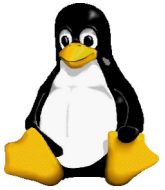
# Système d'exploitation – avancé

## Programmation système

Laurent Mascarilla

2024-2025

# Les systèmes d'exploitation

GNU/Linux



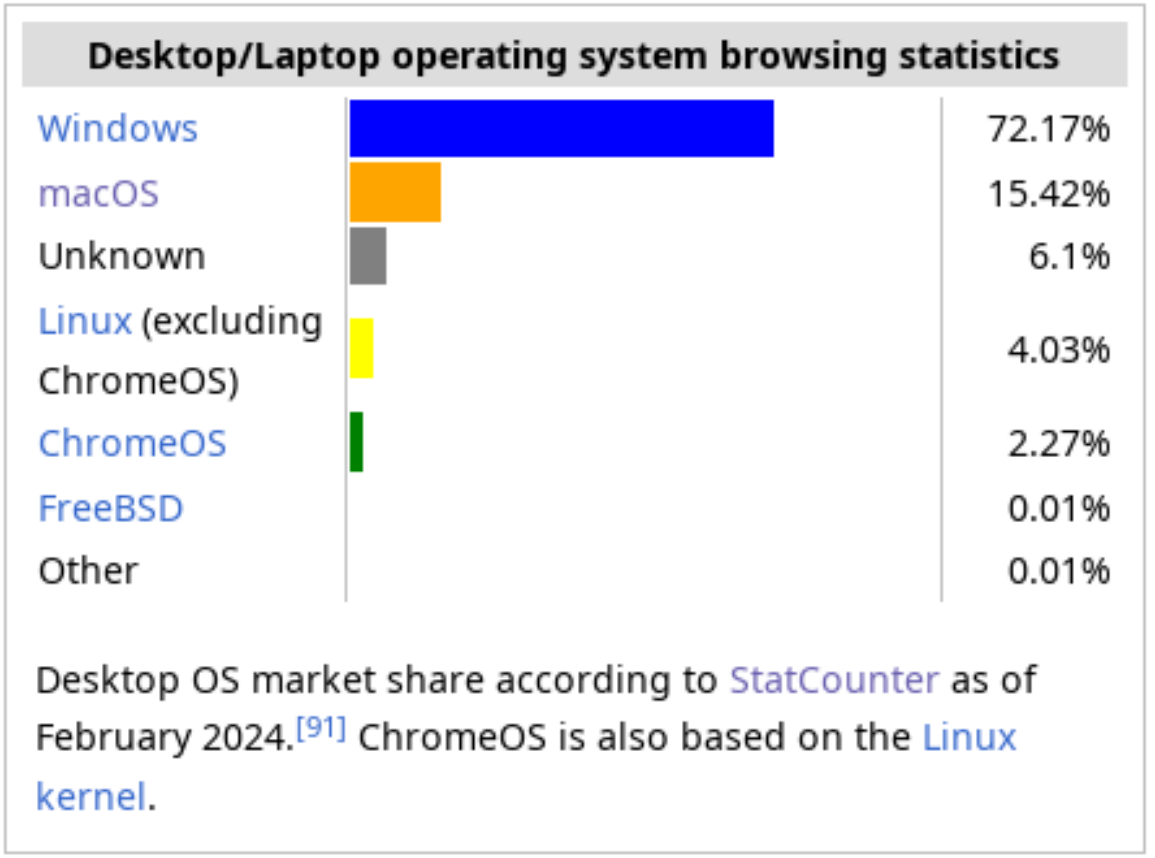



Entre autres....

# Usage share of operating systems(wikipedia)

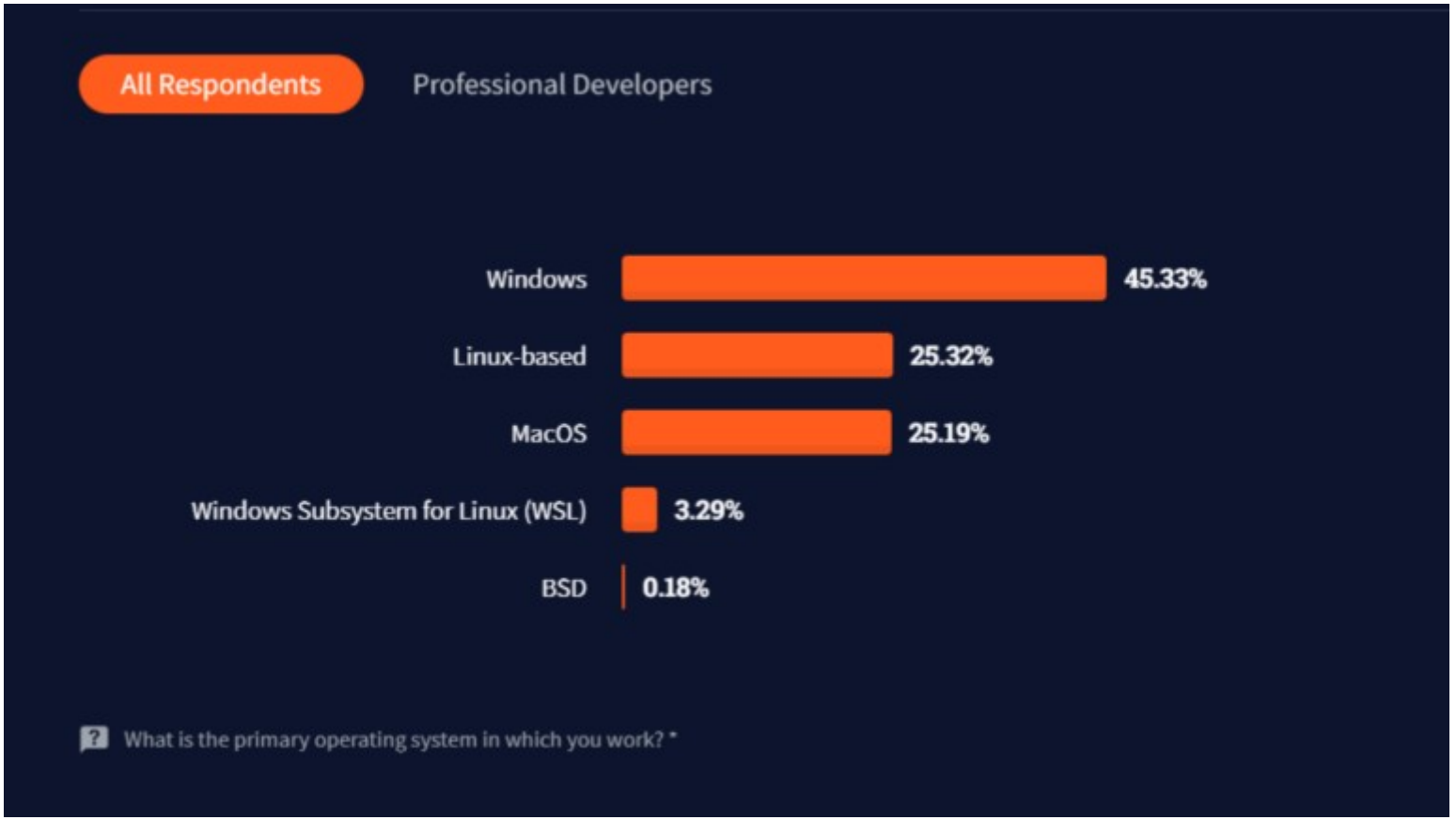
**Desktop / Laptop :**

**Windows est très majoritaire**



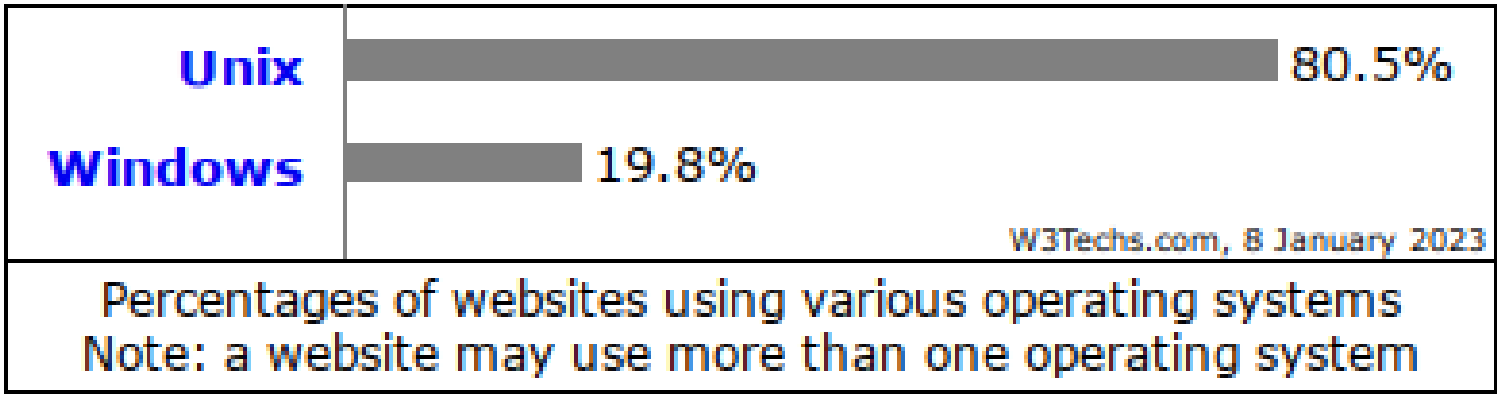
# Côté développeurs..

- **Stackoverflow 2022**



**Serveurs (web ,mail ,dns…)**

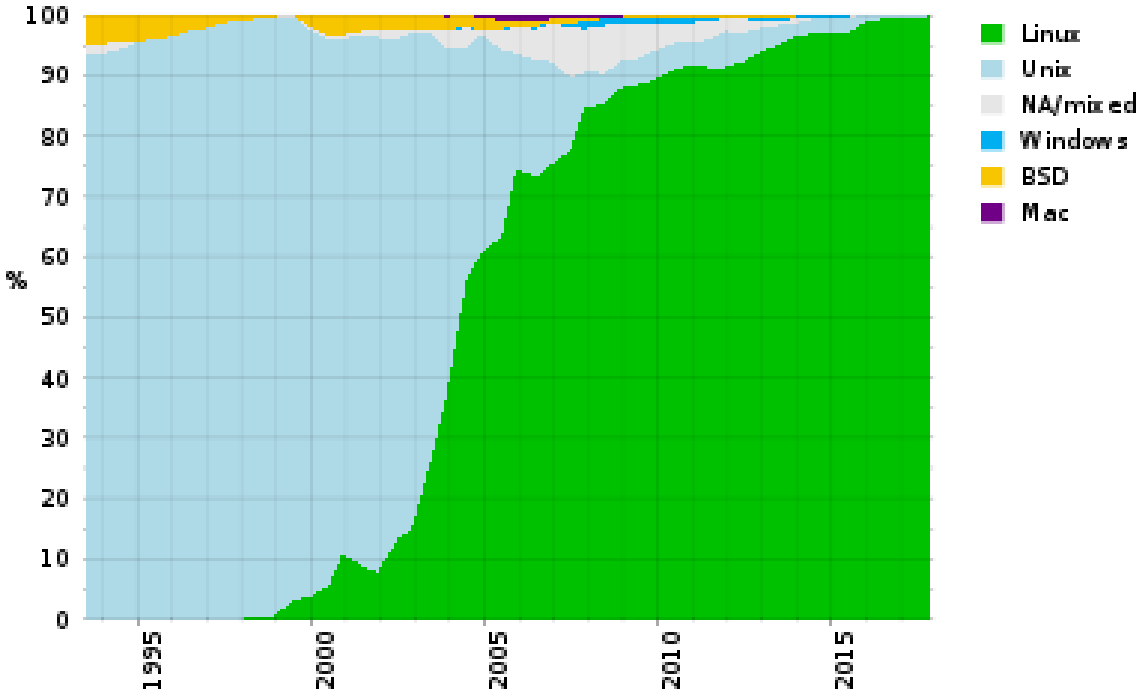
**Source : W3Techs**



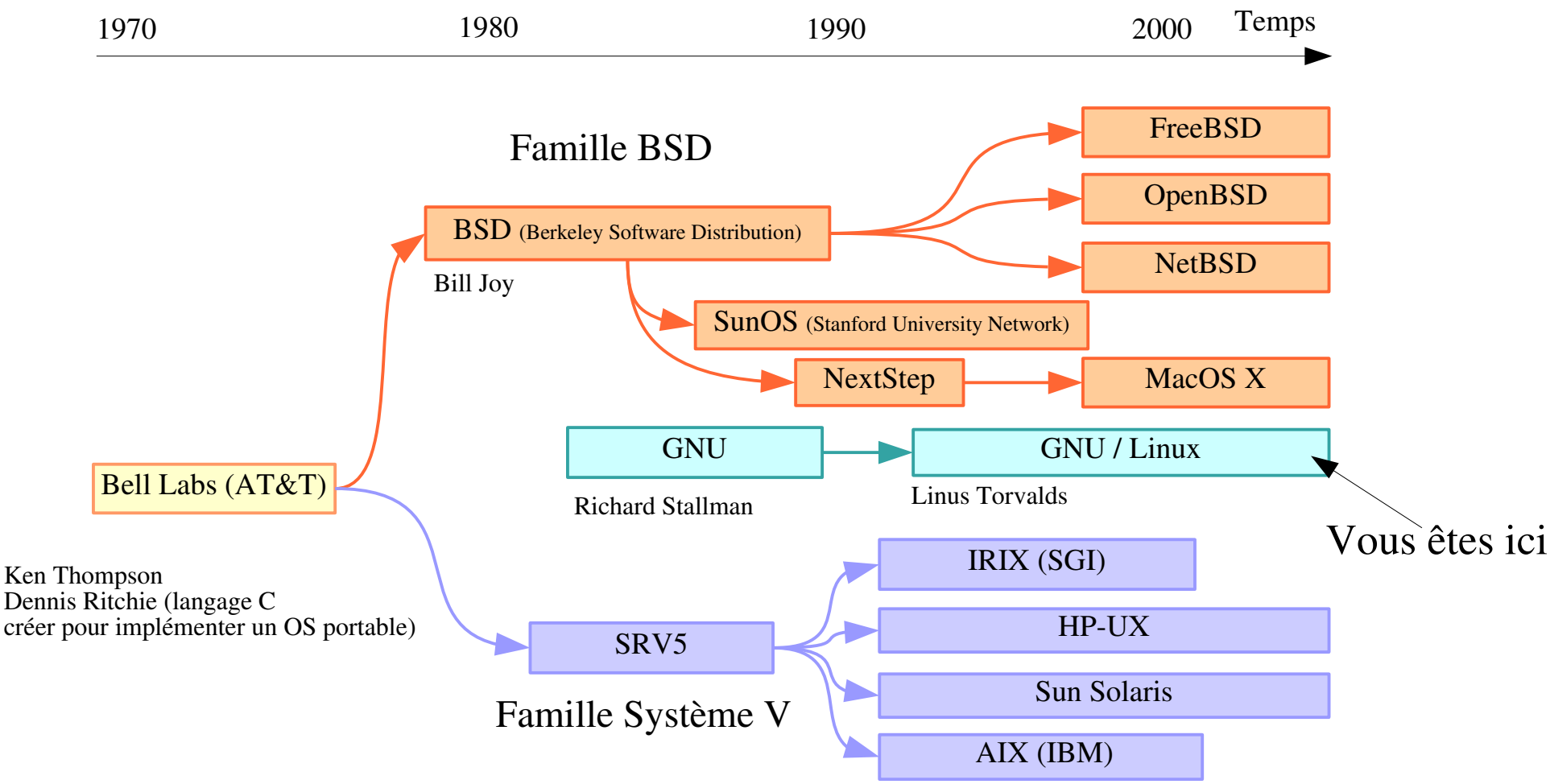

Percentages of websites using various operating systems
Note: a website may use more than one operating system

# Supercalculateurs

| Source | Date | Method | Linux | AIX (Unix) | References |
|---|---|---|---|---|---|
| TOP500 | Nov 2017 | Systems share | 100% | N/A | |
| TOP500 | Nov 2017 | Performance share | 100% | N/A | |
| TOP500 | Jun 2017 | Systems share | 99.6% | 0.4% | |
| TOP500 | Jun 2017 | Performance share | 99.88% | 0.12% | |

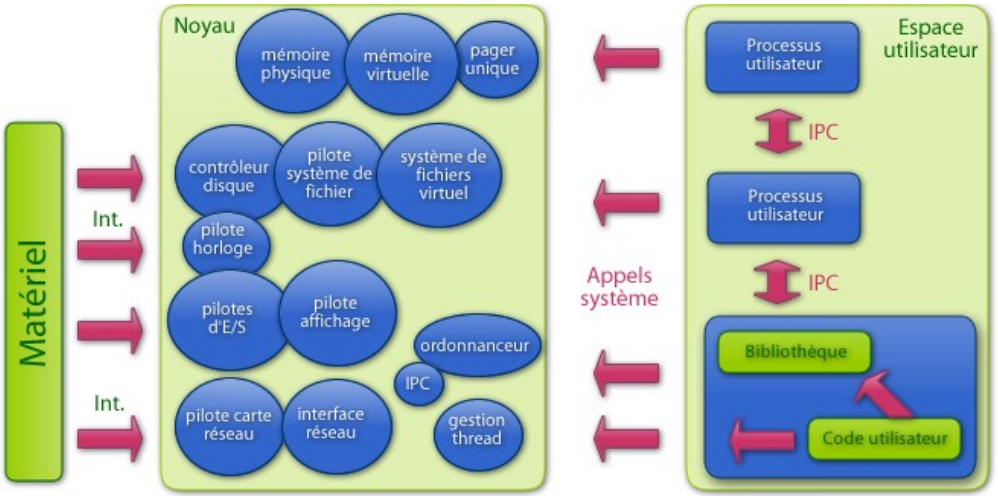


# Arbre généalogique d'Unix



# Noyau/espace utilisateur

- Le noyau d'un système d'exploitation est le logiciel qui assure :
  - la communication entre les logiciels et le matériel
  - la gestion des divers logiciels d'une machine (lancement des programmes, ordonnancement…)
  - la gestion du matériel (mémoire, processeur, périphérique, stockage…)
- Division de l'espace mémoire en un espace noyau et un espace utilisateur




*Noyau Monolithique non modulaire wikipedia*

**Remarque** : aujourd'hui certaines fonctions sont disponibles sous forme de modules (noyaux modulaires)

- Le noyau Linux est écrit en langage C (bientôt partiellement en Rust) # 25 million de lignes !
- L’espace noyau est en mémoire protégée (réservée), isolé des autres programmes
- Les programmes utilisateurs sont dans l’espace mémoire utilisateur (!)
- Séparation essentielle pour la sécurité

« Sous Unix tout est fichier…

… Tout ce qui n’est pas fichier est processus »

# Les processus

# Qu’est ce qu’un processus ?

- Le concept processus est le plus important dans un système d’exploitation (SE) :
  - Tout programme d’un ordinateur est organisé en un certain nombre de processus (processus système, processus utilisateur).
- Un processus (un seul fil d’exécution = thread) est un programme en cours d’exécution :
  - Processus = **instance de programme**
- Un système multi-utilisateurs peut exécuter le même programme dans plusieurs processus, pour plusieurs utilisateurs
- Forment une arborescence (hierarchie)

# Hiérarchie des processus

- Le système d'exploitation fournit un ensemble d'appels système qui permettent la création, la destruction, la communication et la synchronisation des processus.

- Les appels se font en C mais aussi dans tous les langages usuels (Python etc…)

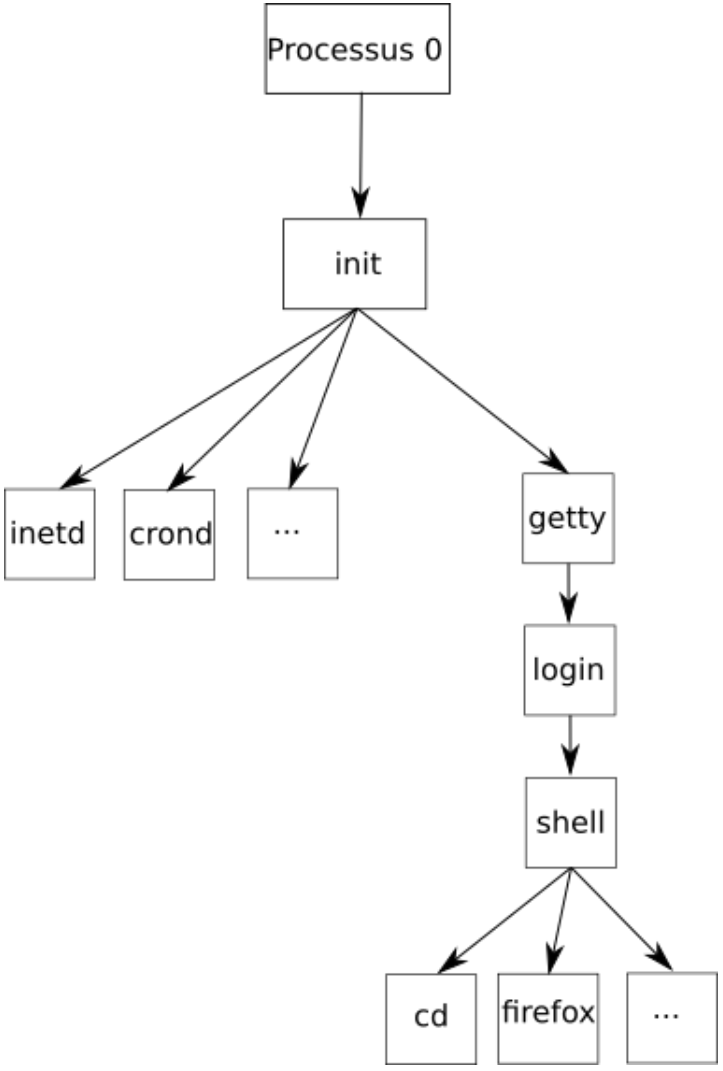
- Les processus sont créés et détruits dynamiquement.

- Un processus peut créer un ou plusieurs processus qui, à leur tour, peuvent en créer d'autres.

- Le programme amorce (boot) charge une partie du système d'exploitation à partir du disque pour lui donner le contrôle. Cette partie détermine les caractéristiques du matériel, effectue un certain nombre d'initialisations et crée le processus 0.

- Le processus 0 réalise d'autres initialisations (ex. le système de fichier) puis crée deux processus : **init** de PID 1 (et le **démon des pages** de PID 2.)

- Ensuite, d'autres processus sont crées à partir du processus init



- Chaque processus s'exécute de manière *asynchrone* et a un numéro d’identification unique (**PID**).
- Chaque processus a un père, à l’exception du premier processus créé (structure arborescente). S’il perd son père (qui se termine), il est tout de suite adopté par le processus **init** de PID 1[(1)].
- Un processus fils peut partager certaines ressources[(2)] (**mémoire**, **fichiers**) avec son processus père ou avoir ses propres ressources. Le processus père peut contrôler l’usage des ressources partagées.

**Remarque (1)** : dans les systèmes récents (notamment basés **systemd**) ce rôle particulier de collecte est délégué à des processus dits **"subreaper"**

**Remarque (2)** : les modalités de partage seront détaillées par la suite : il n’y a pas partage automatique entre père et fils.

- Le processus père peut avoir une certaine « autorité » sur ses processus fils. Il peut les suspendre, les détruire (appel système **kill**), attendre leur terminaison (appel système **wait**) mais ne peut pas les renier :

  Ils sont toujours dans son arborescence

- La création de processus est réalisée par duplication de l'espace d'adressage et de certaines tables du processus créateur (l'appel système **fork**). La duplication facilite la création et le partage de ressources. Le fils hérite des résultats déjà calculés par le père.
- Un processus peut remplacer son code exécutable par un autre (appel système **exec**).

# Caractéristiques d’un processus

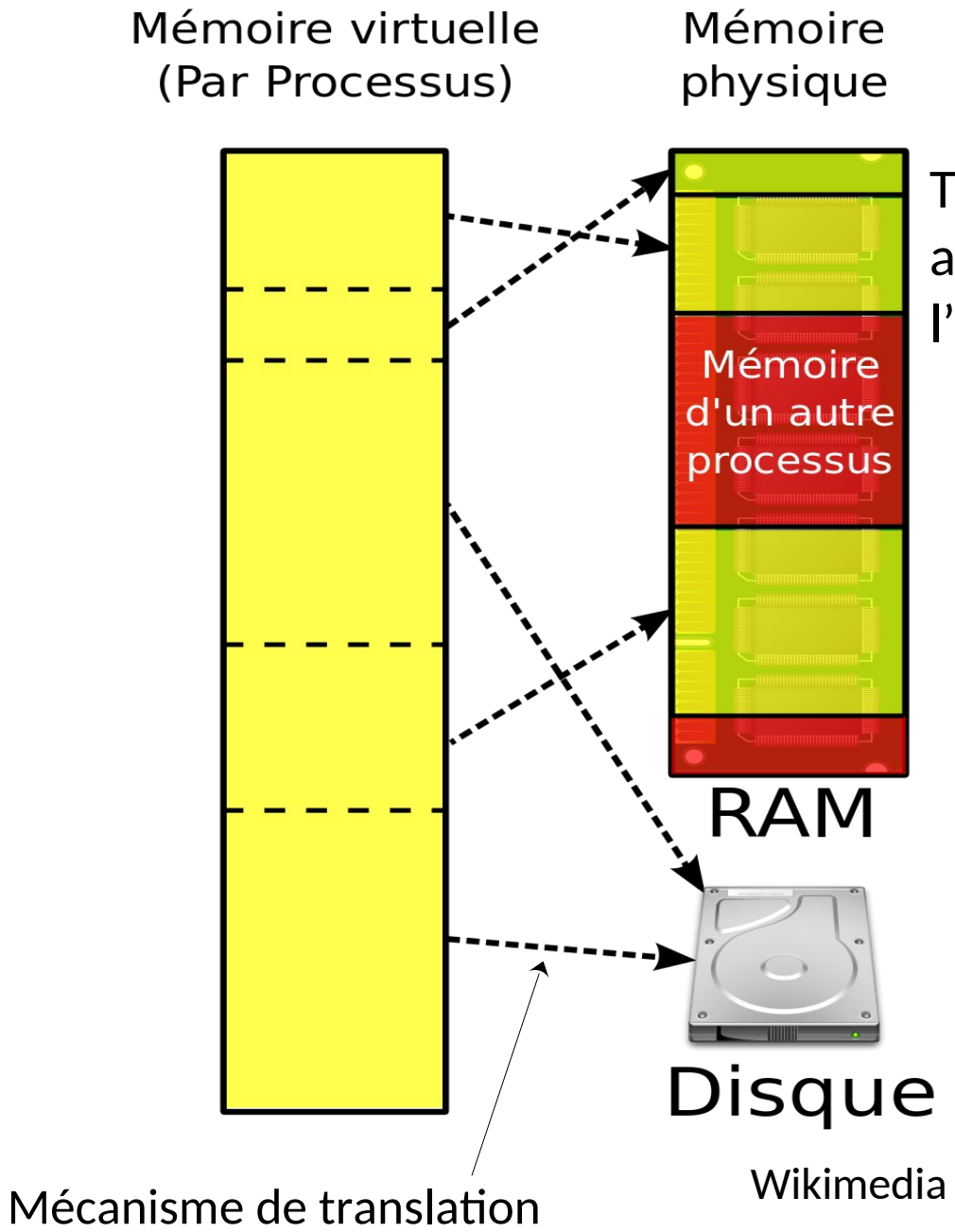
- Un compteur ordinal = pointeur d’exécution = adresse de la prochaine instruction à executer
- Un numéro d’identification unique (PID)
- Un espace d’adressage (code, données, piles d’exécution)
- Un état principal (prêt, en cours d’exécution (élu), bloqué, …)
- Les valeurs des registres lors de la dernière suspension (Sommet de Pile…)
- Une priorité
- Les signaux à capturer, à masquer, à ignorer, en attente ainsi que les actions associées

- Autres informations indiquant, notamment, son processus père, ses processus fils, son groupe, ses variables d'environnement, les statistiques et les limites d'utilisation des ressources….
- Les ressources allouées (fichiers ouverts, mémoires, périphériques …)

L'espace mémoire alloué à un processus est constitué :

- du Code du programme en cours d'exécution (**segment code**)
- des Données manipulées par le programme (**segment data**)
- Des informations relatives au processus : pile d'exécution,…

# Mémoire virtuelle et mémoire physique




Tout se passe comme si chaque processus avait sa mémoire numérotée à partir de l'adresse 0 :

- Abstraction matérielle de la mémoire
- Protection (chaque processus n'accède qu'à sa mémoire)

# Exemple : commande ps

```
$ ps -o uid -o user -o pid -o ppid -o nice -o stime  -o tty -o time -o cmd

 UID  USER        PID      PPID  NI STIME TTY      TIME       CMD
1000 lmascari    5952    5739   0 11:20 pts/1    00:00:21 /usr/bin/zsh
1000 lmascari    6306    1345   0 11:20 pts/1    00:00:00 /usr/bin/zsh
1000 lmascari    6556    1345   0 11:21 pts/1    00:00:00 /usr/bin/zsh
```

•USER : nom de l’utilisateur qui a lancé le processus

•UID : numéro de l'utilisateur qui a lancé le processus

•PID : correspond au numéro du processus

•PPID : correspond au numéro du processus parent

•C priorité : plus la valeur est grande, plus le processus est prioritaire

• NICE : valeur qui  modifie la priorité d'ordonnancement du processus, comprise entre

-20 (la plus favorable) à 19 (la moins favorable).

Valeur par défaut : 0

Une priorité est calculée par l’ordonnanceur à partir de cette valeur.

•STIME correspond à l'heure de lancement du processus

•TTY correspond au nom du terminal

•TIME correspond à la durée de traitement du processus

•COMMAND correspond au nom du processus.

# Table des processus

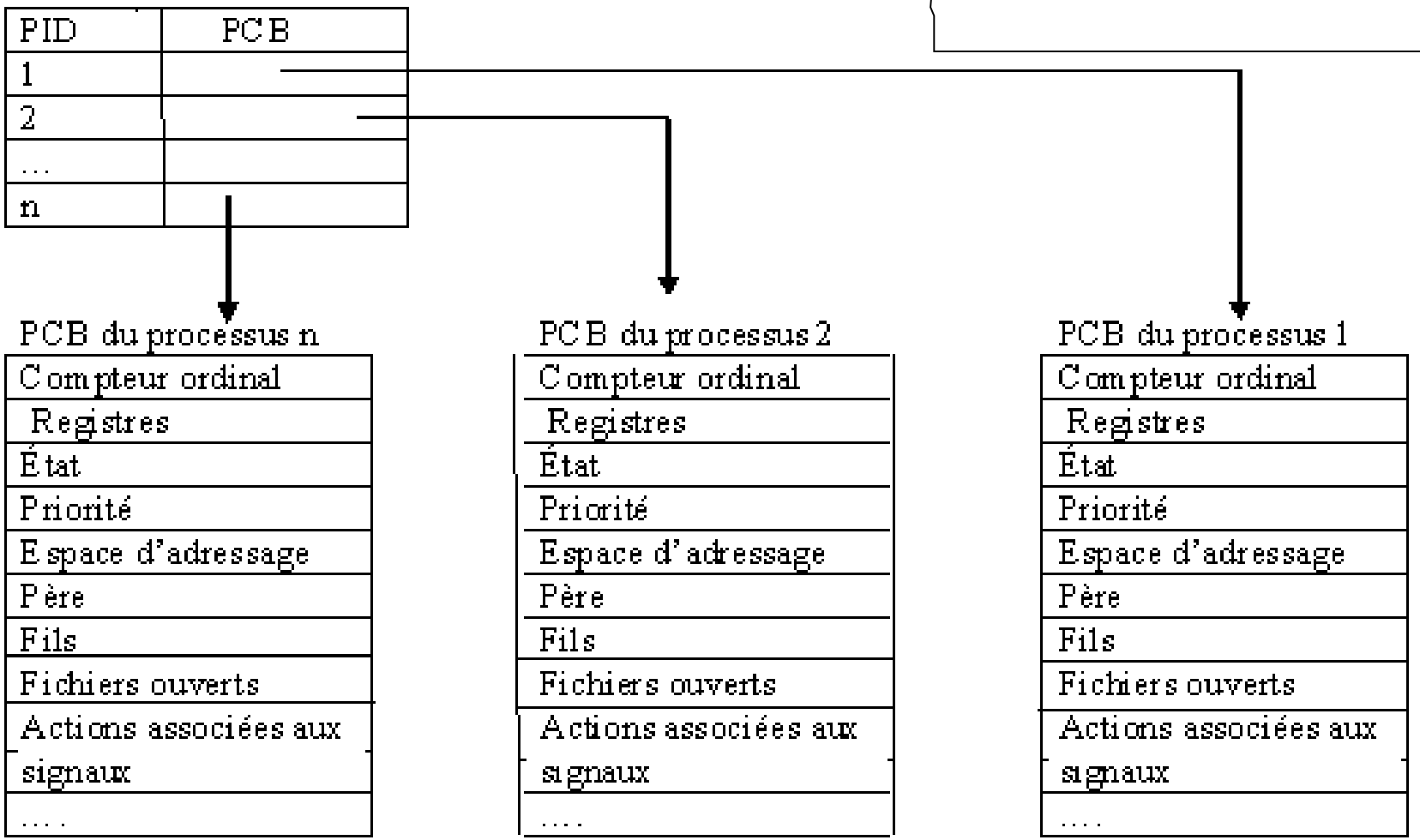
Le système d'exploitation maintient dans une table (au sens d'une structure C) les informations sur tous les processus créés :

- La table des processus :
  - Process Table

- une entrée par processus :
  - Bloc de Contrôle de Processus : PCB (Process Control Block).

# Table des processus/ PCB

PCB =
informations concernant l'état, la mémoire et les ressources du processus

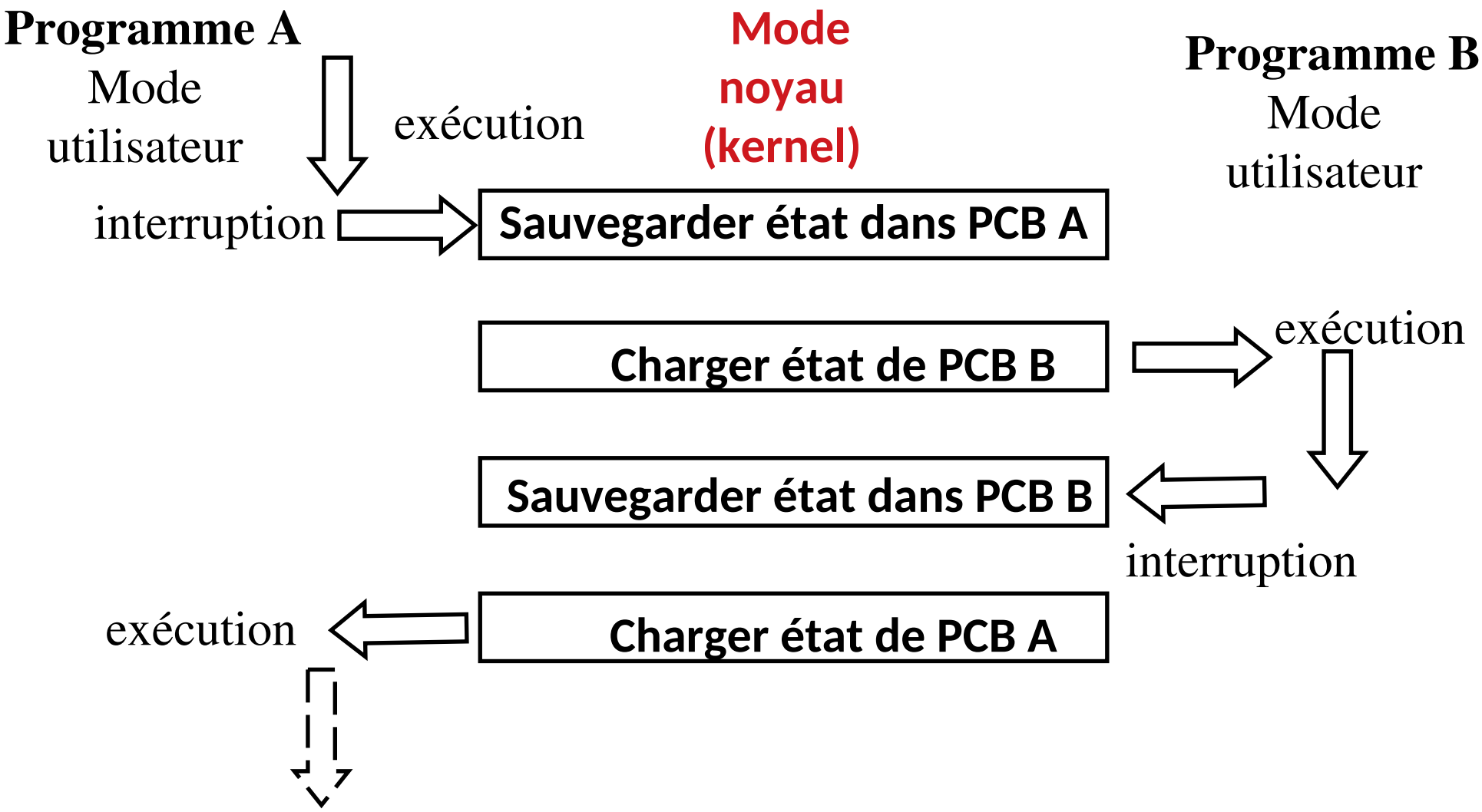


- Dans la plupart des  systèmes d'exploitation, le PCB est composé de deux zones :
  - La première contient les informations critiques dont le système a toujours besoin (toujours en mémoire);
  - La deuxième contient les informations utilisées uniquement lorsque le processus est à l'état élu (i.e. s'exécute).

- Le choix du (des) programme(s) qui est en cours d'exécution est déterminé par un programme appelé ordonnanceur selon un algorithme  (Linux : Completely Fair Scheduler (CFS)) :
  - Changement de contexte

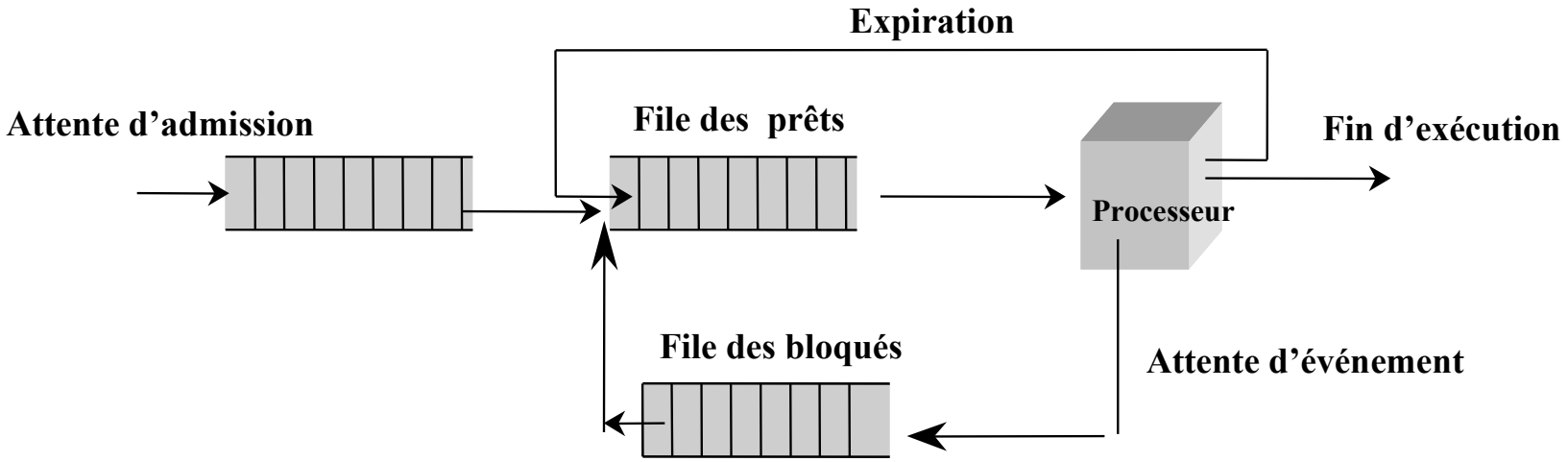
# Changement de contexte

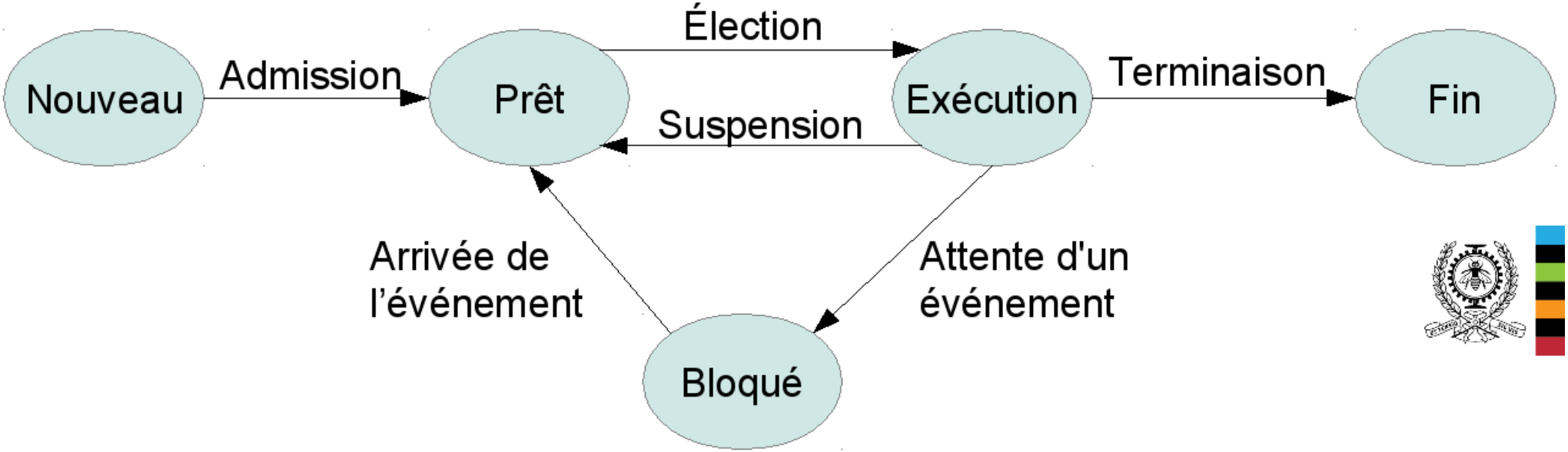
(décidé par l'ordonnanceur (cf. plus tard…))




Un programme change donc d'état (en cours d'exécution, suspendu…) durant son cycle de vie.

# États d'un processus



# Les états des processus Linux

D (TASK_UNINTERRUPTIBLE) :

En sommeil ininterruptible (bloqué sur une E/S par exemple)

R (TASK_RUNNING ) :

En cours d'exécution ou en attente pour l'exécution

S (TASK_INTERRUPTIBLE) :

En sommeil (en attente d'un signal

T (TASK_STOPPED) :

Stoppé

(cf. CTRL+Z, signaux SIGSTOP et SIGCONT)

Z (TASK_ZOMBIE) :

Zombie

D : Mort (on ne le voit jamais)

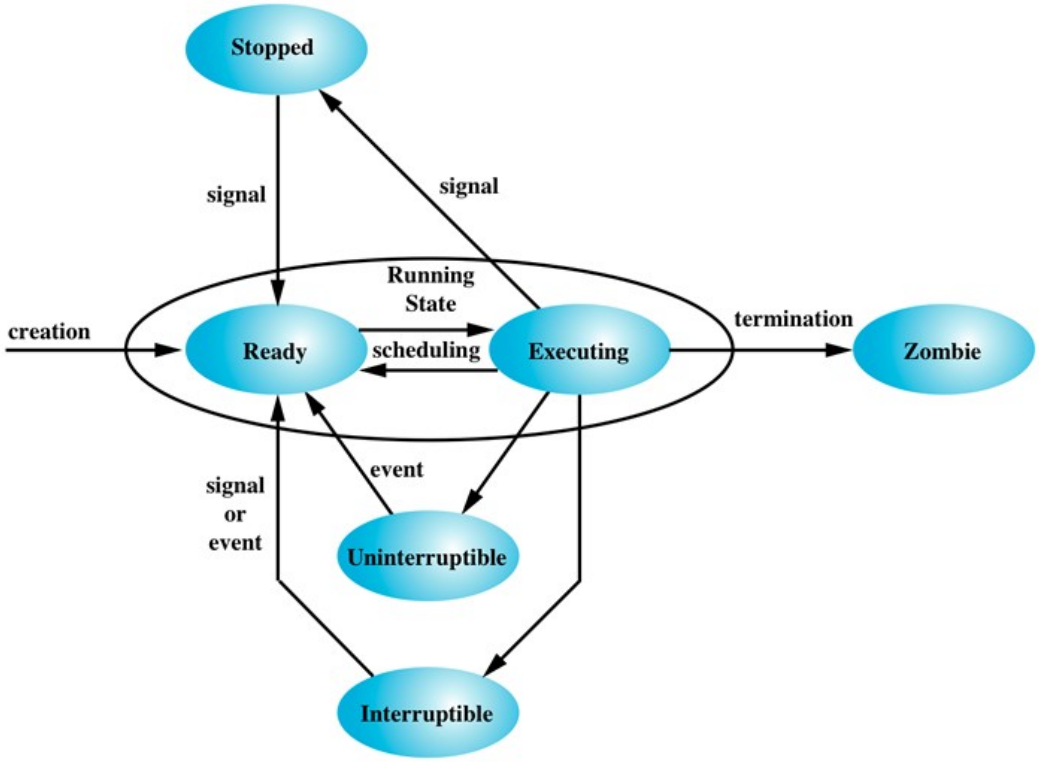



Figure 4.18 Linux Process/Thread Model

# Primitives associées aux processus

| **#include:** `<unistd.h>` | **Primitives** |
|---|---|
| **`pid_t PID`** : numéro de processus,<br>**`pid_t PPID`**: numéro de processus parent,<br>**`uid_t UID`** : numéro de l'utilisateur réel, | **`pid_t getpid(void)`**<br>**`pid_t getppid(void)`**<br>**`uid_t getuid(void)`** |

NAME

pid_t, uid_t, gid_t, id_t - process/user/group identifier

LIBRARY

Standard C library (libc)

SYNOPSIS

```
#include <sys/types.h>

typedef /* ... */ pid_t;

typedef /* ... */ uid_t;

typedef /* ... */ gid_t;

typedef /* ... */ id_t;
```

DESCRIPTION

pid_t is a type used for storing process IDs, process group IDs, and session IDs. It is a signed integer type.

uid_t is a type used to hold user IDs. It is an integer type.

## Création (duplication) de processus

- **`pid_t fork(void)`**
  La valeur retournée est le *pid* du fils pour le processus père, ou 0 pour le processus fils.
  La valeur -1 est retournée en cas d'erreur.
- Le fils hérite de presque tous les attributs du parent : PID/PPID (modifiées), signaux interceptés (conservés), descripteur de fichiers/tubes ouverts (copiés), pointeurs de fichiers (partagés)

NAME

fork — create a new process

SYNOPSIS

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

The fork() function shall create a new process.

The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below: ...

# Création de processus (appel système **fork()**)

- La valeur de retour de fork est :
  - 0 pour le processus créé (fils).
  - le PID du processus fils pour le processus créateur (père).
  - négative si la création de processus a échoué (manque d'espace mémoire ou le nombre maximal de créations autorisées est atteint).
- Au retour de la fonction fork, l'exécution des processus père et fils se poursuit à partir de l'instruction qui suit fork.
- Le père et le fils ont chacun leur propre image mémoire mais ils partagent certaines ressources telles que les fichiers ouverts par le père avant le fork.

# Création de processus (appel système fork)

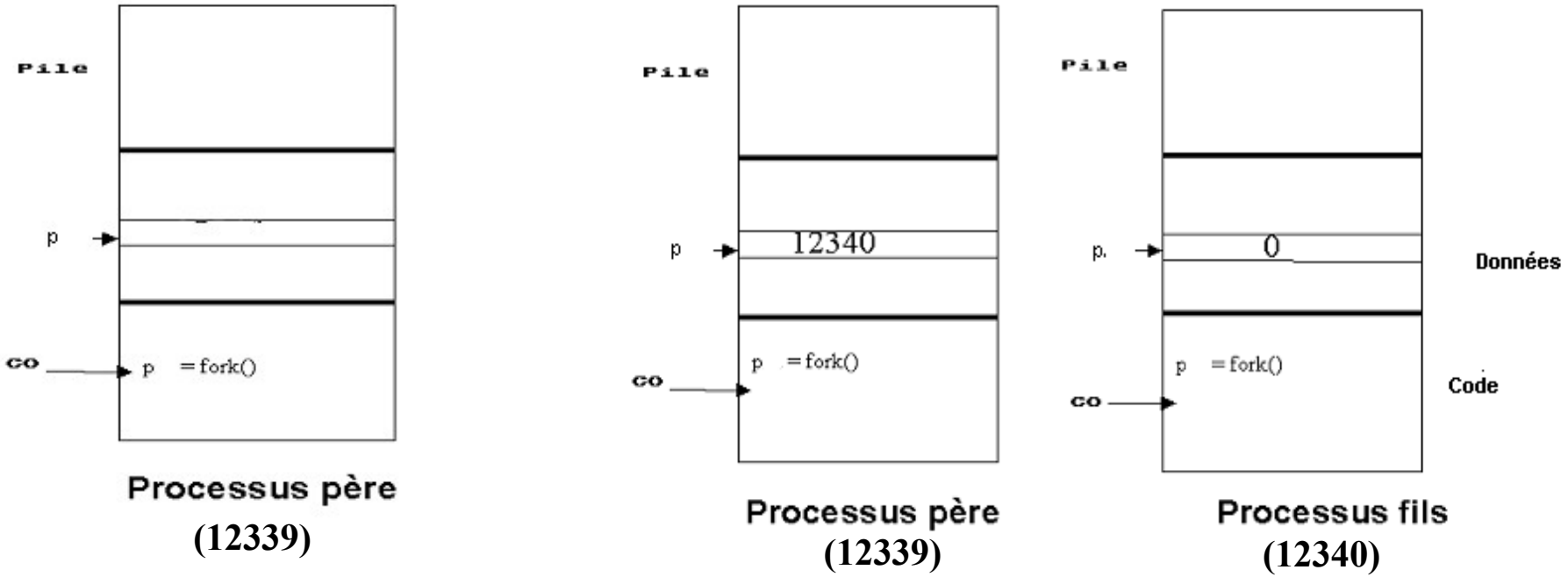
- L'appel système fork :
  - associe un numéro d'identification (le PID du processus);
  - ajoute puis initialise une entrée dans la table des processus (PCB). Certaines entités comme les descripteurs de fichiers ouverts, le répertoire de travail courant, la valeur d'umask, les limites des ressources sont copiées du processus parent;
  - duplique l'espace d'adressage du processus effectuant l'appel à fork (pile+données)

    Principe « **Copy-on-write** ».
  - duplique la table des descripteurs de fichiers mais les descripteurs chez le père et le fils désignent la même entrée dans la table des fichiers (qui est allouée dans la mémoire système) et partagent donc leur position courante: si l'un puis l'autre lit, chacun lira une partie différente du fichier; de même les déplacement effectués par l'un avec lseek sont immédiatement visibles par l'autre.

Après le  fork, le père et le fils vont poursuivre leur exécution à partir de l'instruction qui suit l'appel au fork (voir compteur ordinal) mais chacun a sa propre zone de données.

Le père récupérera dans sa variable p, le PID du fils créé

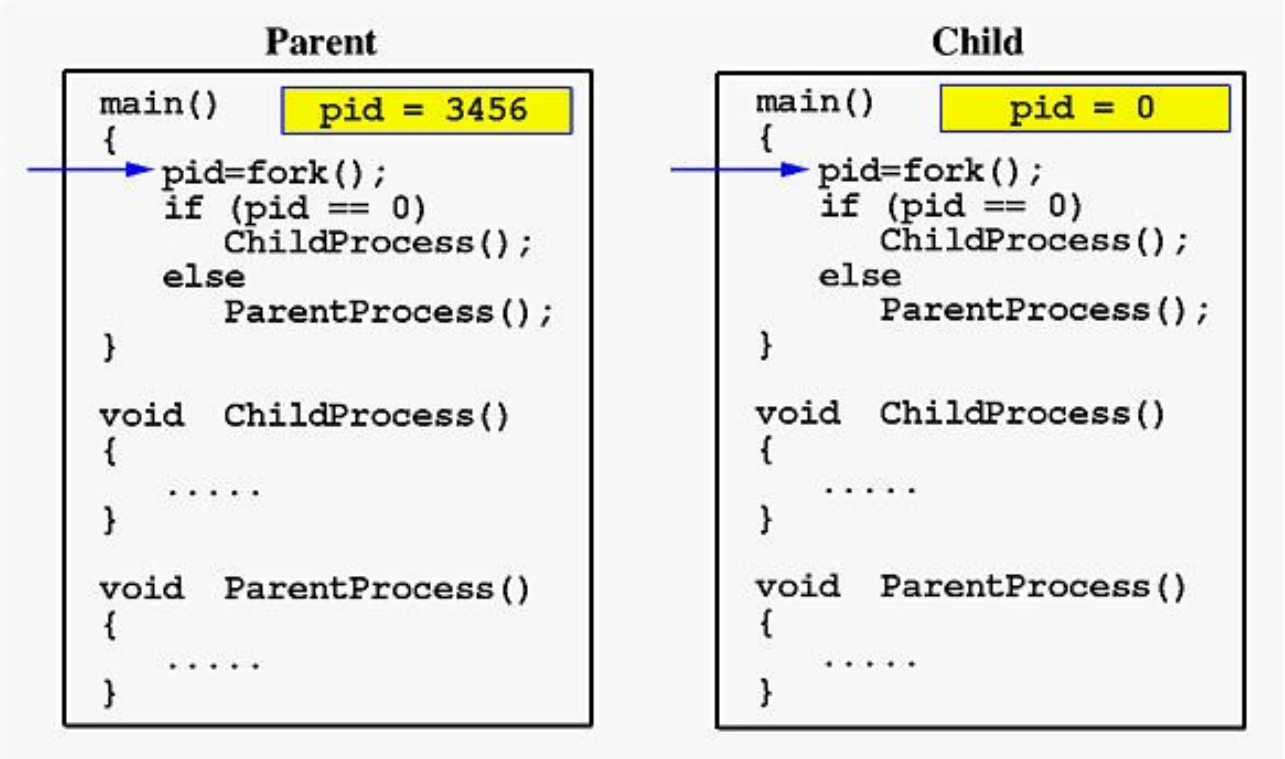
Le fils récupérera dans sa variable p la valeur 0.

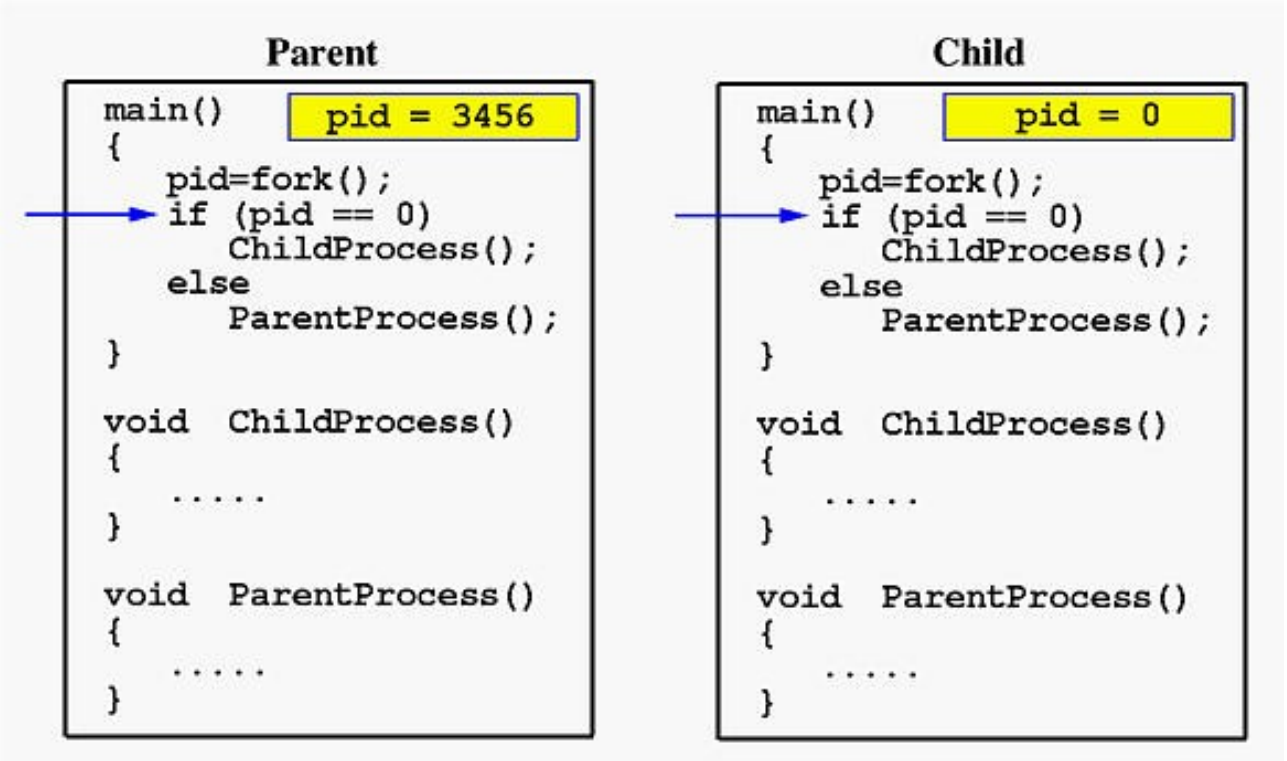


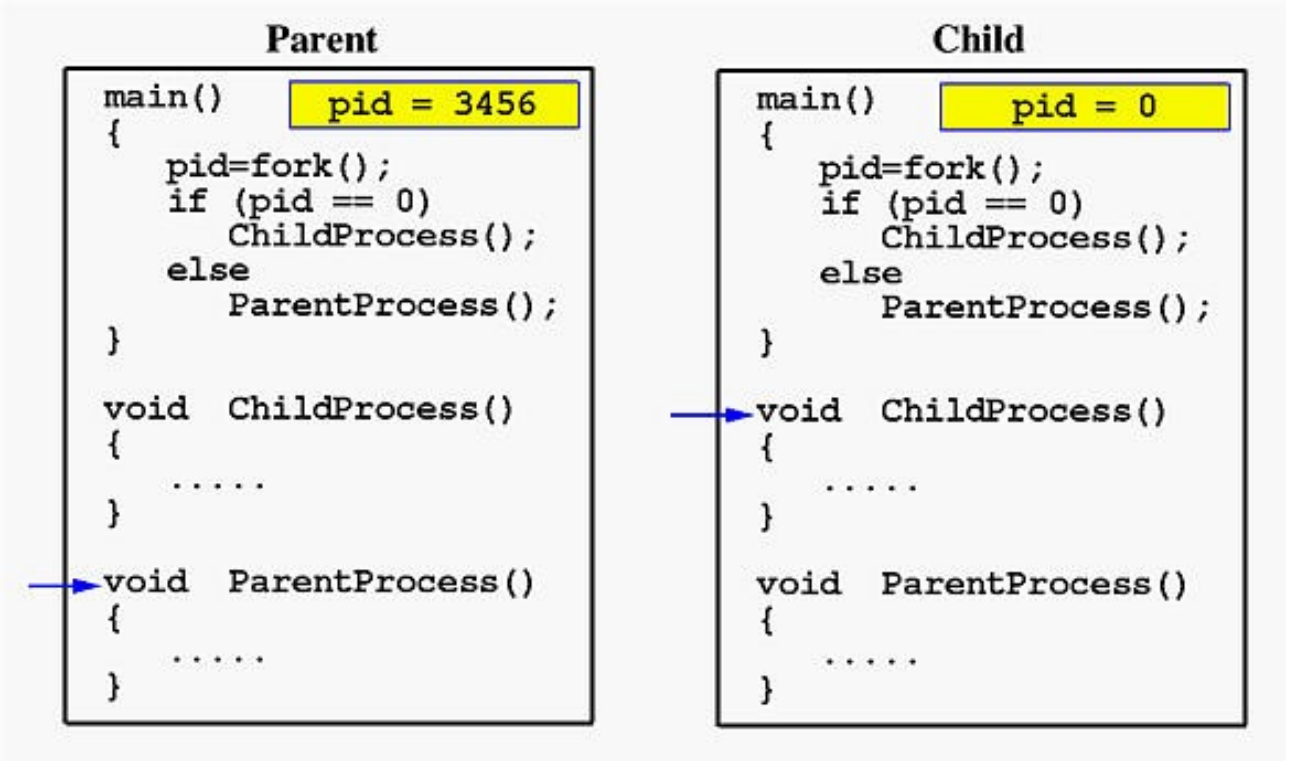
```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```


Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}


Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}

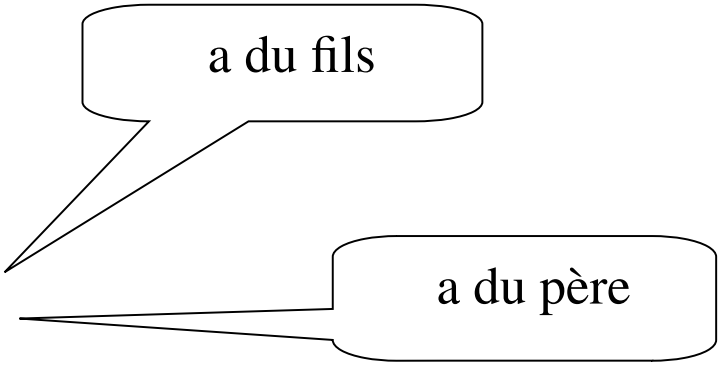

Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}

# Exemple: Création de processus

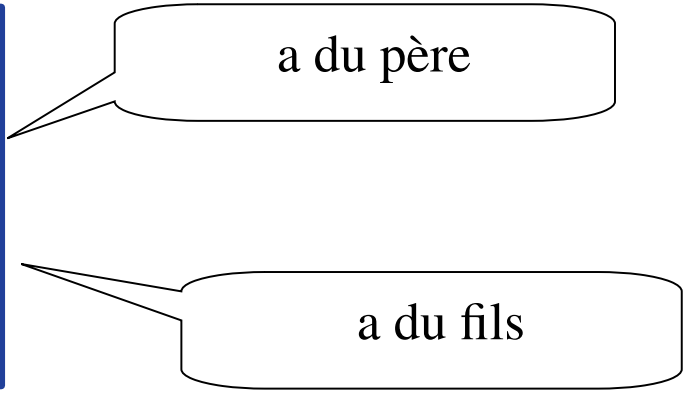
```
// programme tfork.c : appel système fork()
#include <sys/types.h>  /* typedef pid_t */
#include <unistd.h>          /* fork() */
#include <stdio.h>           /* pour perror, printf */
int a=20;
int main(int argc, char *argv) {
   pid_t x;
   // création d'un fils
   switch (x = fork()) {
   case -1: /* le fork a échoué */
      perror("le fork a échoué !");
      break;
   case 0:   /* seul le processus fils exécute  ce « case »*/
      printf("ici processus fils, le PID %d.\n ", getpid());
      a += 10;
      break;
   default:     /* seul le processus père exécute cette instruction*/
      printf("ici processus père, le PID %d.\n", getpid());
      a += 100;
   }
   // les deux processus exécutent ce qui suit
   printf("Fin du Process %d. avec a = %d\n", getpid(), a);
   return 0;
```

# Exemple: Création de processus

```
$ gcc -o tfork tfork.c
$ ./tfork
ici processus père, le PID 12339.
ici processus fils, le PID 12340.
Fin du Process 12340 avec a = 30.
Fin du Process 12339 avec a = 120.
```




```
$ ./tfork
ici processus père, le PID 15301.
Fin du Process 15301 avec a = 120.
ici processus fils, le PID 15302.
Fin du Process 15302 avec a = 30.
$
```



NOM

sleep - Endormir le processus pour un nombre de secondes donné

BIBLIOTHÈQUE

Bibliothèque C standard (libc, -lc)

SYNOPSIS

#include <unistd.h>

unsigned int sleep(unsigned int secondes);

DESCRIPTION

sleep() endort le thread appelant jusqu'à ce qu'un nombre de secondes réelles se soient écoulées ou jusqu'à ce qu'un signal non ignoré soit reçu.

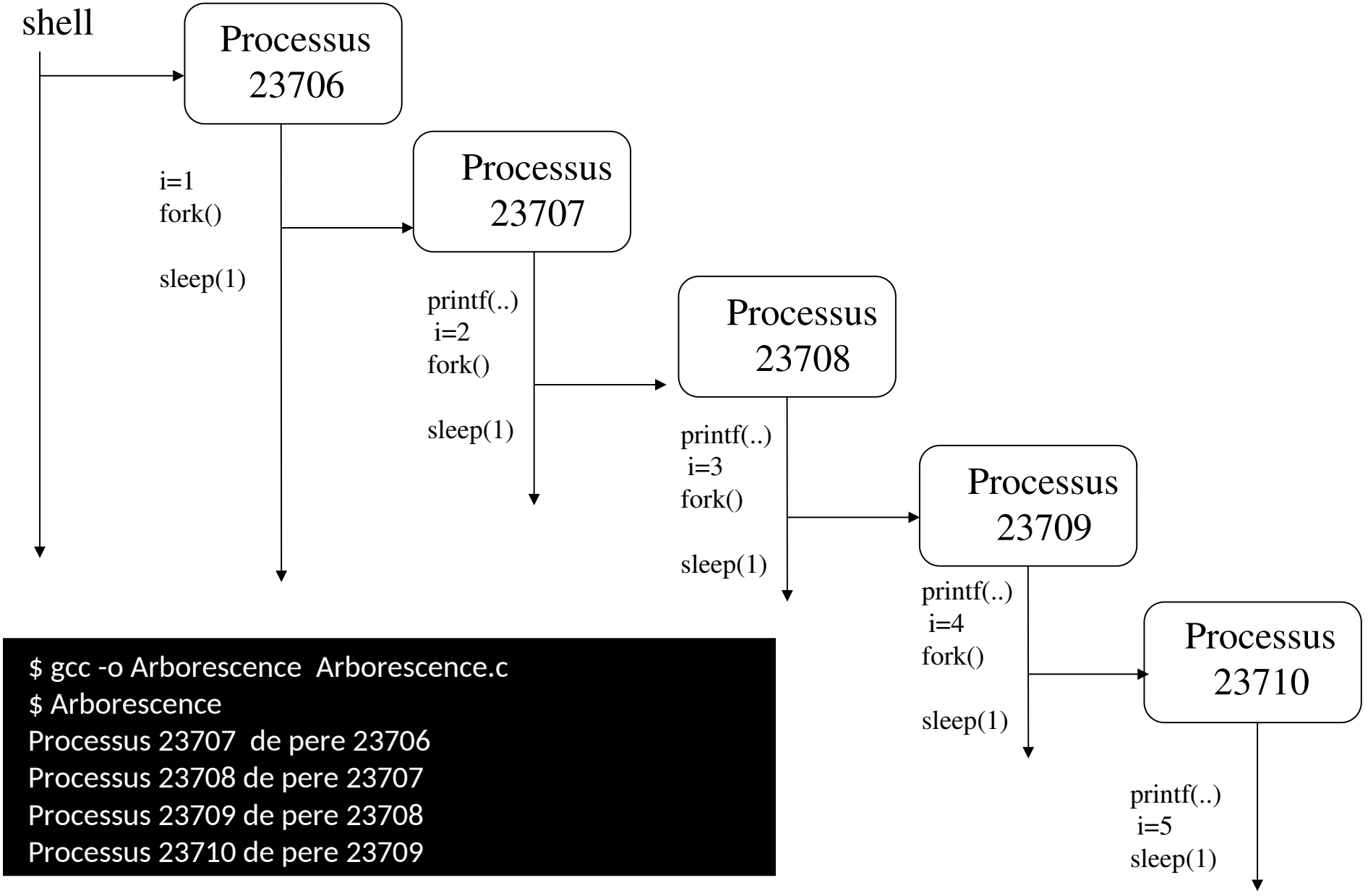
VALEUR RENVOYÉE

sleep() renvoie zéro si le temps prévu s'est écoulé, ou le nombre de secondes restantes si l'appel a été interrompu par un gestionnaire de signal.

# •Exemple: Création de processus imbriqués

```c
// programme arborescence.c : appel système fork()
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv) {
    pid_t p;
    int i, n=5;
    for (i=1; i<n; i++) {
        p = fork();
        if (p > 0) break ;
        printf(" Processus %d  de père %d. \n",
                    getpid(), getppid());
        sleep(2);
    }
    sleep(1);
    return 0;
}
```

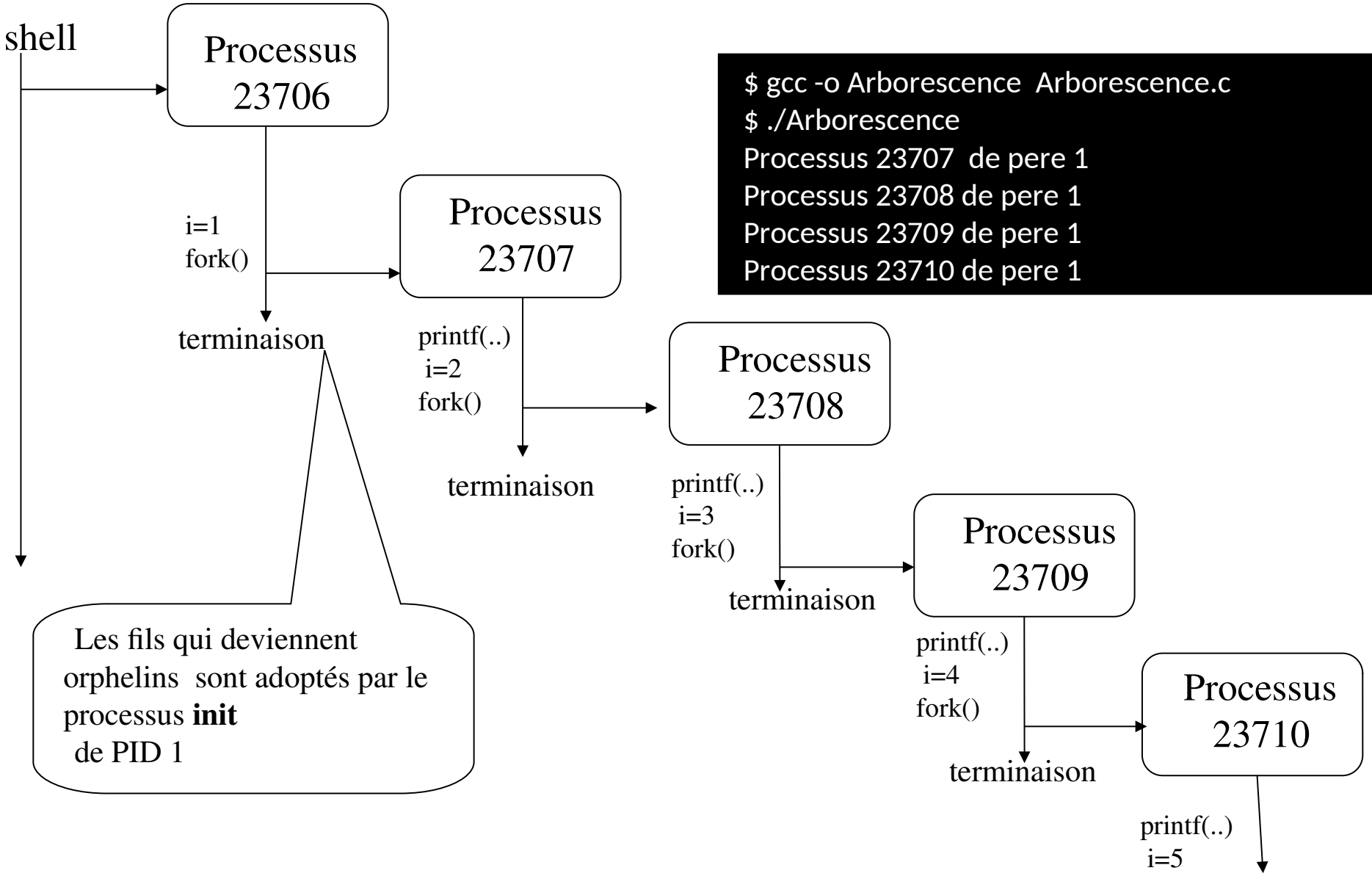
# Exemple: Création de processus imbriqués (2)




```
$ gcc -o Arborescence  Arborescence.c
$ Arborescence
Processus 23707  de pere 23706
Processus 23708 de pere 23707
Processus 23709 de pere 23708
Processus 23710 de pere 23709
```

- **Reprise de « arborescence.c » création de processus imbriqués**

**sans sleep**




```
$ gcc -o Arborescence  Arborescence.c
$ ./Arborescence
Processus 23707  de pere 1
Processus 23708 de pere 1
Processus 23709 de pere 1
Processus 23710 de pere 1
```

# “Rappel”

## Passage de paramètres à un programme C

**int main(int argc, const char *argv[]);**

**int main(int argc, const char *argv[], const char *envp[]);**

- argc: nombre de paramètres sur la ligne de commande (y compris le nom de l’exécutable lui-même)
- argv: tableau de chaînes de caractères contenant les paramètres de la ligne de commande
- envp: tableau de chaînes de caractères contenant les variables d’environnement au moment de l’appel, sous la forme variable=valeur

# Appel système exec sous Unix/Linux

- Le système Linux offre une famille d'appels système **exec** qui permettent à un processus de **remplacer** son code exécutable par un autre processus (programme) spécifié par **path** ou **file**.

```
int execl(const char *path, const char *argv, .....);
int execlp(const char *file, const char *argv, .....);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- Ces appels permettent d'exécuter de nouveaux programmes.

# Appel système exec sous Linux

- Le processus conserve, notamment, son PID, l’espace mémoire alloué, sa table de descripteurs de fichiers et ses liens parentaux (processus fils et père).
- En cas de succès de l’appel système exec, l’exécution de l’ancien code est abandonnée au profit du nouveau.
- En cas d’échec, le processus poursuit l’exécution de son code à partir de l’instruction qui suit l’appel (il n’y a pas eu de remplacement de code).

# exec

Permet le remplacement du contenu actuel du processus par le programme passé en paramètre à la fonction.

• Il n'y a pas de valeur de retour si l'appel réussi car en fait on change de programme.
• Le descripteur du processus et les fichiers ouverts sont les mêmes.
• On utilise cette fonction en général avec fork.

Deux familles :
– execl. . . : nombre de paramètres fixe
= liste d'arguments : connu à la compilation
– execv. . . : nombre de paramètres variable (tableau comme argv)
= peut varier à l'exécution

# Passage de paramètres à un programme

- **Dans le programme à appeler :**

  **int main(int argc, const char *argv[])**

- **Dans votre programme :**
- int execv (const char * path, const char* com[])

  argv ← com

- int execl (const char * path, const char* com0, const char*com1,…, )

  argv [0] ← com0 , argv [1] ← com1, ….,

**Nom**

execl, execlp, execle, execv, execvp - Exécuter un programme.

**Synopsis**

```
#include <unistd.h>
extern char **environ;

int execl (const char *path, const char *arg, ...);
int execlp (const char *file, const char *arg, ...);
int execle (const char *path, const char *arg , ..., char * const envp[]);
int execv (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
```

DESCRIPTION

La famille des fonctions exec() remplace l'image du processus en cours par l'image d'un nouveau processus. Les fonctions décrites dans cette page de manuel sont en réalité des frontaux pour execve(2) (consultez la page de manuel de execve(2) pour plus de détails sur le remplacement de l'image du processus en cours).

Le premier argument de toutes ces fonctions est le nom du fichier à exécuter.

Les fonctions peuvent être regroupées en se basant sur les lettres qui suivent le préfixe "exec".

- Les fonctions se terminant par un "e" transmettent l’environnement dans un tableau envp [] explicitement passé dans les arguments de la fonction, alors que les autres utilisent la variable globale **environ :**

  **extern char **environ;**

- Les fonctions se finissant par un "p" utilisent la variable d’environnement **PATH** (par défaut “/bin:/usr/bin:”) pour rechercher le répertoire dans lequel se situe l’application à lancer, alors que les autres nécessitent un chemin d’accès absolu.

# Exemple: appel système exec

```c
// programme test_exec.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main ()
{
   char* arg[] = {"ps", "f", NULL};

   printf("Bonjour\n");
   execvp("ps", arg);
   printf("Echec de execvp\n");
   printf("Erreur %s\n",strerror(errno));

   return 0;
}
```

# Passage de paramètres à un programme

```c
#include <stdio.h>
#include <stdlib.h>
char *chaine1;
int main(int argc, const char *argv[], const char
*envp[]) {
    int k;

    printf("Paramètres:\n");
    for (k = 0; k < argc; k++)
        printf("%d: %s\n", k, argv[k]);

    printf("Variables d'environnement:\n");
    for (k = 0; envp[k] != NULL; k++)
        printf("%d: %s\n", k, envp[k]);

    return 0;
}
```

# Terminaison de processus

- **void exit (int code_retour)**
- **void _exit(int code_retour)**

  **Différence : _exit ne ferme pas les descripteurs (partage fichiers fils/père)**

NOM

exit - Terminer normalement un processus

BIBLIOTHÈQUE

Bibliothèque C standard (libc, -lc)

SYNOPSIS

#include <stdlib.h>

noreturn void exit(int status);

DESCRIPTION

La fonction exit() termine normalement un processus et l'octet de poids faible de statut (c'est-à-dire statut & 0xFF) est envoyé au processus parent (consultez wait(2))

# Appel système **wait**

- Attendre ou vérifier la terminaison d'un de ses fils :
  - int wait (int * status);  // Attendre après n'importe lequel des fils
  - int waitpid( int pid, int * status, int options);  // attendre pid spécifié

- wait et waitpid retournent :
  - le PID du fils qui vient de se terminer,
  - -1 en cas d'erreur (le processus n'a pas de fils).
  - le paramètre **status** dont les informations peuvent être récupérées au moyen de macros telles que :
    - WIFEXITED(status) : fin normale avec exit
    - WIFSIGNALED(status) : tué par un signal
    - WIFSTOPPED(status) : stoppé temporairement
    - WEXITSTATUS(status) : valeur de retour du processus fils ( **exit(valeur)**)

- waitpid(pid, status, WNOHANG) vérifie seulement la terminaison sans bloquer ; il retourne 0 en cas de non terminaison (pas d'attente).

NAME

wait, waitpid — wait for a child process to stop or terminate

SYNOPSIS

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

The wait() and waitpid() functions shall obtain status information (see Section 2.13, Status Information) pertaining to one of the caller's child processes. The wait() function obtains status information for process termination from any child process.

The waitpid() function obtains status information for process termination, and optionally process stop and/or continue, from a specified subset of the child processes.

# Appel système waitpid

- int waitpid( int pid, int * status, int options);  // attendre pid spécifié

- Options :

  -WNOHANG
  ne pas bloquer si aucun fils ne s'est terminé.
  - WUNTRACED
  recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue...
  Combinables par des OU (|)

- Ex : waitpid(pid, status, WNOHANG) vérifie seulement la terminaison sans bloquer ; il retourne 0 en cas de non terminaison (pas d’attente).

# Exemple: Attente de la fin d'un processus fils

```c
// programme test de execvp : texecvp.c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main (int argc , char * argv[]) {
   if (fork() == 0) { // il s'agit du fils
      // exécute un autre programme
      execvp(argv[1], &argv[1]) ;
      fprintf(stderr, "invalide %s\n ",  argv[1]);
   } else if(wait(NULL) > 0)
      printf("Le père détecte la fin du fils\n");
   return 0;
}
```

```c
// programme fils.c
#include <stdio.h>
#include <unistd.h>
int main() {
  printf("fils [%d]\n", getpid());
  return 0;
}
```

```
$ gcc –-o texecvp texecvp.c
$ ./texecvp date
mercredi,  6 septembre 2022,
16:12:49 EDT
Le père détecte la fin du fils

$ ./texecvp ugg+kjù
invalide ugg+kjù
Le père détecte la fin du fils

$ gcc -o fils fils.c
$ ./fils
 fils [18097]
$ ./texecvp fils
fils [18099]
 Le père détecte la fin du fils
```

# Exemple: Attente de la fin d'un processus fils

```c
// programme parent.c
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  int p, child, status;
  p = fork();
  if (p == -1)
    return -1;
  if(p > 0) { /* parent */
    printf ("père[%d], fils[%d]\n", getpid(), p);

  if ((child=wait(&status)) > 0)
    printf("père[%d], Fin du fils[%d]\n", getpid(), child);
    printf("Le père[%d] se termine \n", getpid());
  } else { /* enfant */
    if ((status=execl("/home/user/a.out", "a.out", NULL)) == -1)
        printf("le programme n'existe pas : %d\n",status);
     else
        printf("cette instruction n'est jamais exécutée\n");
  }

  return 0;
}
```

Le père attend la fin du fils

Le fils change de code exécutable

```c
// programme fils.c
#include <stdio.h>
#include <unistd.h>
int main() {
  printf("fils [%d]\n", getpid());
  return 0;
}
```

```
$ gcc fils.c
$ gcc -o parent parent.c
$./parent
  père[10524],  fils[10525]
  fils [10525]
  père[10524] Fin du fils[10525]
  Le père[10524] se termine
```

# Exemple: Terminaison de processus

```c
// programme deuxfils.c
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

void fils(int i);
int main() {
   int status;
   if (fork()) { // création du premier fils
      if (fork() == 0) // création du second fils
         fils(2);
   } else
   fils(1);

   if (wait(&status))
       if (WIFEXITED(status))
          printf("fin du fils%d\n", WEXITSTATUS(status));

   if (wait(&status))
       if (WIFEXITED(status))
          printf("fin du fils%d\n", WEXITSTATUS(status));

   return 0;
}

void fils(int i) {
   sleep(2);
   exit(i);
}
```

```
$ gcc -o deuxfils deuxfils.c
$ ./deuxfils
fin du fils1
fin du fils2
$ ./deuxfils
fin du fils2
fin du fils1
```

L'ordre d'exécution n'est pas garanti (sauf si on utilise des outils dédiés!)

# Terminaison de processus

- Un processus se termine par une demande d'arrêt volontaire (appel système **exit**) ou par un arrêt forcé provoqué par un autre processus (appel système **kill**) ou une erreur.

- Lorsqu'un processus fils se termine :
  - son état de terminaison est enregistré dans son PCB,
  - la plupart des autres ressources allouées au processus sont libérées.
  - le processus passe à l'état zombie (<defunct>).

# Terminaison de processus

- Son PCB et son PID sont conservés jusqu'à ce que son processus père ait récupéré cet état de terminaison.

- Les appels système wait(status) et waitpid(pid, status, option) permettent au processus père de récupérer, dans le paramètre status, cet état de terminaison.

- Que se passe-t-il si le père meurt avant de récupérer ces informations ?

  - Classiquement, les processus fils sont alors automatiquement rattachés au processus n°1, init, qui se charge à la place du père original d'appeler wait sur ces derniers.

Remarque : aujourd'hui, des processus dits « subreaper » sont chargés de récupérer les états de terminaison à la place du processus init.

# Processus Zombie

- Un processus fils se termine
- Mais le père ne récupère pas le code de retour par wait, ou waitchild.

- Impossible à tuer (même par root)

- Un processus zombie n'utilise que son entrée dans la table des processus (car il a un PID)

- Pour tuer un processus zombie il faut tuer son père !
  - Les processus fils sont alors automatiquement rattachés au processus n°1 init (ou à un subreaper) , qui se charge
  - à la place du père original d'appeler wait sur ces derniers.

# Gestion des erreurs systèmes

En programmation système en C, pas de try/catch ou équivalent, tester la réussite d'un appel système consiste à regarder la valeur retournée.
De façon générale, la convention est de retourner -1 ou NULL (si un pointeur est attendu) en cas d'erreur. Toutefois, on peut obtenir un code d'erreur et un message en utilisant errno et les fonctions associées.

La variable **errno** contient le code de la dernière erreur déclenchée par un appel de fonction système :

**int errno ;**

La liste globale d'erreurs (sous forme de chaînes de caractères) est contenue dans **sys_errlist[]** qui peut être indexée par errno pour obtenir le message d'erreur associé.

La fonction **perror** permet d'afficher en clair sur la sortie d'erreur standard le message d'erreur associé à errno :

**void perror( const char * s);**

Il est important d'afficher ces messages juste après l'appel système :
Il n'y a pas de remise à zéro automatique de errno.

# Gestion des erreurs systèmes

NOM

errno - Code de la dernière erreur

BIBLIOTHÈQUE

Bibliothèque C standard (libc, -lc)

SYNOPSIS

#include <errno.h>

DESCRIPTION

Le fichier d'en-tête <errno.h> définit la variable de type entier errno qui est renseignée par les appels système et quelques fonctions de bibliothèque pour décrire les conditions de la survenue d'une erreur.

errno

La valeur de errno n'est significative que lorsque la valeur de retour de l'appel système indique une erreur (c'est-à-dire -1 pour la plupart des appels système ; -1 ou NULL pour la plupart des fonctions de bibliothèque) ; une fonction qui réussi est autorisée à modifier errno. La valeur de errno n'est jamais mis à zéro par un appel système ou une fonction de bibliothèque.

NOM

perror - Afficher un message d'erreur système

SYNOPSIS

#include <stdio.h>

void perror(const char *s);

...

DESCRIPTION

La fonction perror() produit un message sur la sortie d'erreur standard décrivant la dernière erreur rencontrée lors d'un appel à une fonction système ou de bibliothèque.

Premièrement, la chaîne s en argument est imprimée (si s n'est pas NULL et *s n'est pas l'octet NULL (« e0 »), suivie d'une virgule et d'espaces, puis un message d'erreur correspondant à la valeur courante de errno et un saut de ligne.

Pour être la plus utile possible, la chaîne en argument doit inclure le nom de la fonction dans laquelle l'erreur est survenue.

# Exemple de gestion d'erreur

```
if ((fd = open(nom_fichier, mode)) == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

En cas d'erreur (le fichier n'existe pas), ce code affiche :

open: Aucun fichier ou répertoire de ce type