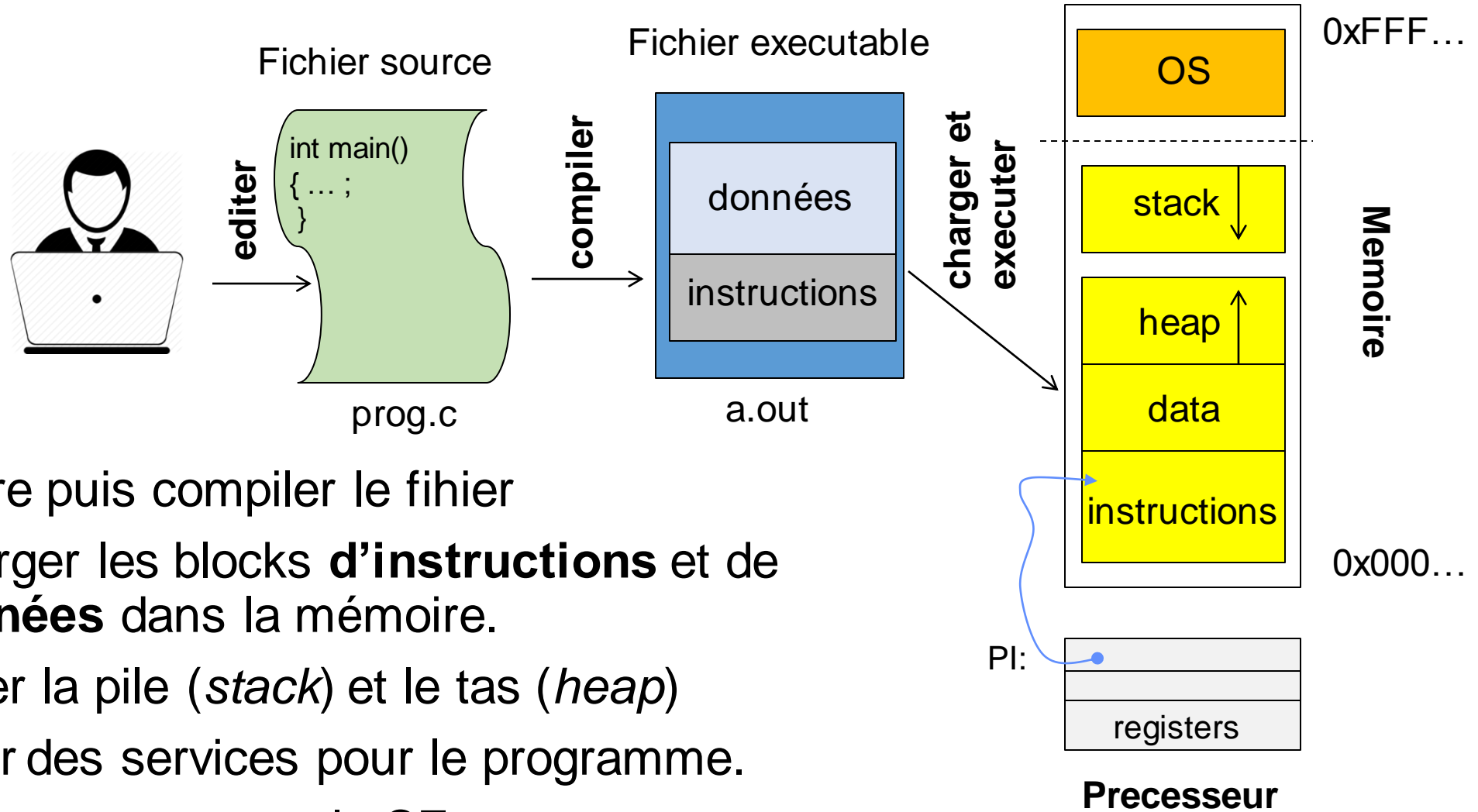
Programme vs Processus

- Un programme se constitue de **code** et **données**.
- Spécifié dans un langage de programmation.
- Sauvegarder sur fichier qui lui est stocké sur le disque.
- “**Executer un programme**” = **creation d’un processus**.

Process \neq Program

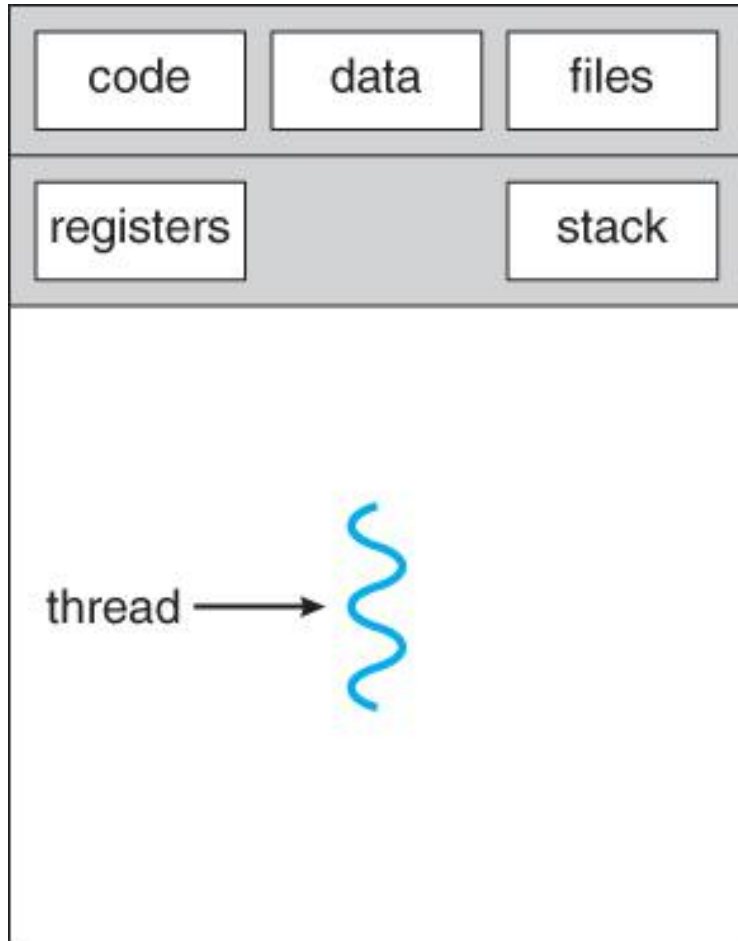
- A process is **alive**
- Un programme peut être executer plusieurs fois simultanement (1 programme, 2 processus).
 - > ./program &
 - > ./program &

Execution d'un programme

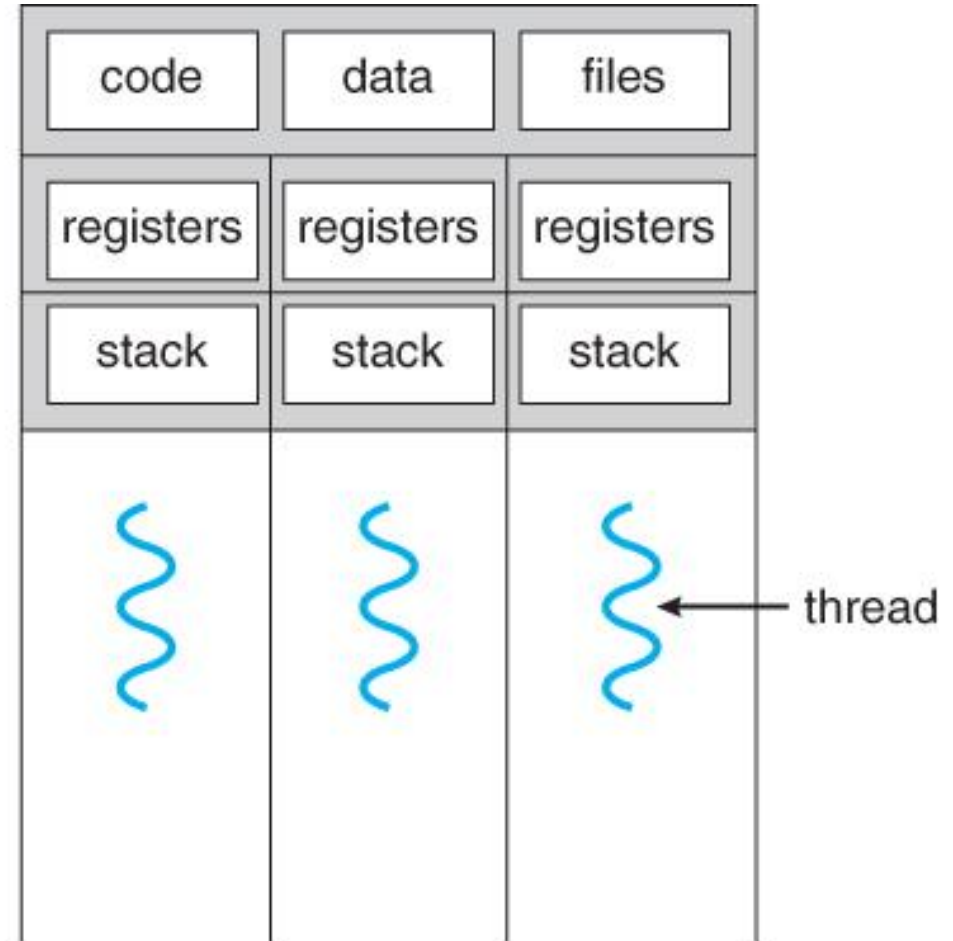


- Ecrire puis compiler le fichier
- Charger les blocks **d'instructions** et de **données** dans la mémoire.
- Créer la pile (*stack*) et le tas (*heap*)
- Offrir des services pour le programme.
- Tout en protégeant le SE.

Processus mono-thread vs processus multi-thread



single-threaded process



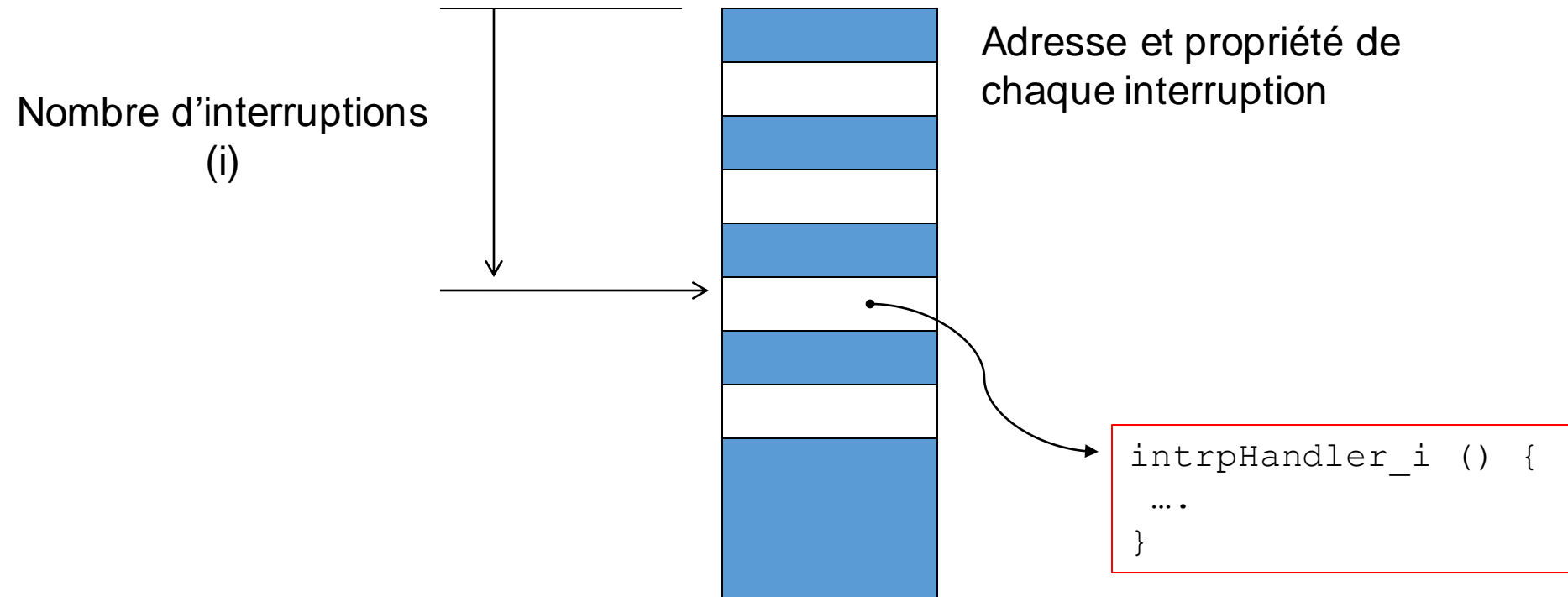
multithreaded process

3 Types de changement « User-->Kernel »

- Appels systèmes (*syscall*)
 - Processus demande un service système, comme “exit”
- Interruptions (*Interrupt*)
 - Timer, I/O
 - Indépendant du processus utilisateur.
- Exception
 - événement interne au niveau du processus
 - Violation de protection (*segmentation fault*), division par zéro, ...

Où sont sauvegardés ces interruptions?

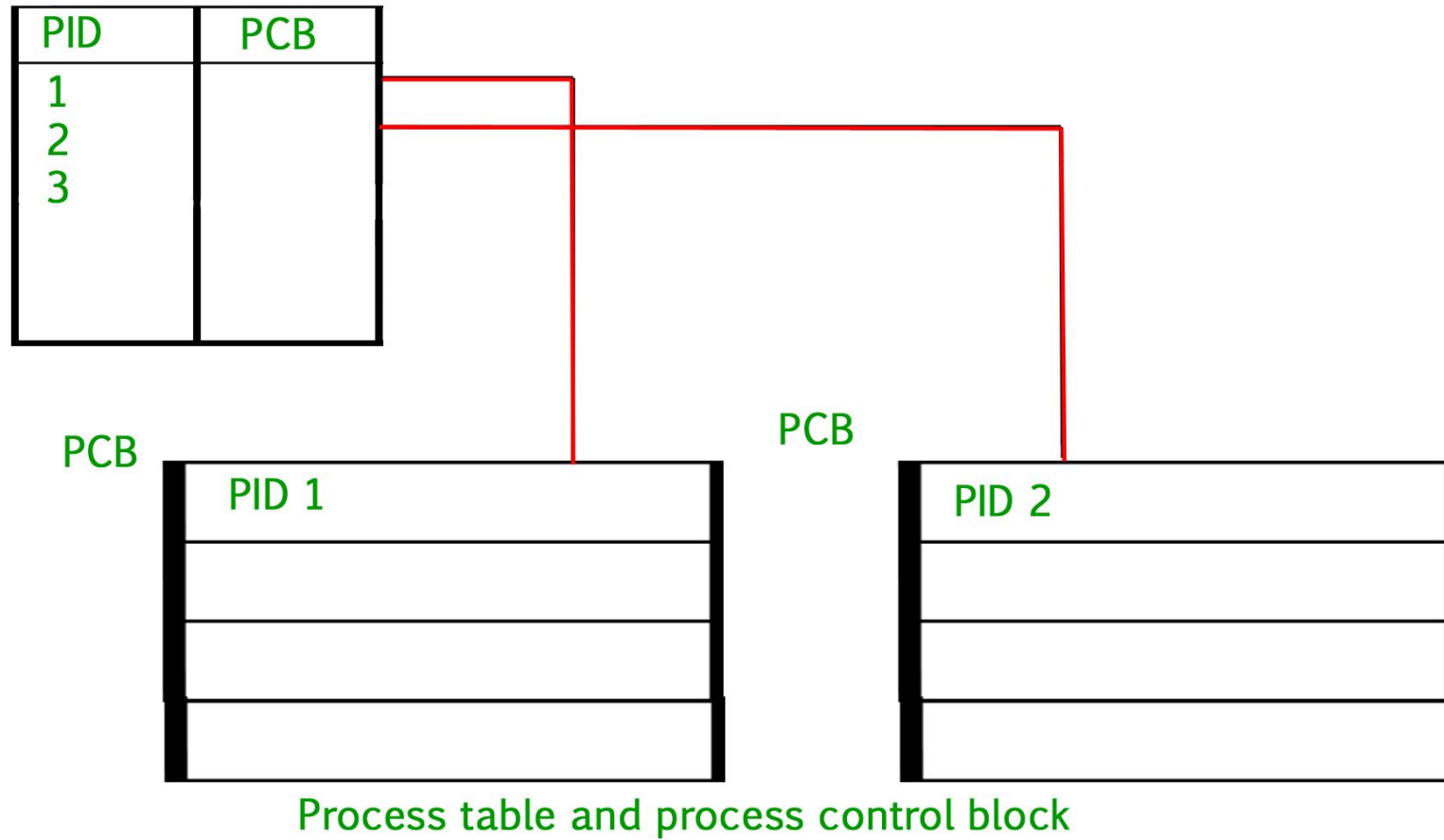
Vecteur d'interruptions (*Interrupt Vector*)



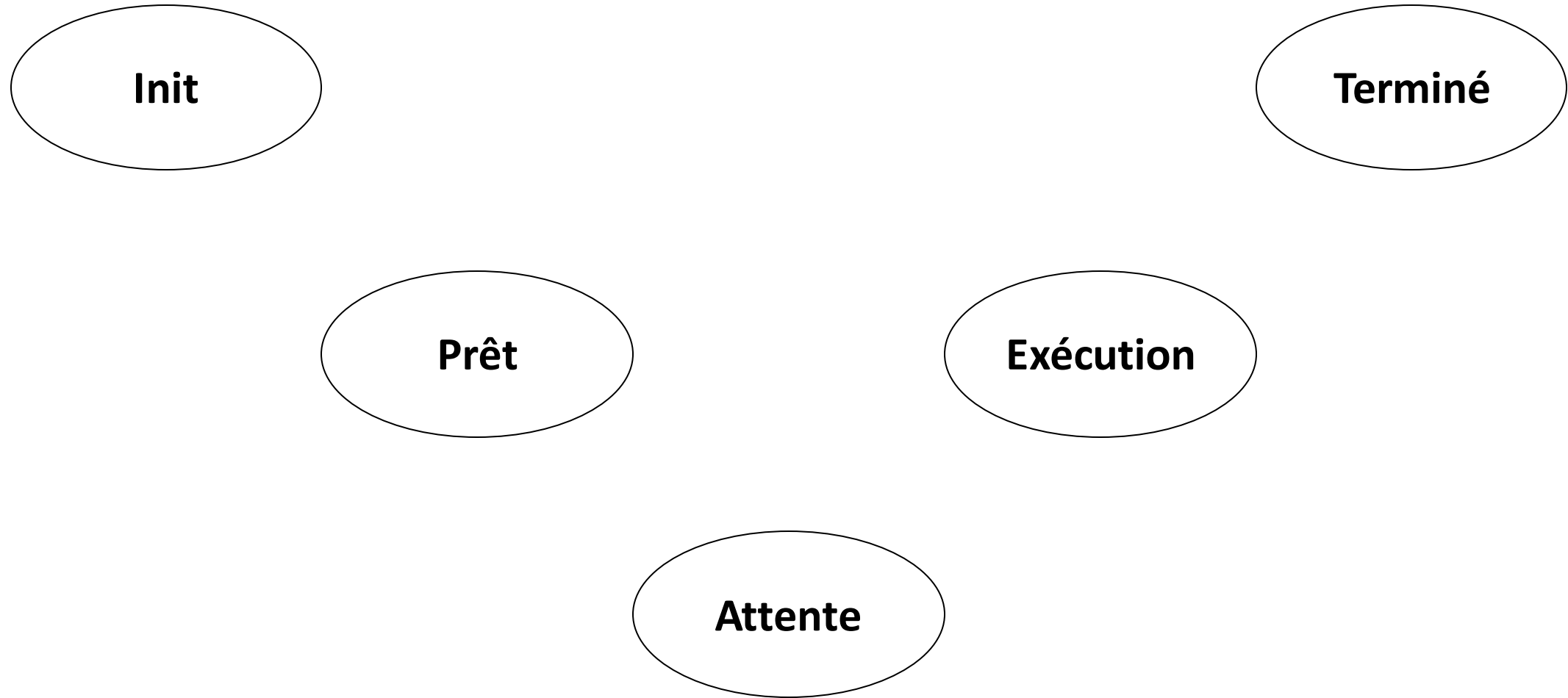
Process Control Block (PCB)

- Le noyau represente chaque processus comme un “block de control” (*Process Control Block*)
 - Emplacement dans la memoire (page table).
 - L’etat des registres.
 - Emplacement de l’executable sur le disque.
 - Utilisateur executant (uid).
 - Identificateur du Processus (pid).
 - Etat (execution, prêt, bloquer, ...).
 - Temps d’execution, ...
- Le planificateur du noyau (*Kernel Scheduler*) maintient une structure de données contenant les PCBs.

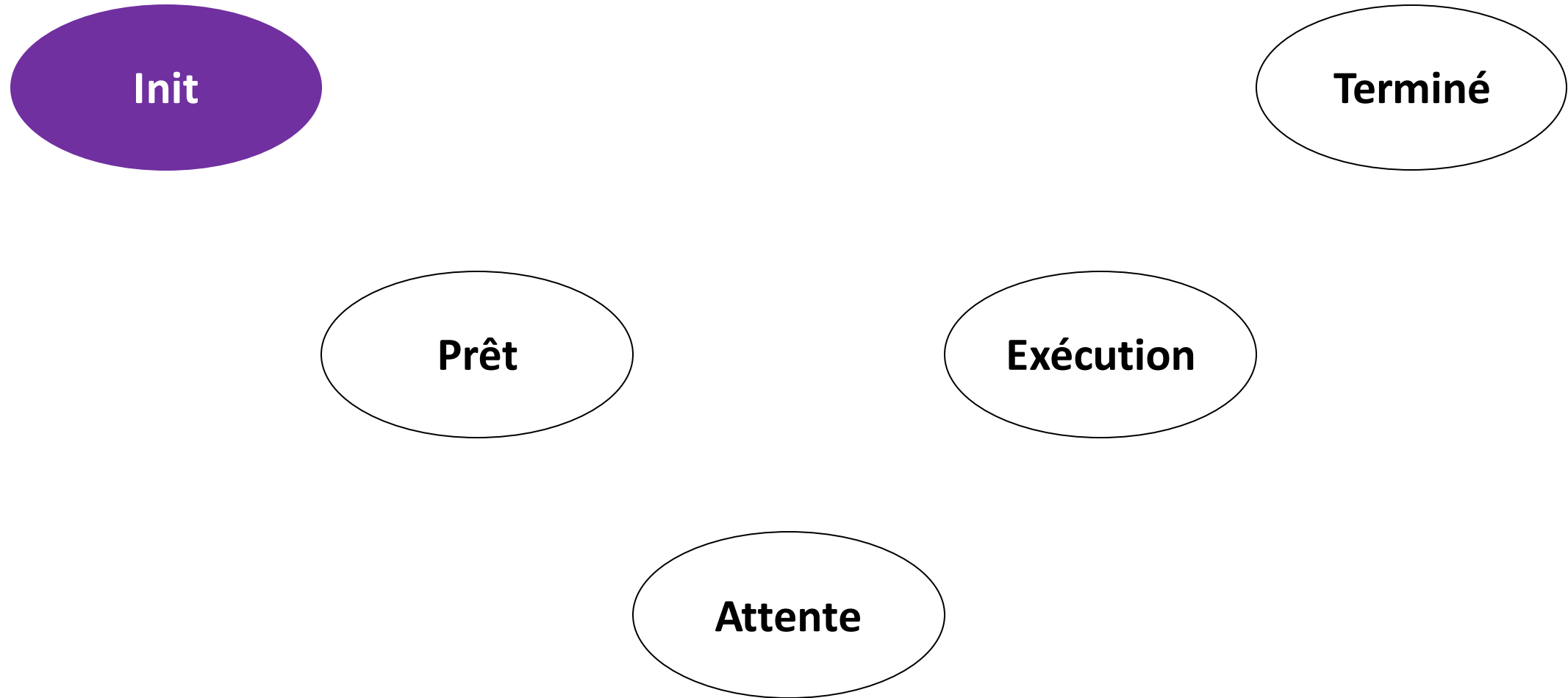
Table des processus



Cycle de vie d'un processus



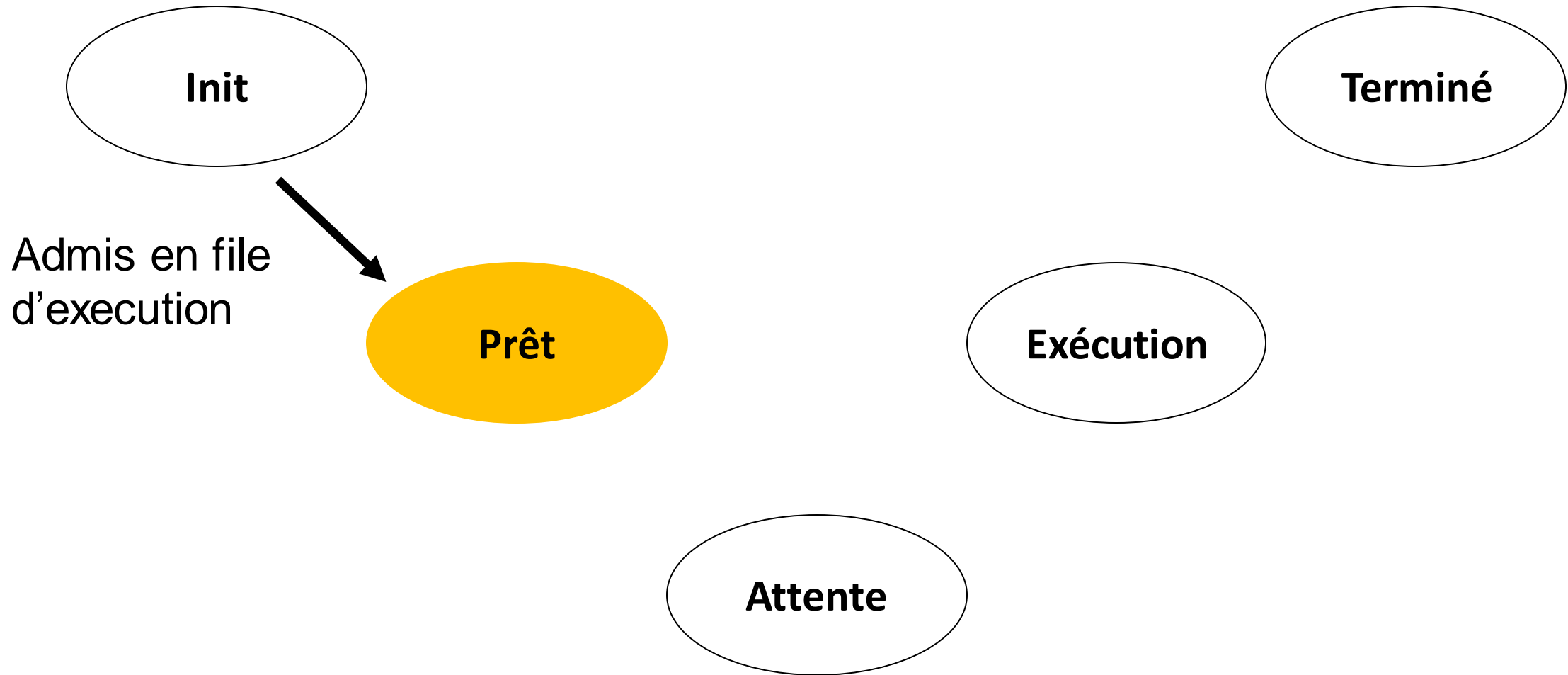
Création du processus



Etat PCB: Création

Registres: Non initialisés

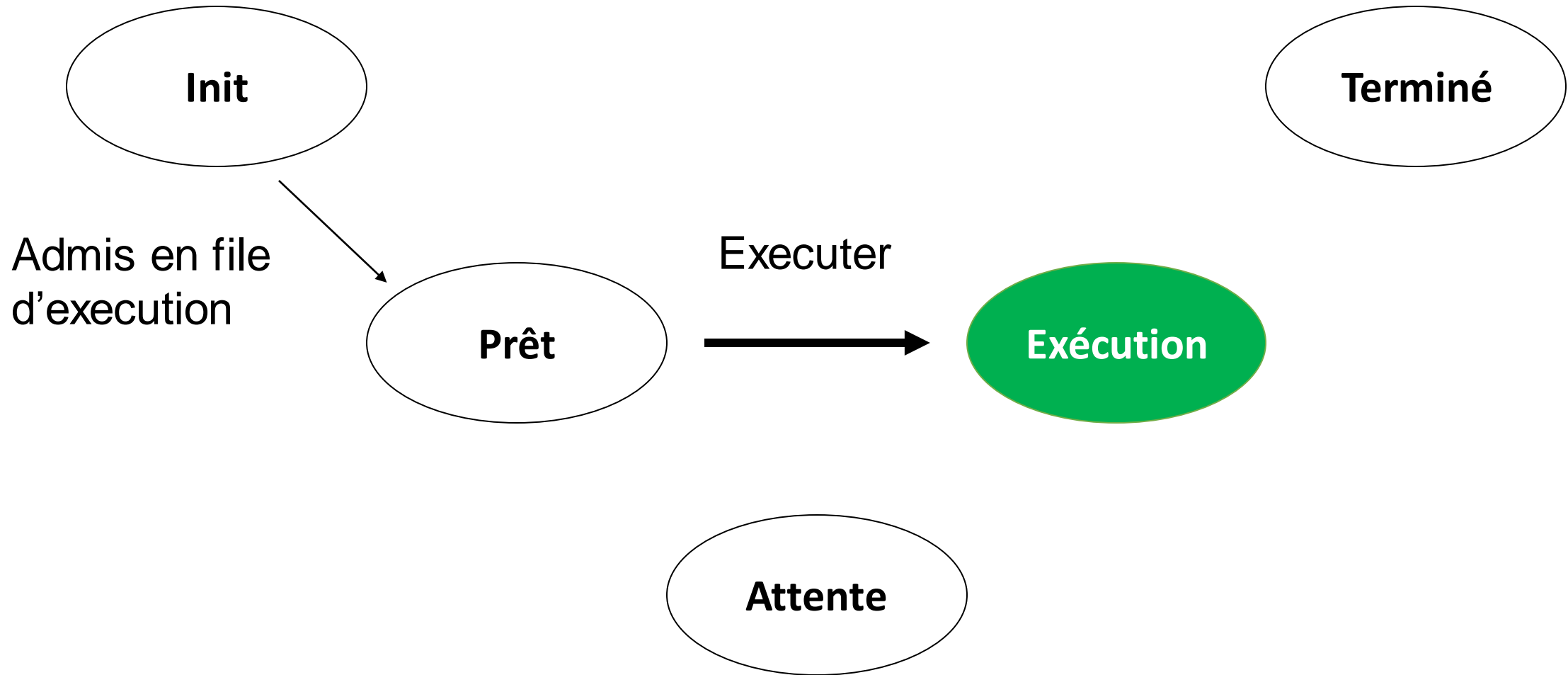
Processus est prêt



PCB: en file d'exécution

Registres: Pile d'interruption (noyau)

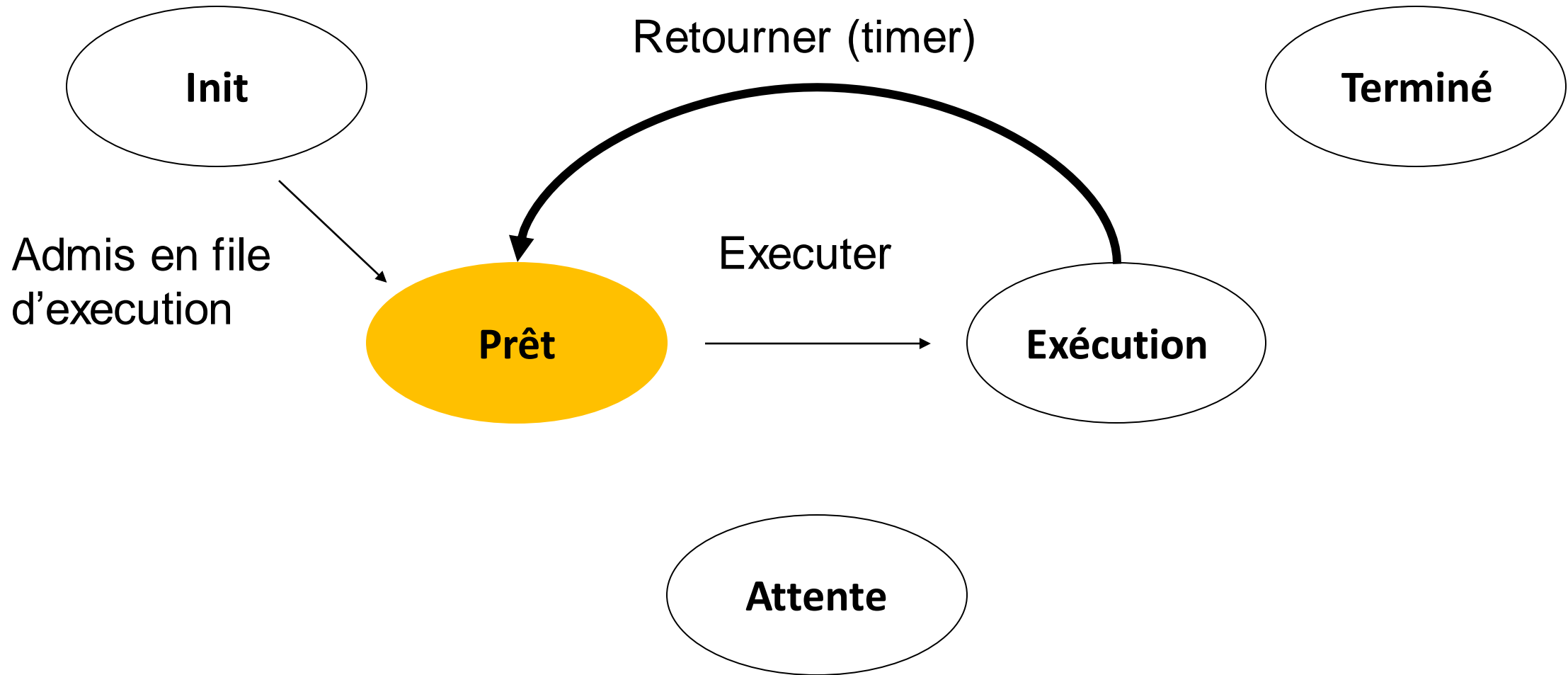
Processus en execution



PCB: En execution

Registres: au niveau du CPU

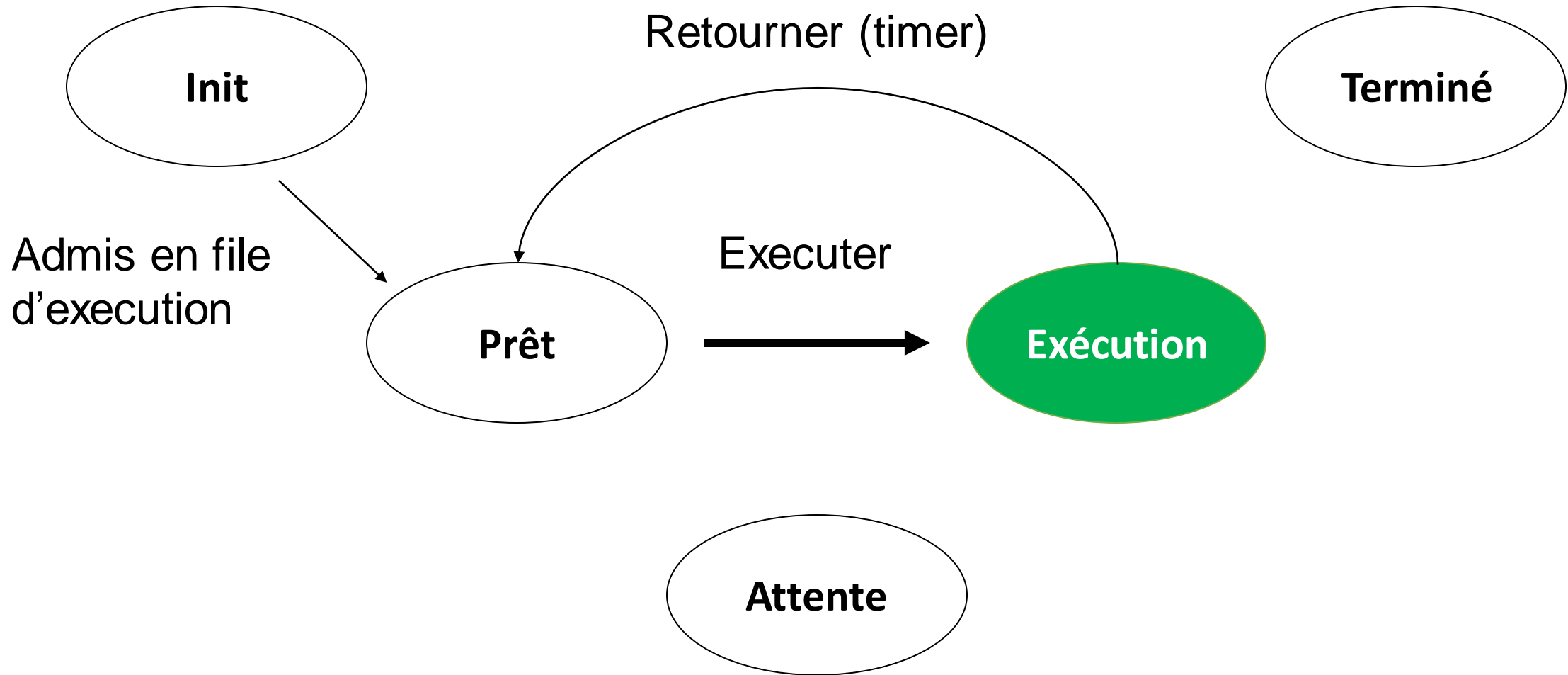
Processus prêt (interruption temps)



PCB: en fil d'exécution

Registres: enregistré sur pile d'interruption (sp enregistré sur PCB)

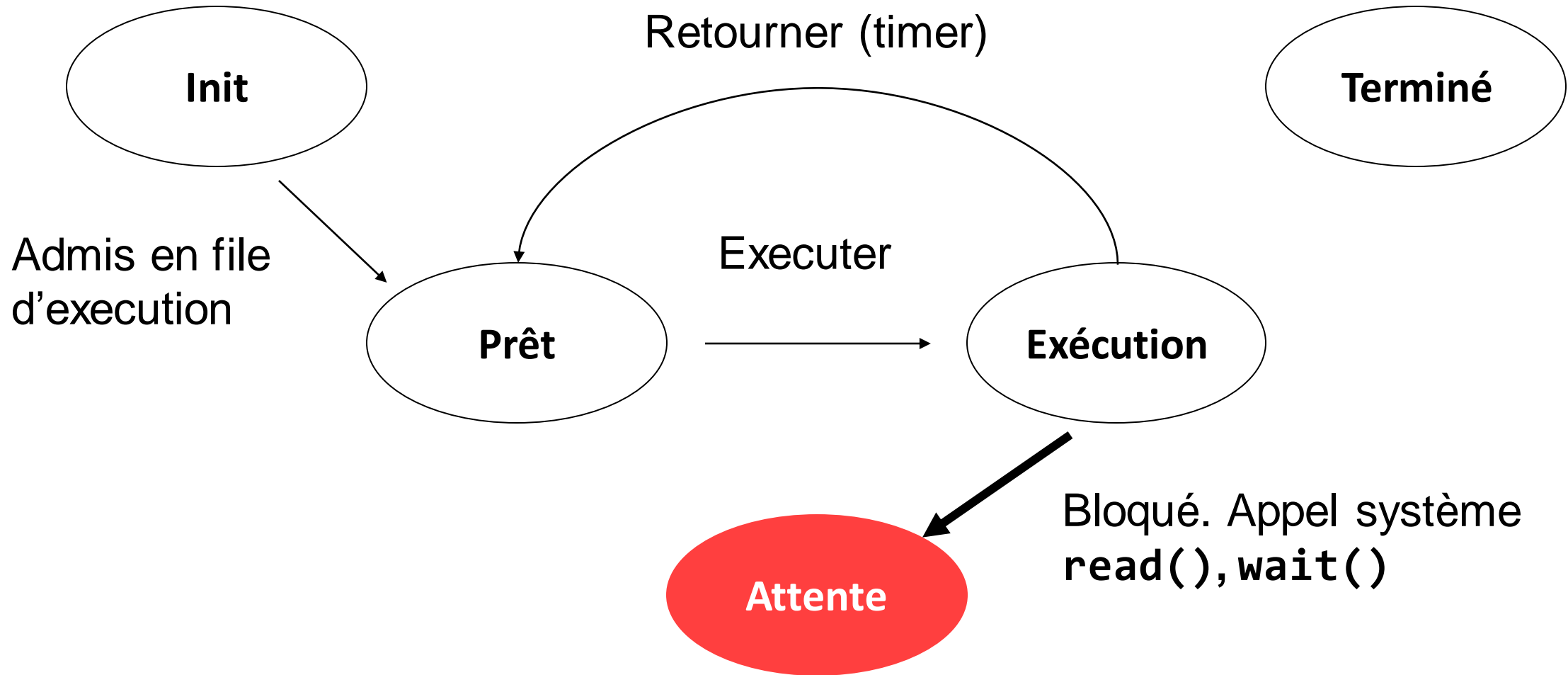
Processus en execution



PCB: En execution

Registres: Restaurés du PCB

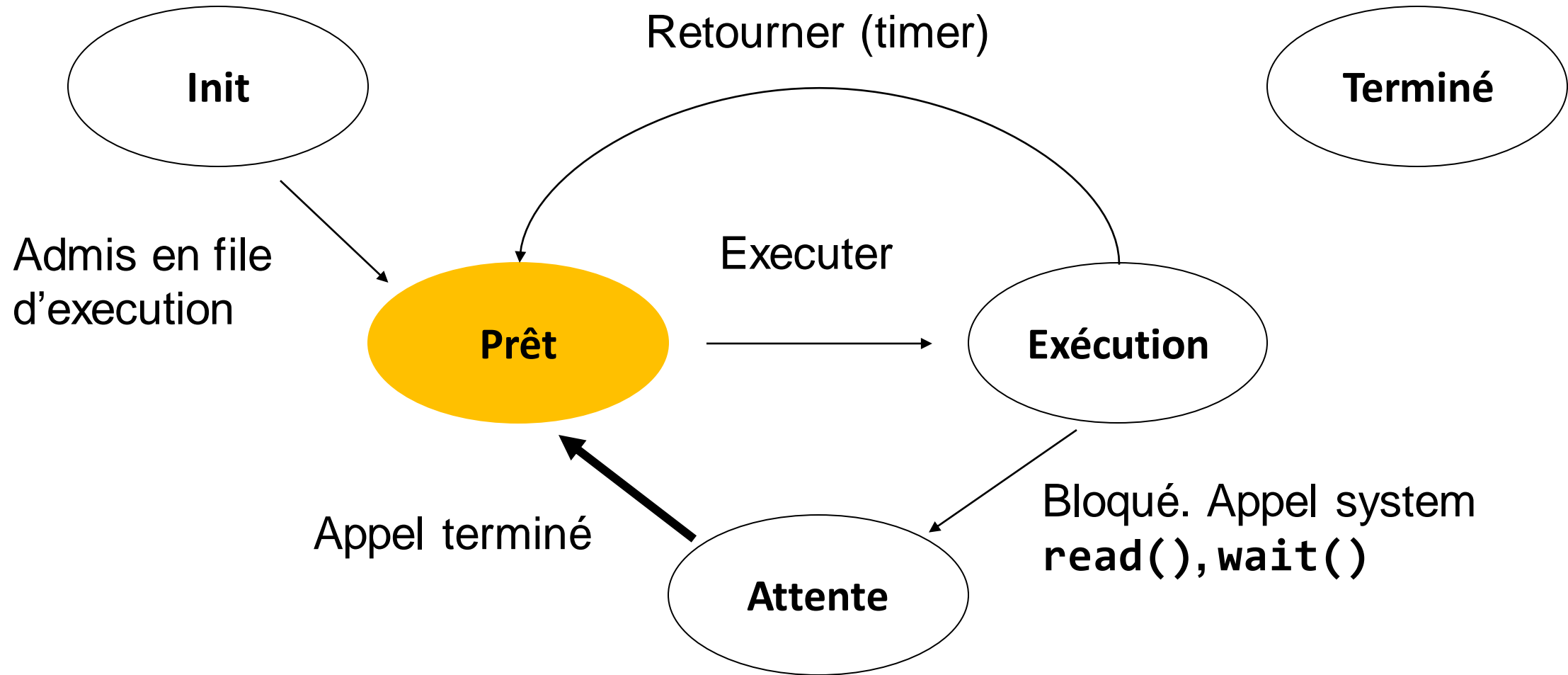
Processus en attente (appel système)



PCB: en fil d'attente spécifique

Registres: pile d'interruption

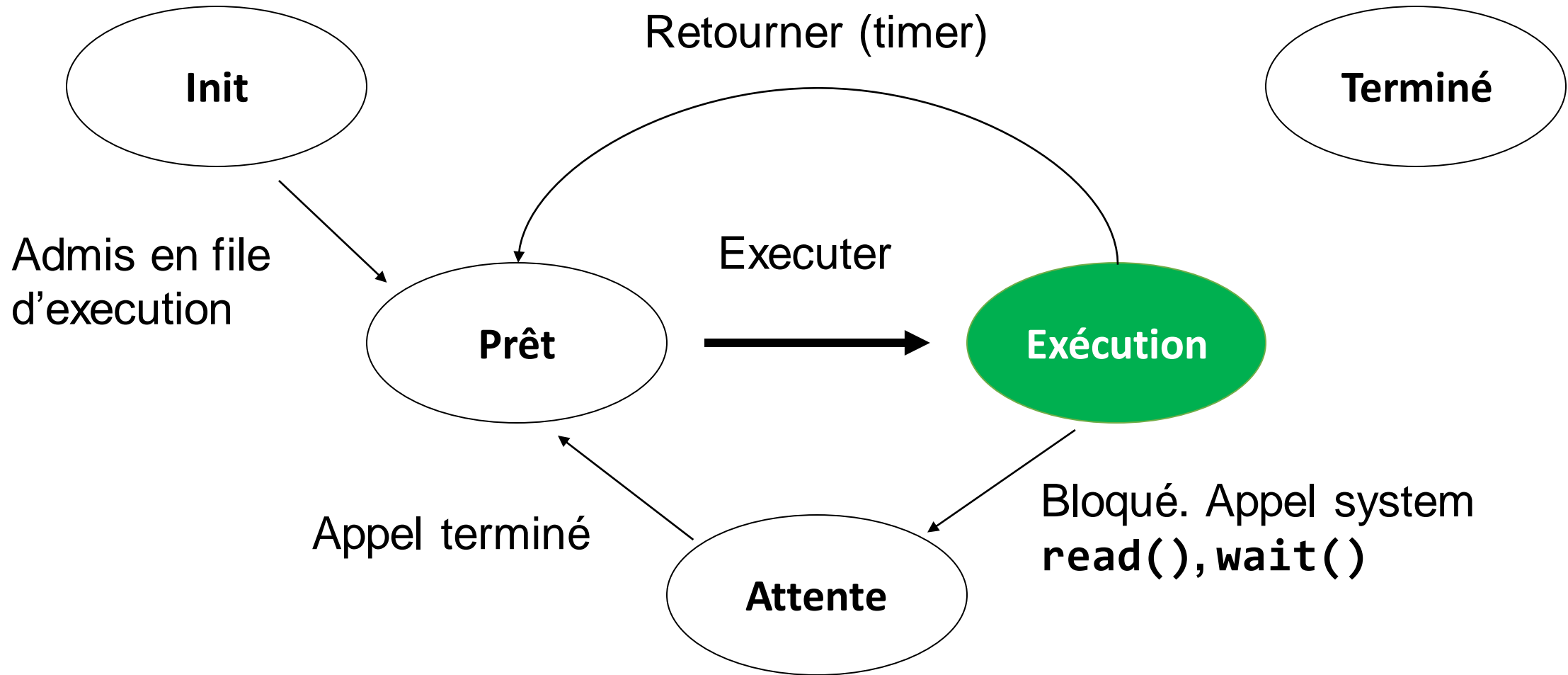
Processus prêt (après interruption)



PCB: en fil d'exécution.

Registres: pile d'interruption.

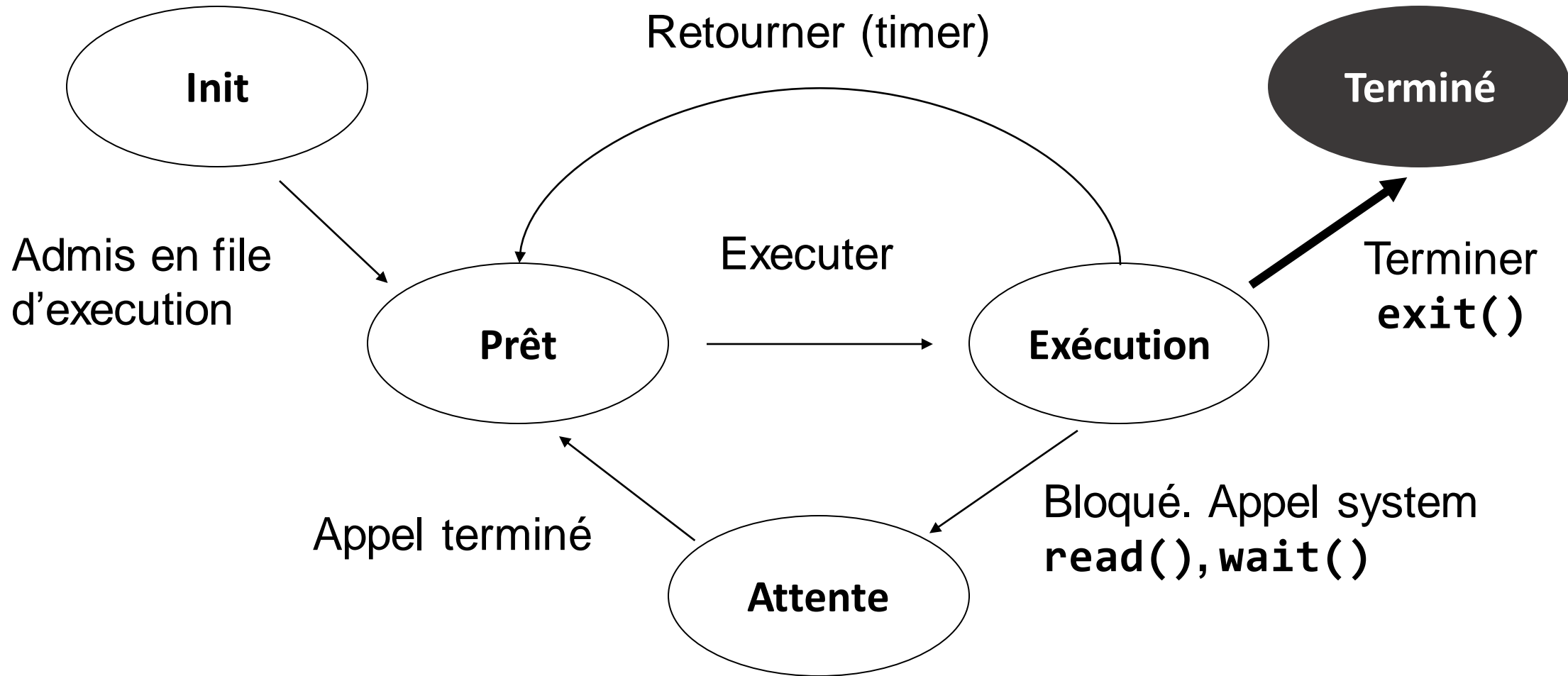
Processus en execution



PCB: en execution.

Registres: Réinitialisés vers CPU.

Processus terminé



PCB: supprimé.

Registres: pas en utilisation

Processus terminé: Etat Zombie

Un processus fils Zombie peut être nettoyé par:

1. Un autre processus. Verifie l'existence de Zombie avant d'entrer en **état d'exécution**.
2. Ou par un processus père après sa terminaison.

Si le père termine avant son fils?

- Si le processus père termine son execution avant, le processus fils est adopté par le processus (initial, PID=1).

Opérations sur les processus

- Ces opération sont disponible sur les tous les systèmes d'exploitation modernes.
- **Create:** Créer un processus.
- **Destroy:** Tuer un processus.
- **Wait:** Attendre qu'un processus termine son execution.
- **Miscellaneous Control:** Autres operations de controle. Par exemple, suspendre un processus puis continuer son execution.
- **Status:** Obtenir des informations sur un processus

Création d'un processus

- Affecter un identificateur unique au processus (PID).
- Initialiser le descripteur (PCB).
- **Qui peut créer des processus?**
- Tous les processus : processus systèmes et processus utilisateurs,

The `fork()` System Call

- Créer un nouvel processus (UNIX/Linux) en utilisant `fork()`
- Processus créant est appelé: **Père**.
- Le processus créé est appelé: **Fils**.
- Le processus fils est quasiment la même copie que son **père**.
- Chaque processus est identifié par un **PID** (Process IDentification).
- Le Processus fils hérite tout l'environnement du processus père (priorité, code, copie de la zone des données, copie de la pile, descripteurs des fichiers ouverts, ...).

Exécution des processus père et fils

- Processus fils démarre son exécution après `fork()`.
- Le père reprend son exécution au même point que le fils.

Differencier entre processus père et fils

- **int p = fork();**

Utiliser la valeur retournée (code de retour) par la fonction `fork()` :

P = -1 Le fils n'a pas été créé : erreur;

P = 0 On est dans le programme du fils;

P > 0 On est dans le programme du père,

Exemple d'utilisation N°1


```
int main(int argc, char *argv[]) {  
    printf("hello world (pid:%d)\n", (int) getpid());  
    int p = fork(); ← Création d'un nouveau processus  
    if (p < 0) { // fork failed; exit  
        fprintf(stderr, "fork failed\n");  
        exit(1);  
    } else if (p == 0) { // child (new process)  
        printf("hello, I am child (pid:%d)\n", (int) getpid());  
    } else { // parent goes down this path (main)  
        printf("hello, I am parent of %d (pid:%d)\n", p, (int)  
getpid());  
    }  
    return 0;  
}
```

Processus fils
(p = 0)

Processus père
(p > 0)

Exemple d'utilisation N°1 (code exécuté)

/* Processus père */



```
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int)
getpid());
    int p = fork();
    if (p < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (p == 0) {
        printf("hello, I am child
(pid:%d)\n", (int) getpid());
    } else {
        printf("hello, I am parent of %d
(pid:%d)\n", p, (int) getpid());
    }
    return 0;
}
```

/* Processus fils */

```
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int)
getpid());
    int p = fork();
    if (p < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (p == 0) {
        printf("hello, I am child
(pid:%d)\n", (int) getpid());
    } else {
        printf("hello, I am parent of %d
(pid:%d)\n", p, (int) getpid());
    }
    return 0;
}
```

Exemple d'utilisation N°1 (code exécuté)

```
/* Processus père */

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int)
getpid());
    int p = fork();
    if (p < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (p == 0) {
        printf("hello, I am child
(pid:%d)\n", (int) getpid());
    } else {
        printf("hello, I am parent of %d
(pid:%d)\n", p, (int) getpid());
    }
    return 0;
}
```

```
/* Processus fils */

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int)
getpid());
    int p = fork(); ← Création du processus fils
    if (p < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (p == 0) {
        printf("hello, I am child
(pid:%d)\n", (int) getpid());
    } else {
        printf("hello, I am parent of %d
(pid:%d)\n", p, (int) getpid());
    }
    return 0;
}
```

Exemple d'utilisation N°1 (code exécuté)

```
/* Processus père */

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int)
getpid());
    int p = fork();
    if (p < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (p == 0) {
        printf("hello, I am child
(pid:%d)\n", (int) getpid());
    } else {
        printf("hello, I am parent of %d
(pid:%d)\n", p, (int) getpid());
    }
    return 0;
}
```

```
/* Processus fils */

int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int)
getpid());
    int p = fork();
    if (p < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (p == 0) {
        printf("hello, I am child
(pid:%d)\n", (int) getpid());
    } else {
        printf("hello, I am parent of %d
(pid:%d)\n", p, (int) getpid());
    }
    return 0;
}
```


Exemple d'utilisation N°2

```
int main (){
    int i=4, j=10;
    int p;
    p = fork(); ← Création d'un nouveau processus
    j += 2; /* Le père et le fils continuent leur exécution à partir
de cette instruction */
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d; père = %d \n",getpid(), i, j,
getppid());
    return 0;
}
```

Exemple d'utilisation N°2 (code exécuté)

```
/* Processus père */

int main (){
    int i=4, j=10;
    int p;
    → p = fork();
    j += 2;
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

```
/* Processus fils */

int main (){
    int i=4, j=10;
    int p;
    p = fork(); ← Création du processus fils
    j += 2;
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

Exemple d'utilisation N°2 (code exécuté)

```
/* Processus père */

int main (){
    int i=4, j=10;
    int p;
    p = fork();
    j=12 → j += 2;
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

```
/* Processus père */

int main (){
    int i=4, j=10;
    int p;
    p = fork();
    j += 2; ← j=12
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

Exemple d'utilisation N°2 (code exécuté)

```
/* Processus père */
```

```
int main (){
    int i=4, j=10;
    int p;
    p = fork();
    j += 2;
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
    père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

i=8

j=24

```
/* Processus père */
```

```
int main (){
    int i=4, j=10;
    int p;
    p = fork();
    j += 2;
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
    père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

i=7

j=15

Exemple d'utilisation N°2 (code exécuté)

```
/* Processus père */
```

```
int main (){
    int i=4, j=10;
    int p;
    p = fork();
    j += 2;
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
    père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

i=8

j=24

Résultat

proc= 18317, i=8, j=24; père = 18310
proc= 18318, i=7, j=15; père = 18317

```
/* Processus père */
```

```
int main (){
    int i=4, j=10;
    int p;
    p = fork();
    j += 2;
    if (p == 0){
        i += 3; j += 3;
    }
    else{
        i *= 2; j *= 2;
    }
    printf("\n proc= %d, i=%d, j=%d;
    père = %d \n",getpid(), i, j, getppid());
    return 0;
}
```

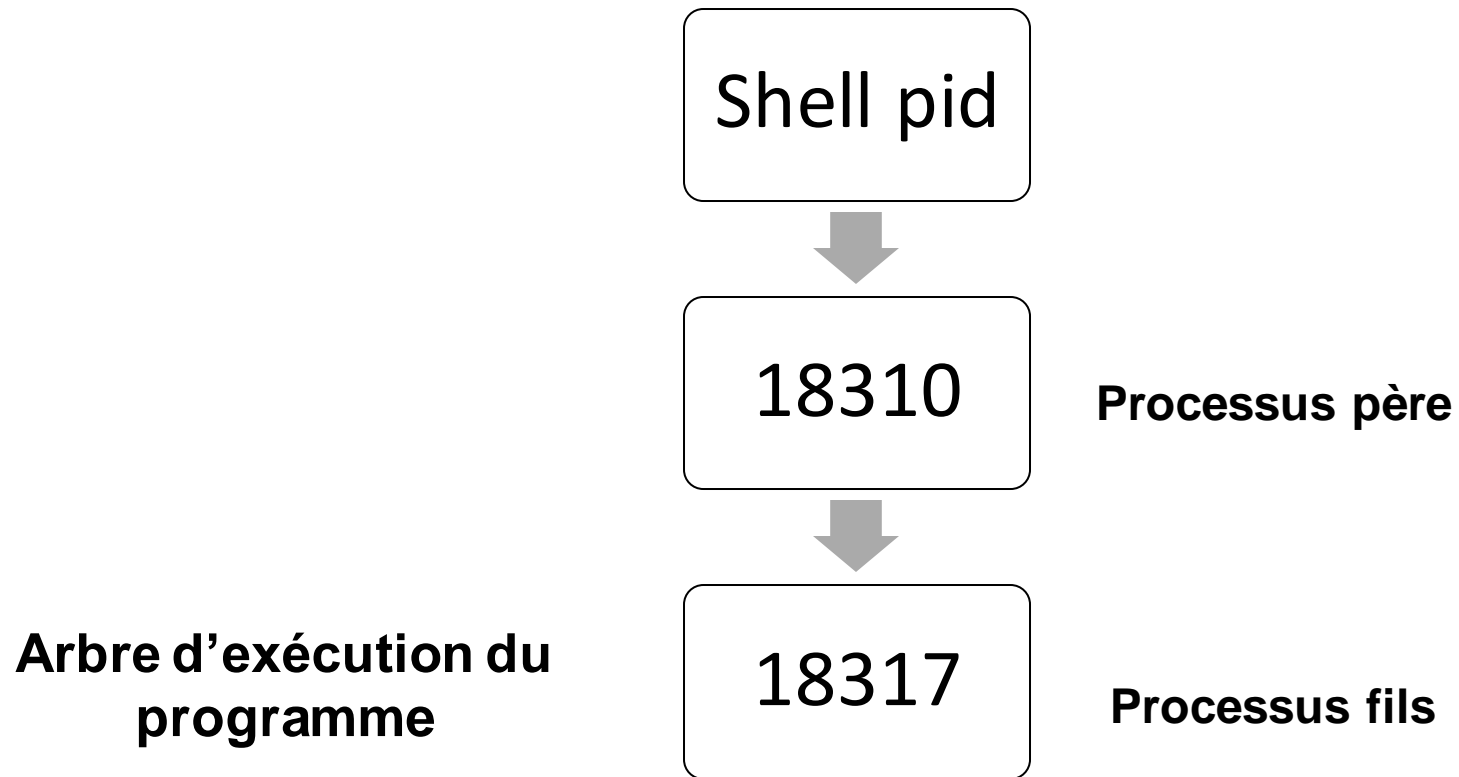
i=7

j=15

Exemple d'utilisation N°2 (arbre d'exécution)

Résultat

proc= 18317, i=8, j=24; père = 18310
proc= 18318, i=7, j=15; père = 18317



Exemple d'utilisation N°3

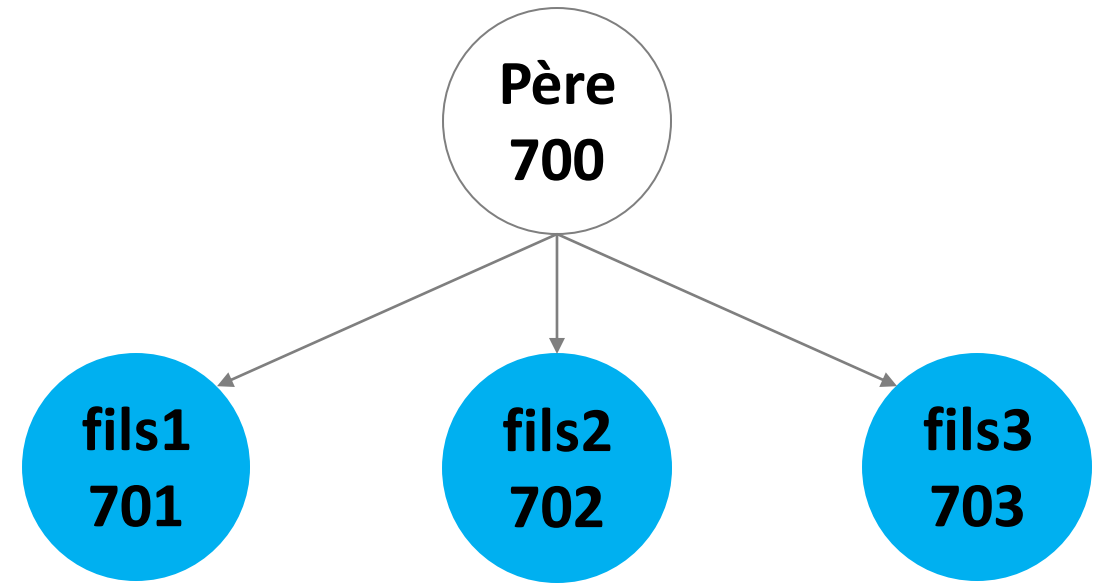
```
Int main(){  
    int p;  
    p=fork();  
    p=fork();  
    p=fork();  
    return 0;  
}
```



Père
700

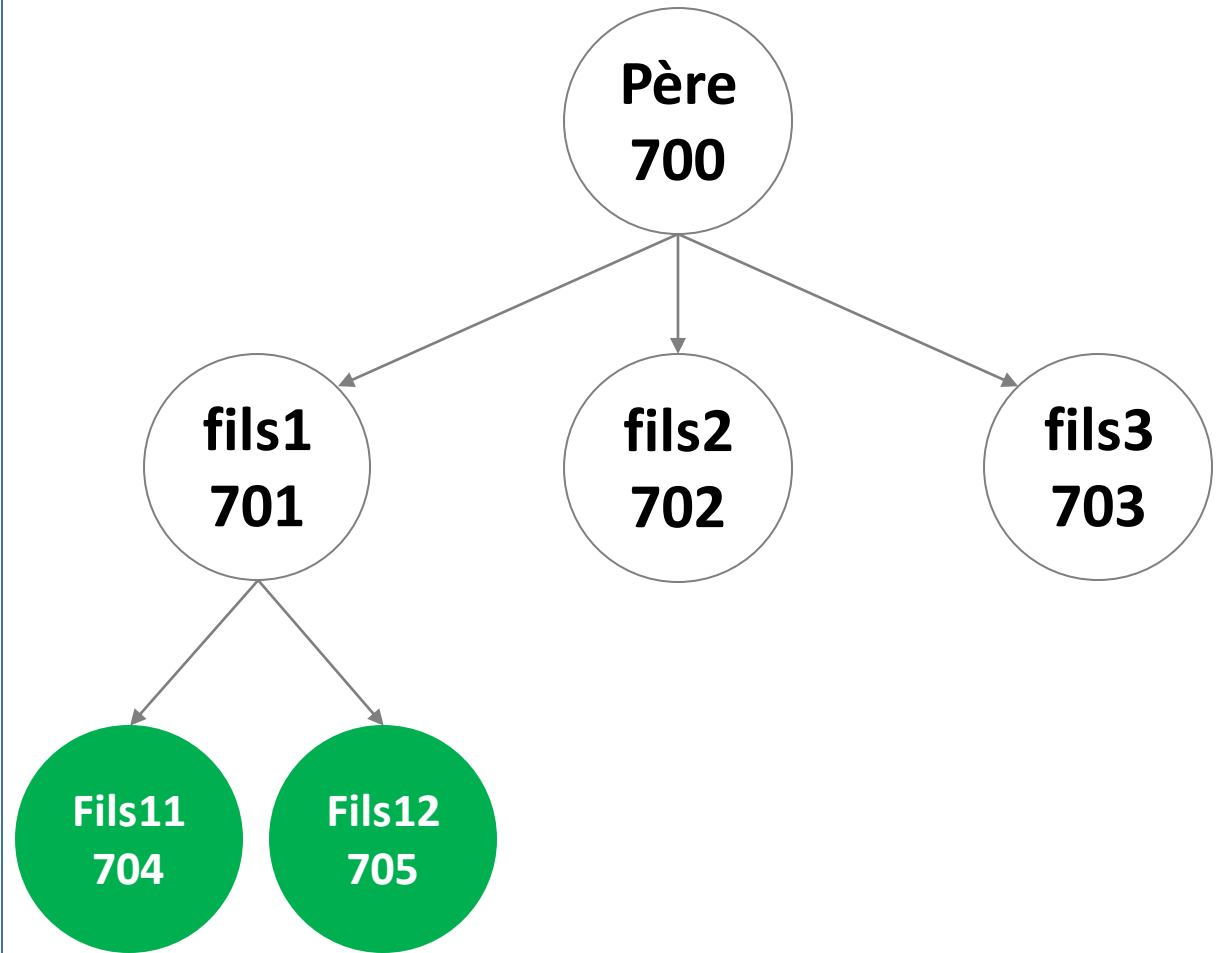
Exécution du processus père

```
Int main(){  
    int p;  
    → p=fork();  
      p=fork();  
      p=fork();  
    return 0;  
}
```



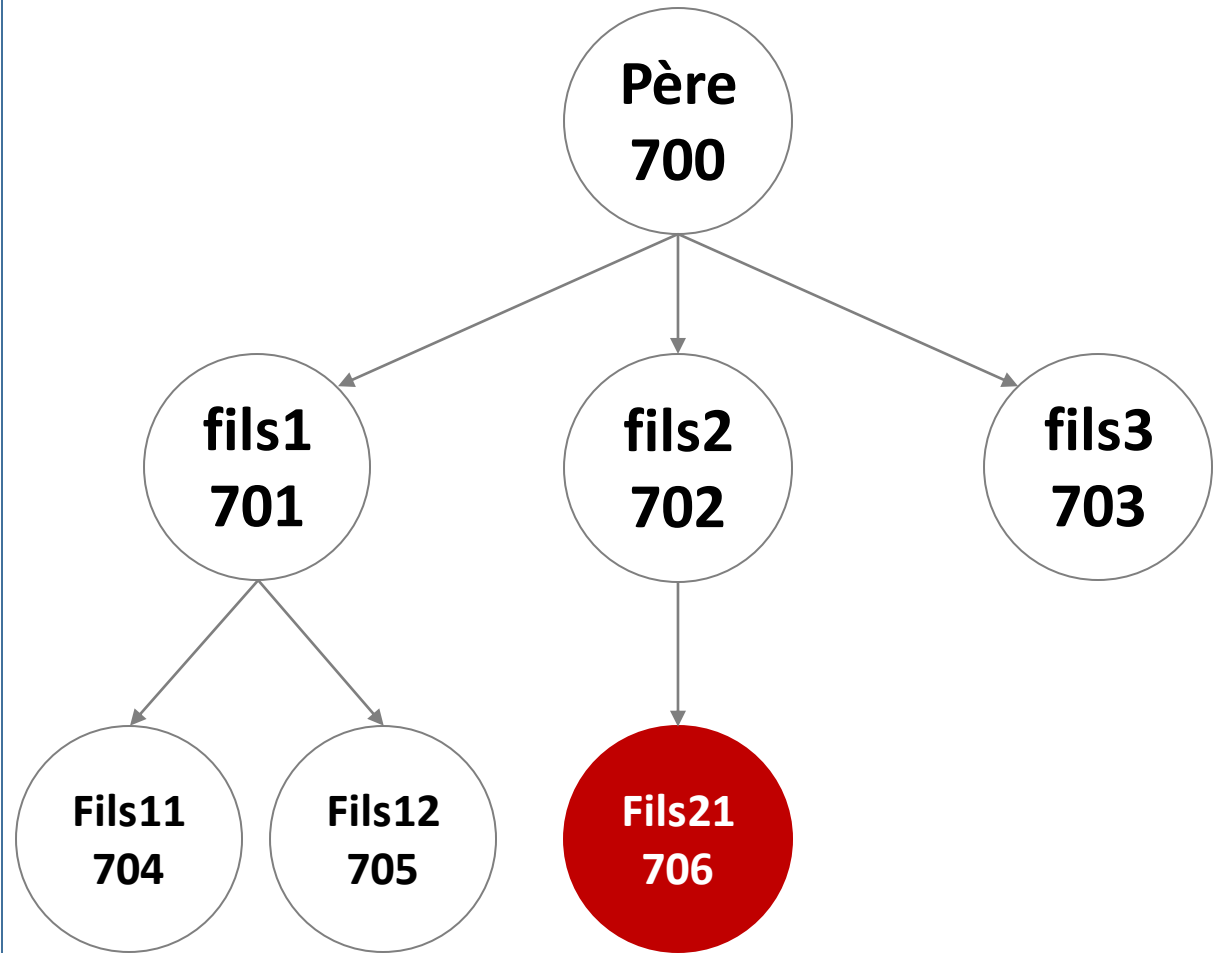
Exécution du processus fils 1 (pid:701)

```
Int main(){  
    int p;  
    p=fork();  
    → p=fork();  
    p=fork();  
    return 0;  
}
```



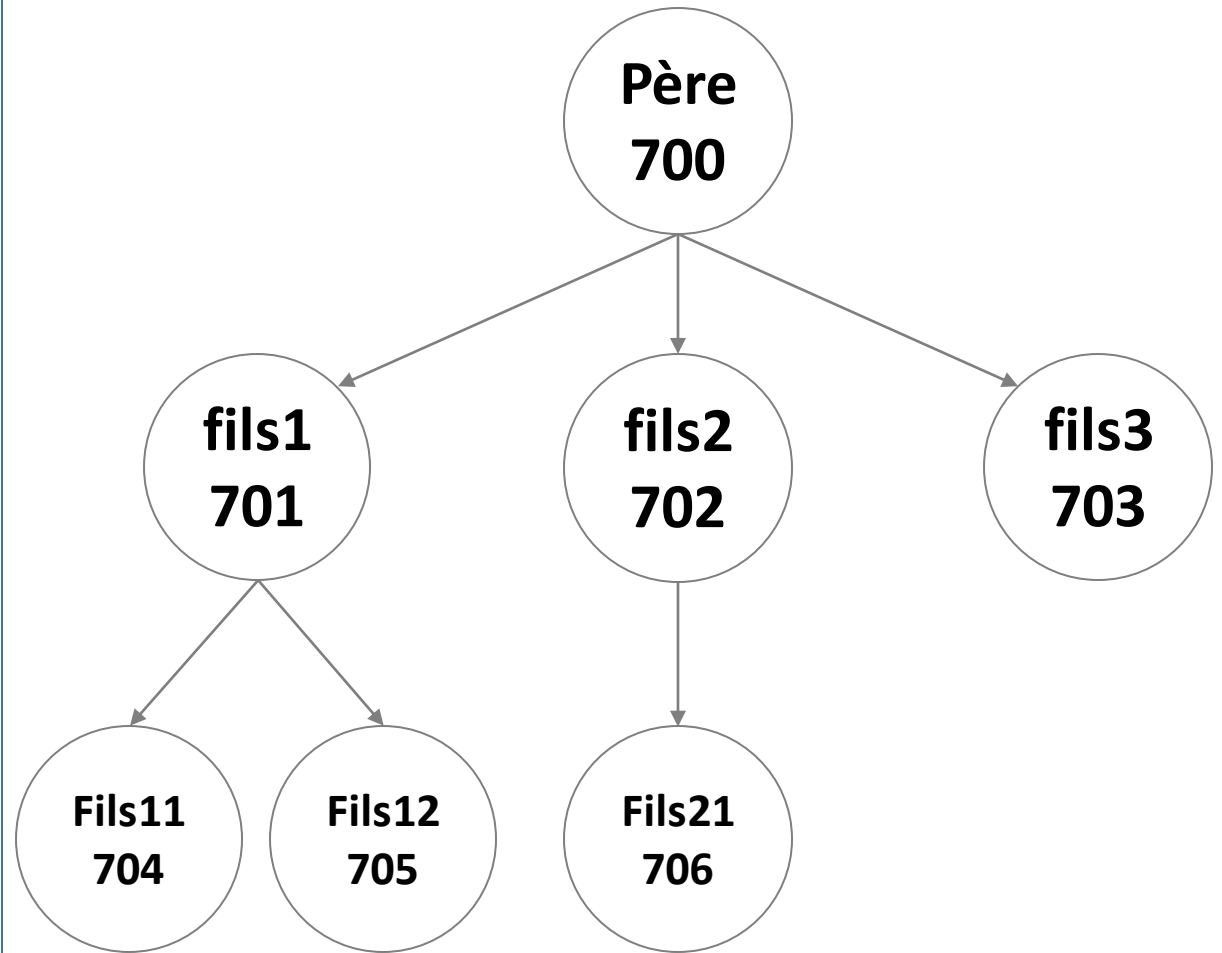
Exécution du processus fils 2 (pid:702)

```
Int main(){  
    int p;  
    p=fork();  
    p=fork();  
    → p=fork();  
    return 0;  
}
```



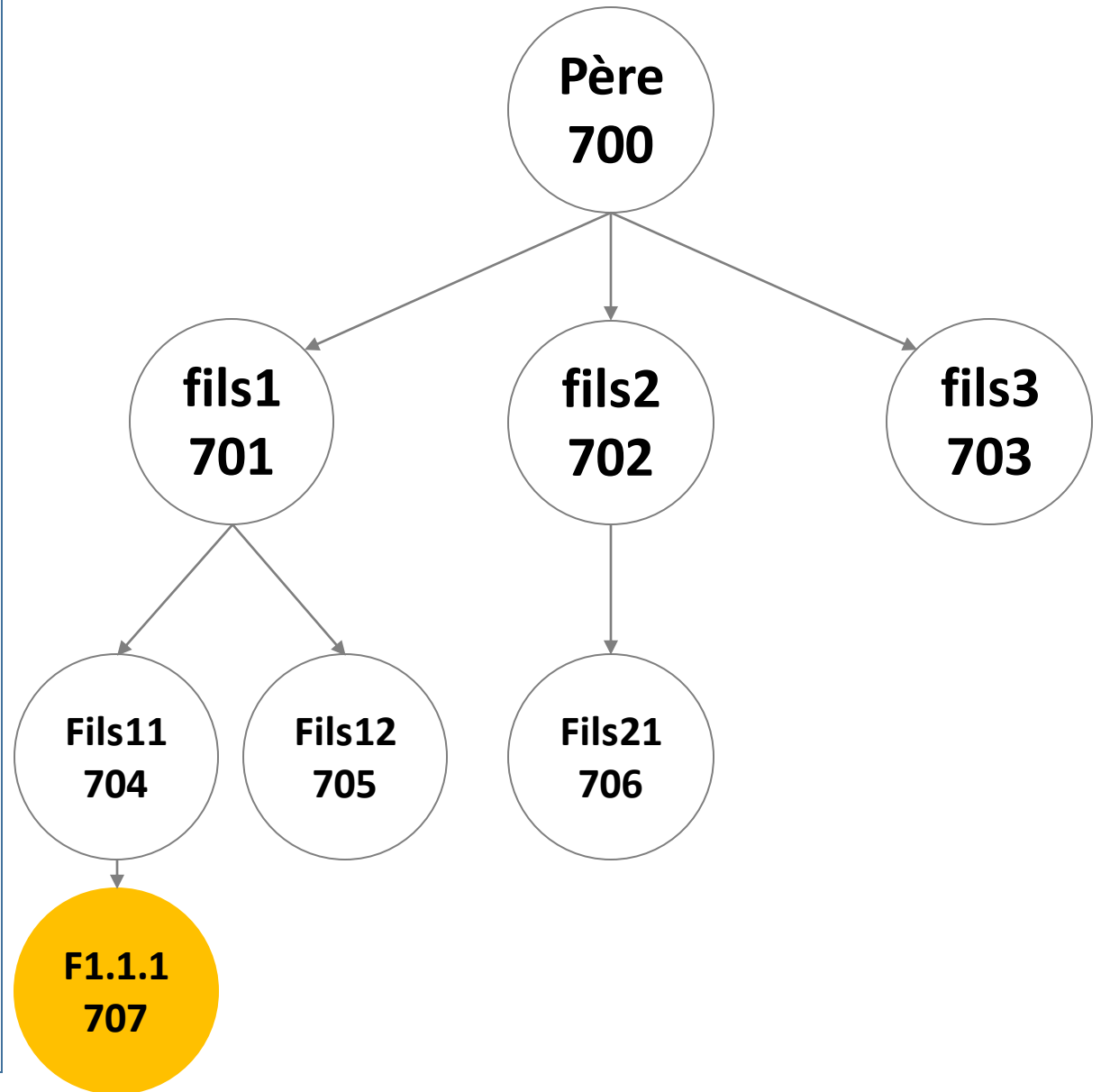
Exécution du processus fils 3 (pid:703)

```
Int main(){  
    int p;  
    p=fork();  
    p=fork();  
    p=fork();  
    return 0;  
}
```



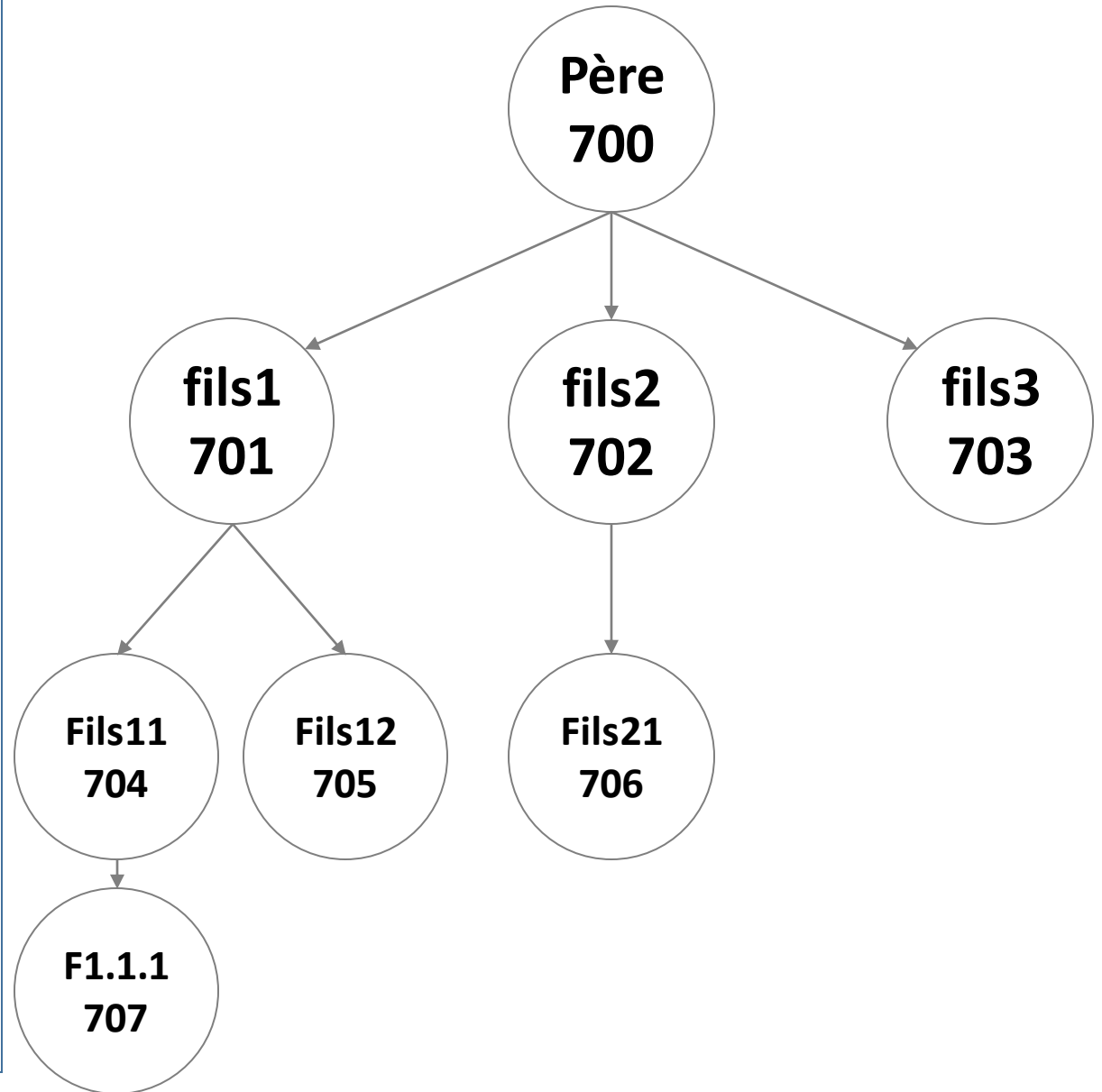
Exécution du processus fils 1.1 (pid:704)

```
Int main(){  
    int p;  
    p=fork();  
    p=fork();  
    → p=fork();  
    return 0;  
}
```



Exécution des autres processus (pid:705, 706, 707)

```
Int main(){  
    int p;  
    p=fork();  
    p=fork();  
    p=fork();  
    return 0;  
}
```



The `wait()` System Call

- Utiliser pour retarder l'exécution d'un processus.
- Processus parent attendant qu'un processus enfant finisse son exécution pour reprendre.

Exemple d'utilisation

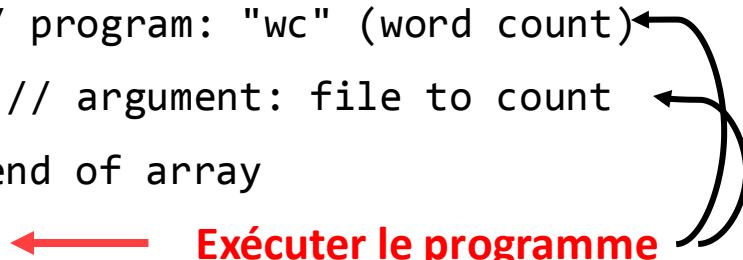
```
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path (main)
        int rc_wait = wait(NULL); ← Attendre la terminaison du Proc fils
        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc,
rc_wait, (int) getpid());
    }
    return 0;
}
```

The `exec ()` System Call

- Utiliser pour executer un autre programme.
- Permet au processus fils d'échapper au similarité du processus père.
- **Six variantes:** `execl`, `execlp ()`, `execle ()`, `execv ()`, `execvp ()`, et `execvpe ()`.

Exemple d'utilisation

```
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc"); // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL; // marks end of array
        execvp(myargs[0], myargs);
    } else { // parent goes down this path (main)
        int rc_wait = wait(NULL);
        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc, rc_wait, (int) getpid());
    }
    return 0;
}
```



Exécuter le programme

Conclusion

- **Etats d'un processus:** Init, prêt, en execution, en attente, terminé
- **Opérations sur un processus**
- **fork()** est utilisé pour créer de nouveaux processus.
- **wait()** permet a un processus père d'attendre la terminaison de ces processus fils.
- **exec()** permet a un processus fils d'échapper au similarité du processus père et d'exécuter un autre programme.