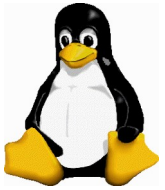
Programmation système

Laurent Mascarilla

2019-2020

Les systèmes d'exploitation



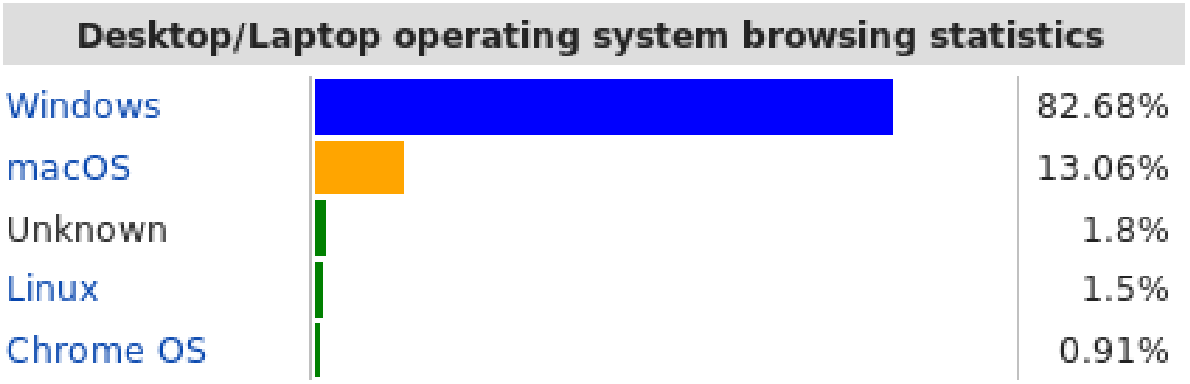
GNU/Linux

Mac OS X

Usage share of operating systems(wikipedia)

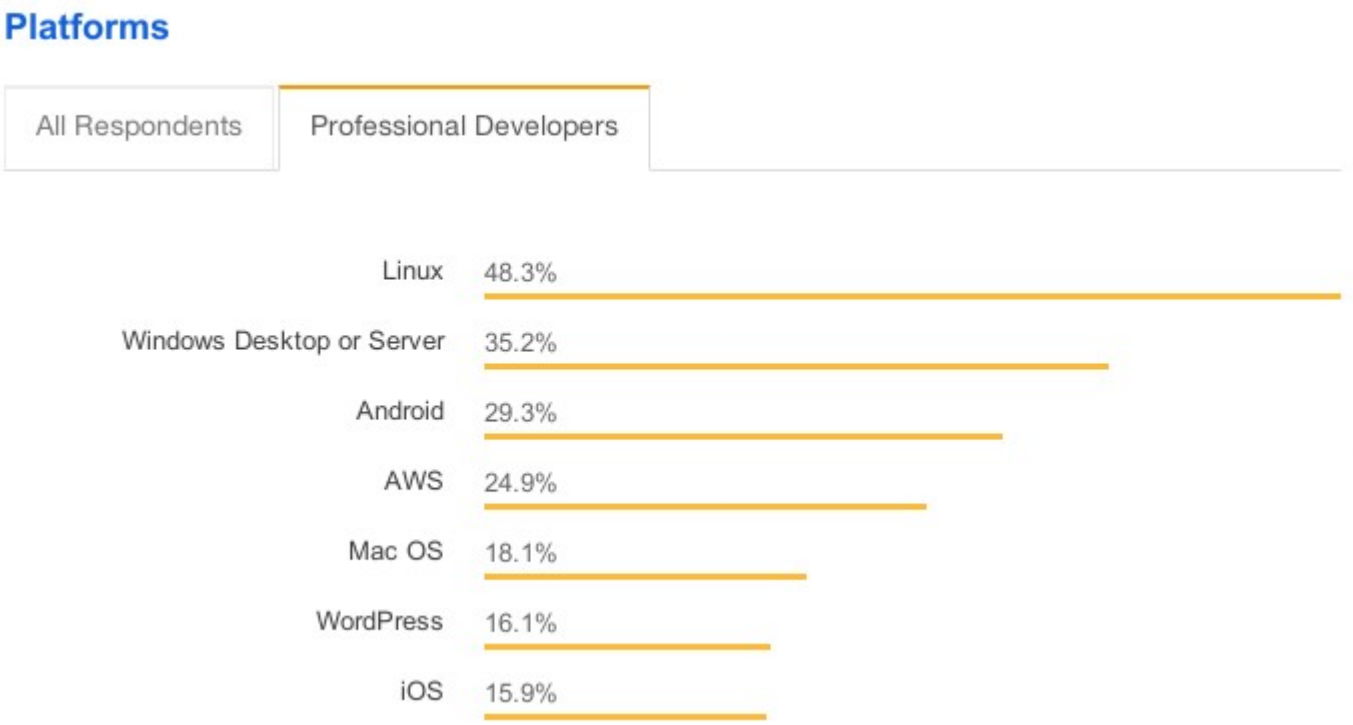
Desktop :

Windows est très majoritaire



Mais côté developpeurs...

- Stackoverflow 2018



Serveurs (web ,mail ,dns...)

Source	Date	Unix, Unix-like				Microsoft Windows	References
		All	Linux	FreeBSD	Unknown		
W3Techs	Feb 2015	67.8%	35.9%	0.95%	30.9%	32.3%	[77][78]
Security Space	Feb 2014	<79.3%	N/A			>20.7%	[79][80]
W3Cook	May 2015	98.3%	96.6%	1.7%	0%	1.7%	[81]

Note

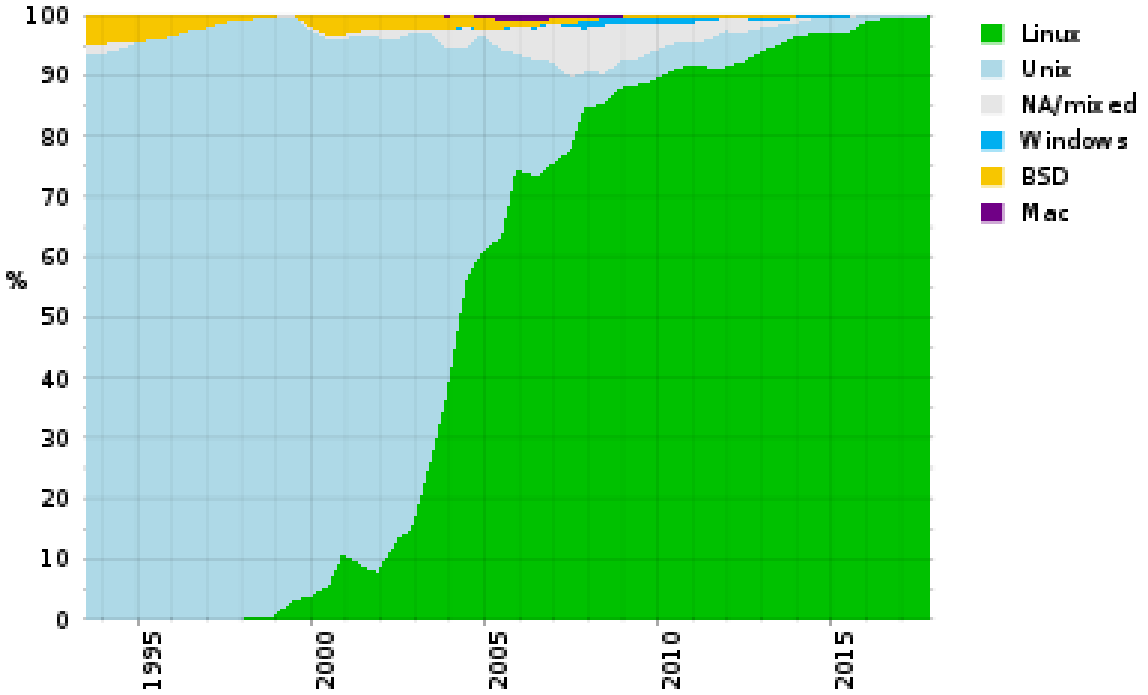
W3Techs checked the top **ten** million web servers daily from June 2013, but W3Techs's definition of "website" differs a bit from Alexa's definition; the "top 10 million" websites are actually fewer than 10 million. W3Techs claims that these difference "have no statistical significance".[82]

Note

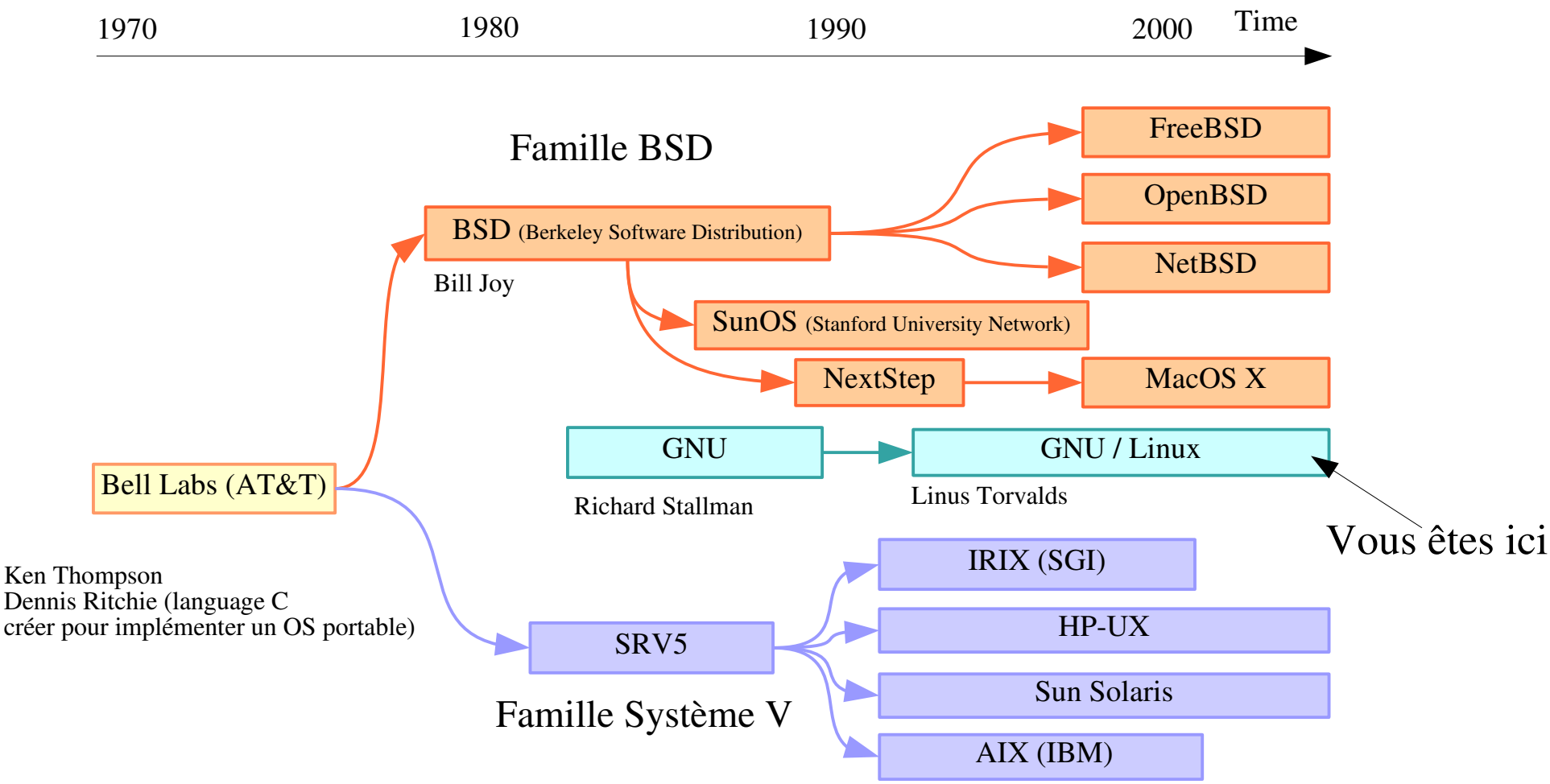
W3Cook checks the top **one** million web servers monthly, taken from the Alexa ranking, using HTTP headers, DNS records, and WHOIS data, among other sources.[83]

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



« 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 le logiciel d’un ordinateur est organisé en un certain nombre de processus (processus système, processus utilisateur).

- Un processus (un seul fil) est un programme en cours d’exécution. On dit qu'il s'agit d'une instance de programme

- Un processus est caractérisé par:
  - 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 ressources allouées (fichiers ouverts, mémoires, périphériques …);
  - 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….

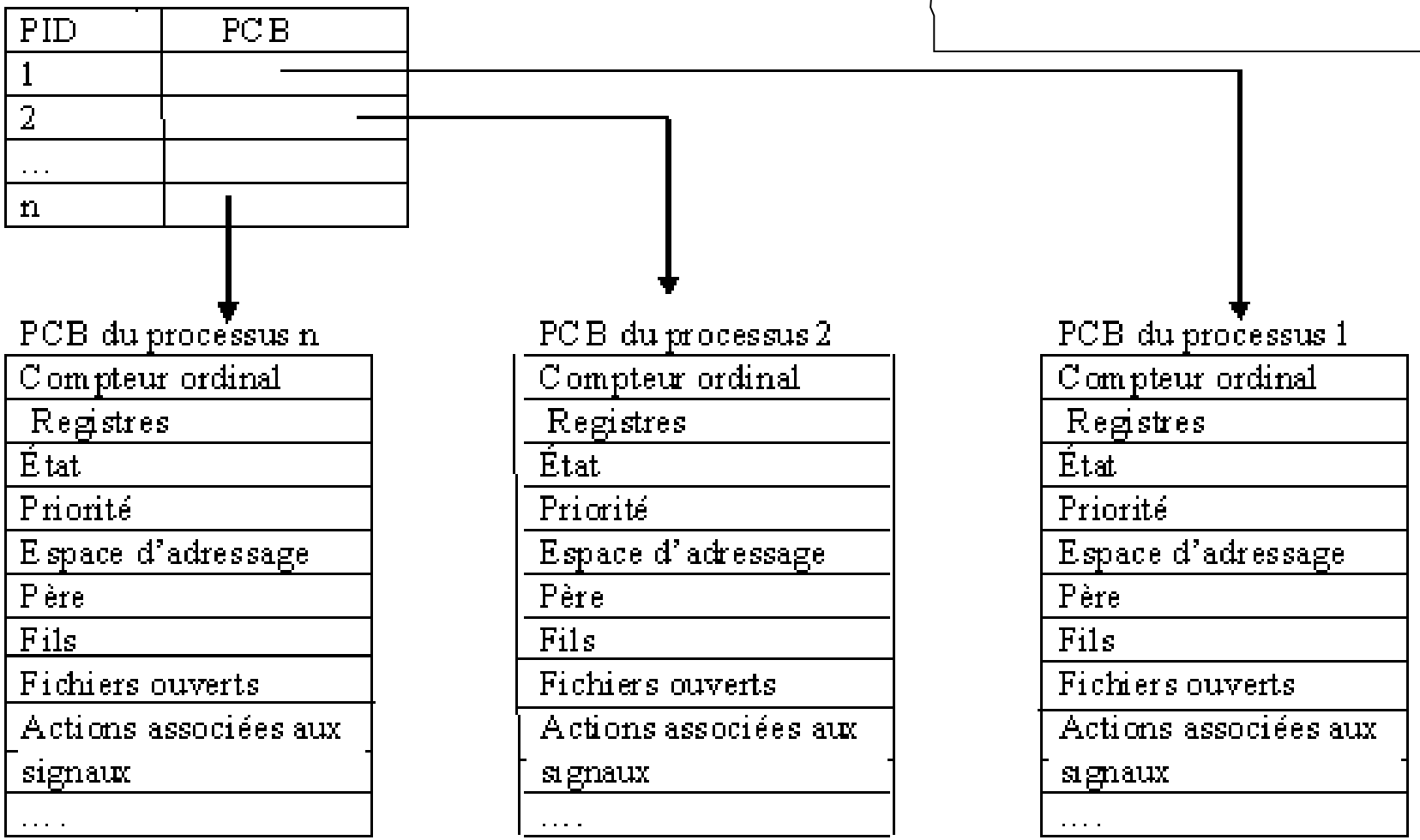
Exemple : commande ps

- UID nom de l'utilisateur qui a lancé le process
- PID correspond au numéro du process
- PPID correspond au numéro du process parent
- C au facteur de priorité : plus la valeur est grande, plus le processus est prioritaire
- 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.

- Le système d’exploitation maintient dans une table appelée «table des processus» les informations sur tous les processus créés (une entrée par processus : Bloc de Contrôle de Processus PCB).

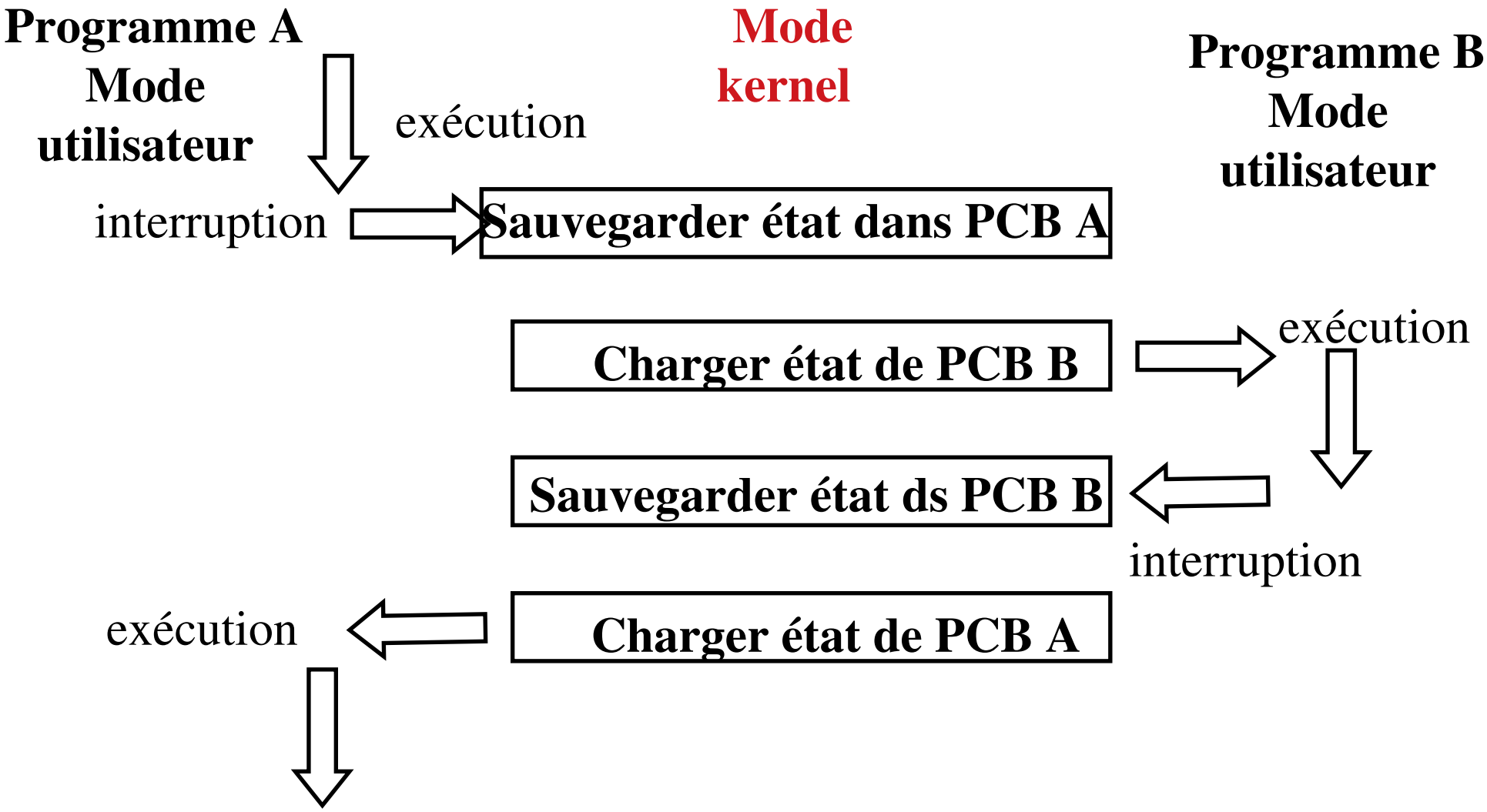
Table des processus

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).

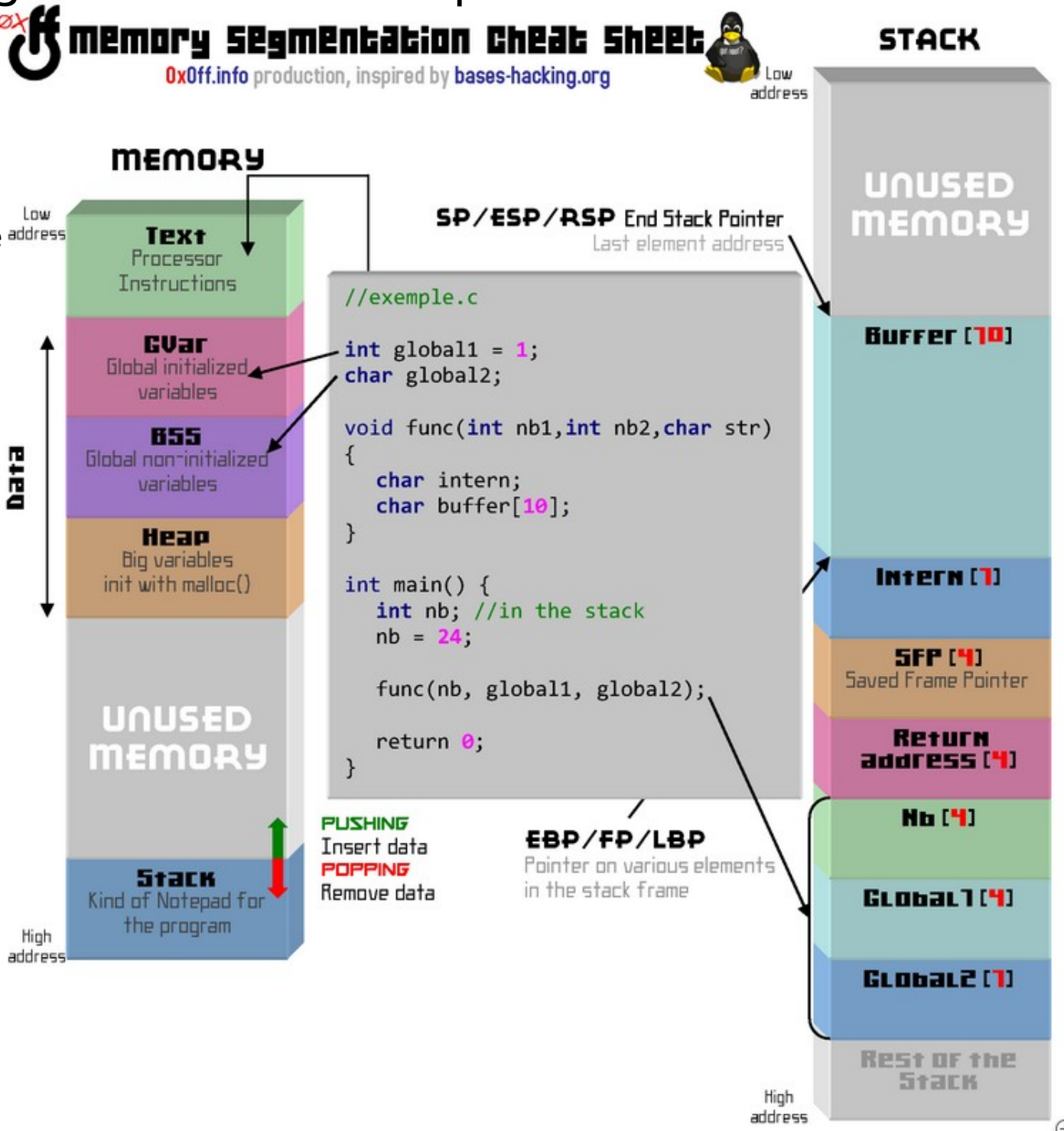
Exemple: changement de contexte
(décidé par l'ordonnanceur (cf. plus tard...))



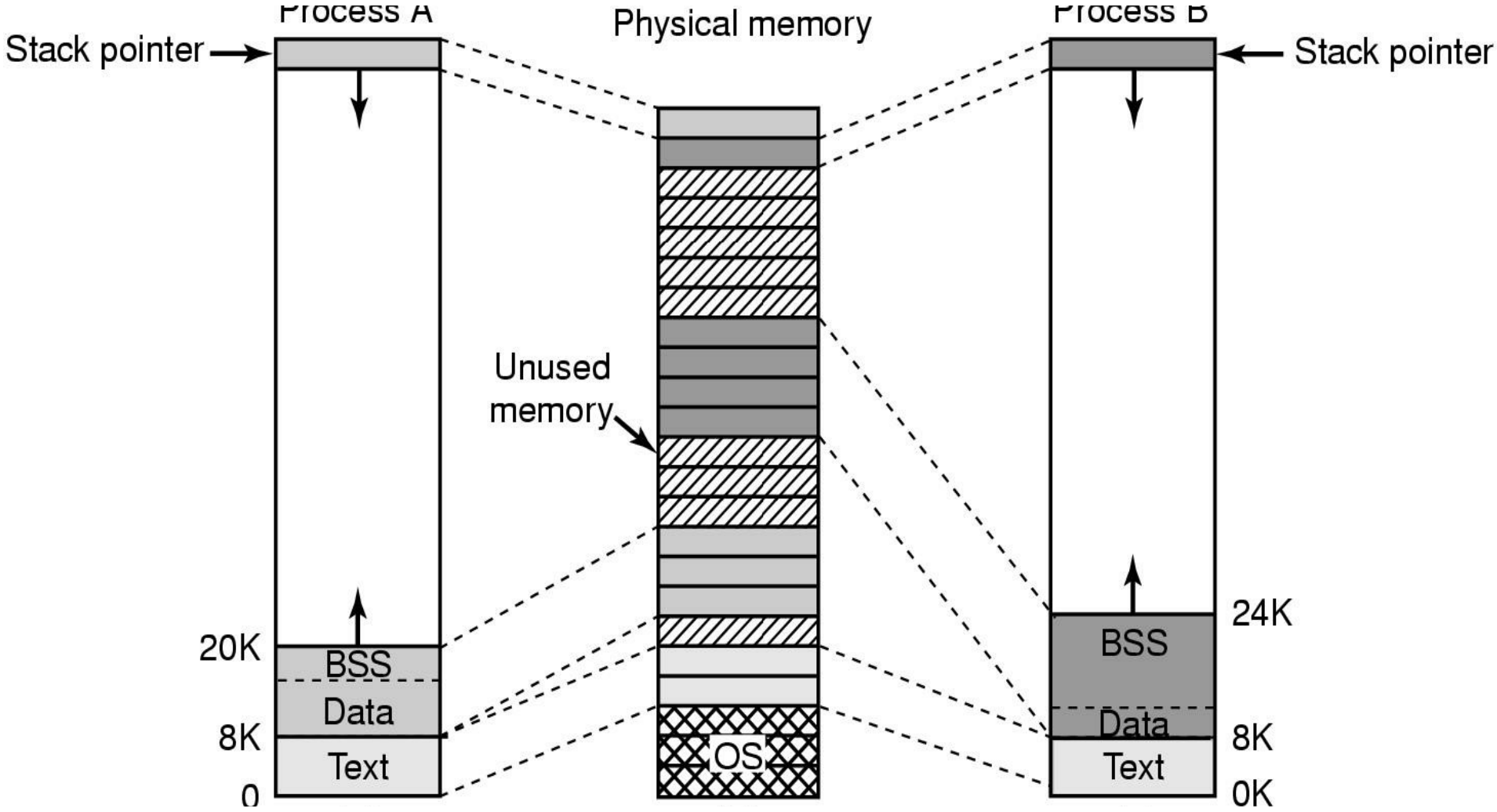
La mémoire d'un programme informatique est divisée ainsi :

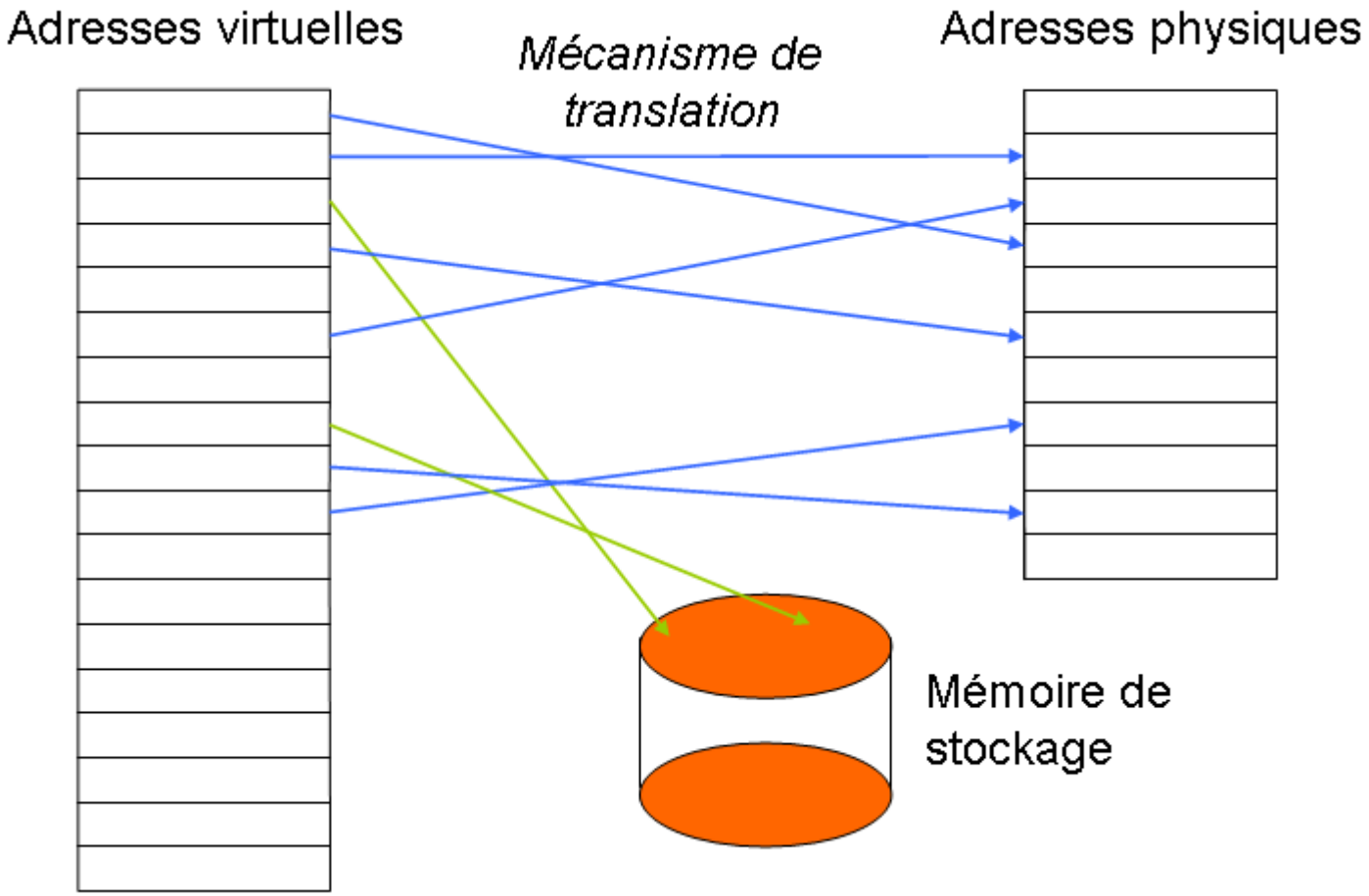
- segment de données (données + BSS + tas binaire)
- Pile d'exécution, souvent abrégée par "la pile"
- segment de code (aka Text)

BSS : Block Started by Symbol



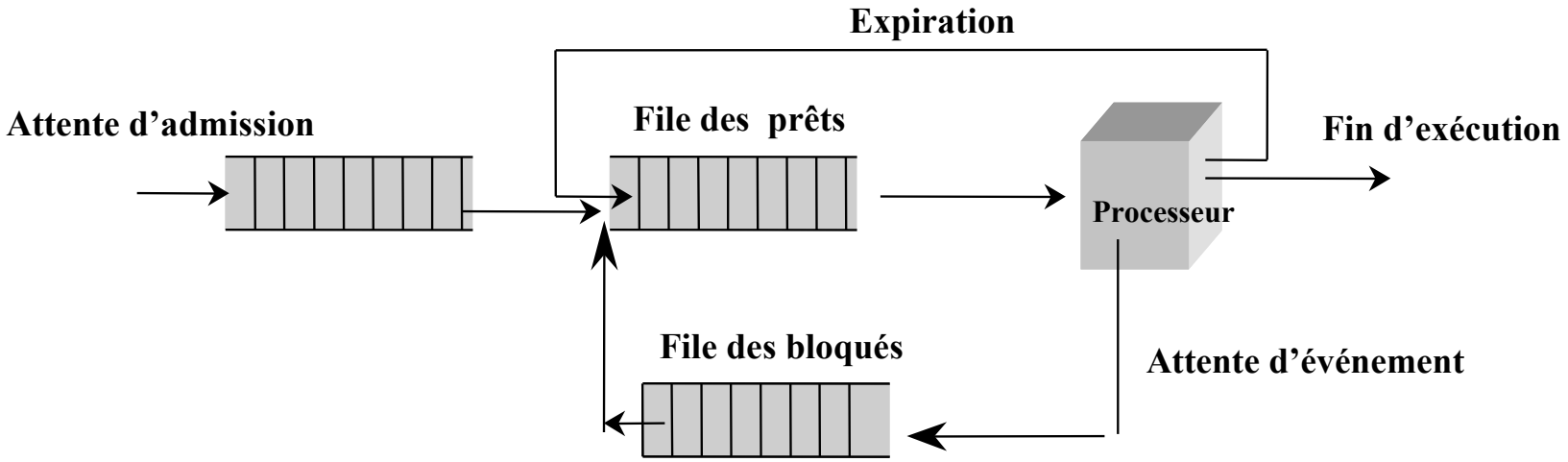
L’espace d’adressage virtuel

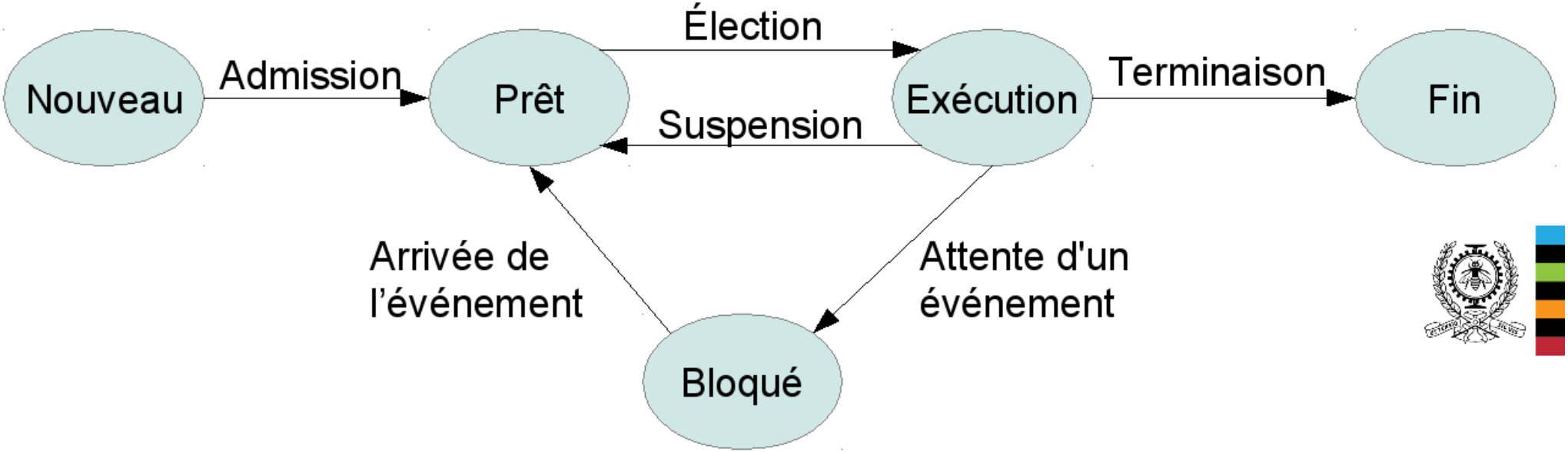




Mémoires virtuelles et mémoire physique

États d'un processus





Les 5 états des processus

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 d'un autre processus)

T (TASK_STOPPED) : Stoppé

Z (TASK_ZOMBIE) : Zombie

D : Mort (on ne le voit jamais)

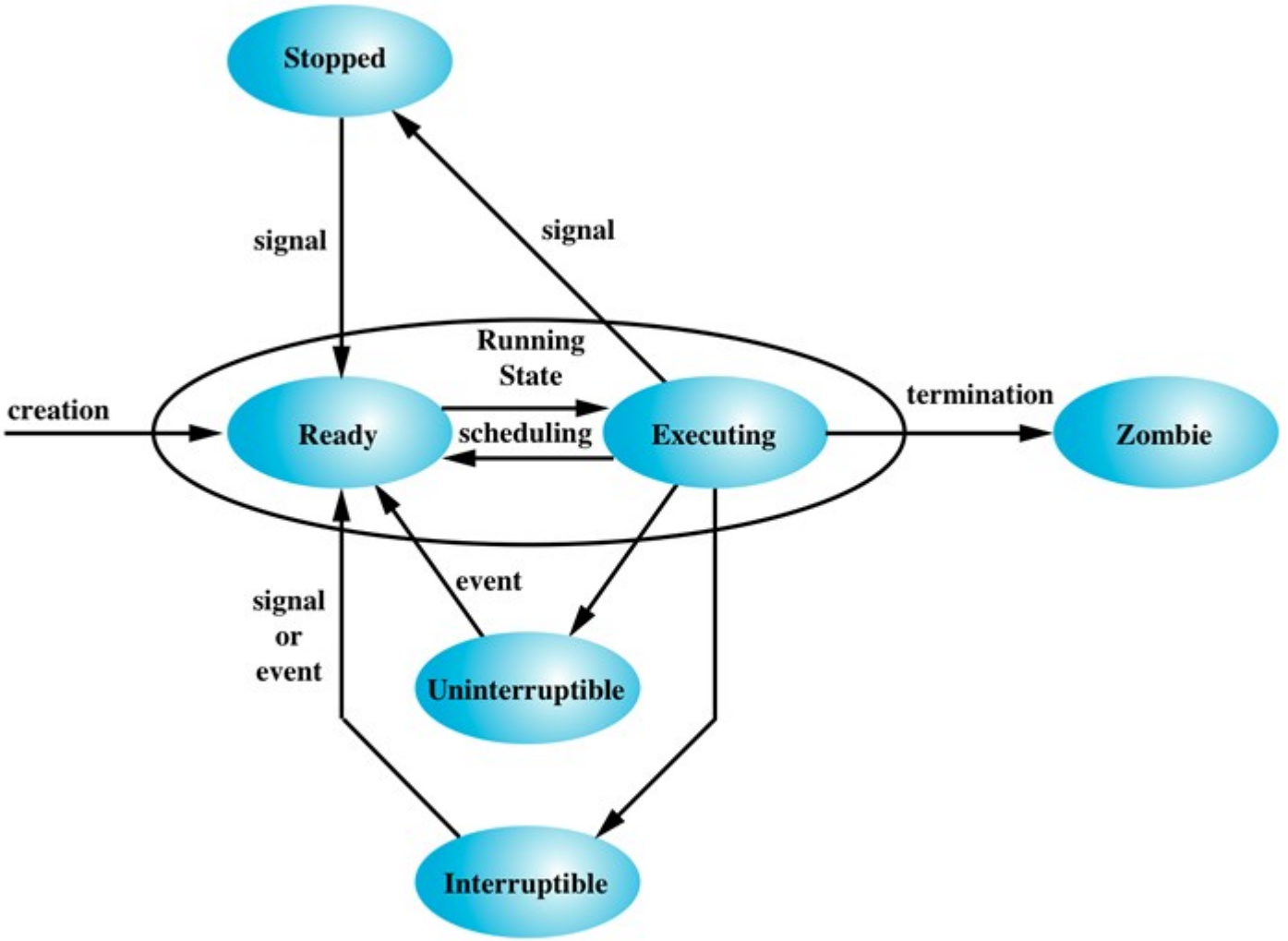


Figure 4.18 Linux Process/Thread Model

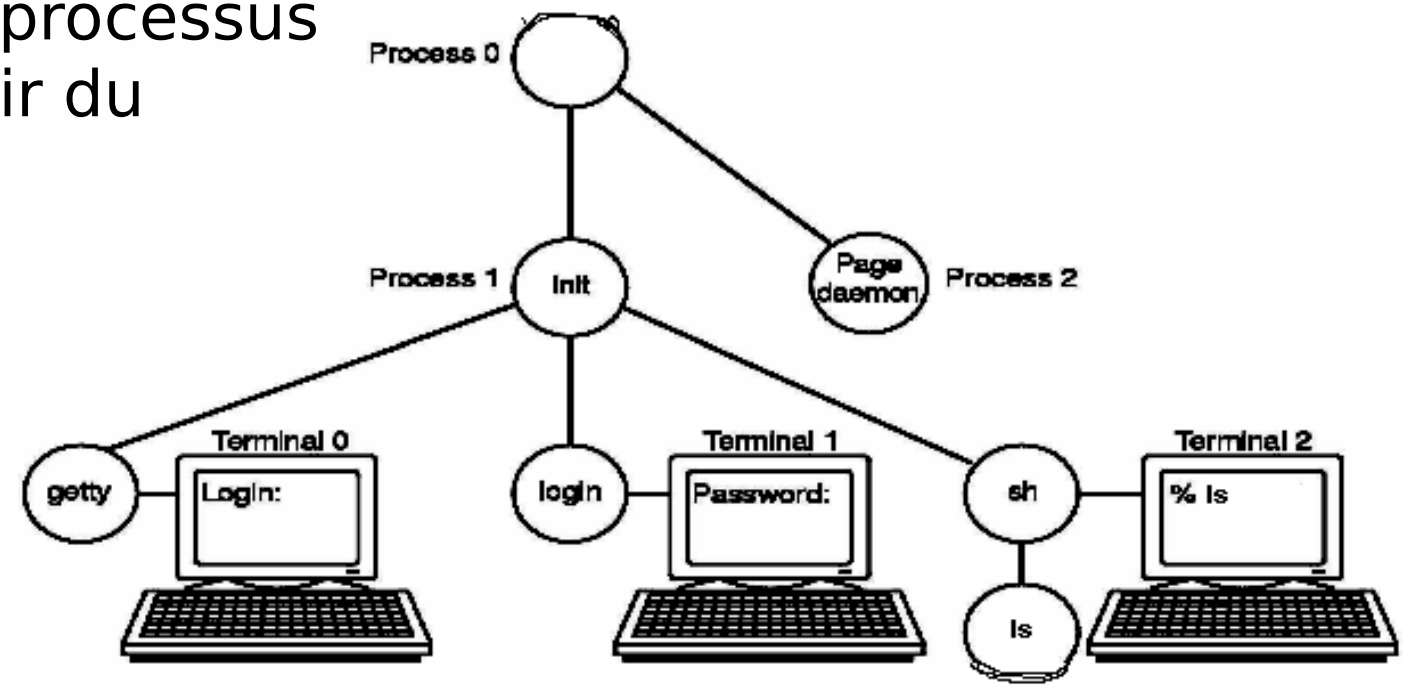
Hiérarchie des processus

- Le système d’exploitation fournit un ensemble d’appels système qui permettent la création, la destruction, la communication et la synchronisation des processus.
- Les 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 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 **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). L'appel system **getpid**() permet de récupérer le PID du processus : **int getpid()**;
- Chaque processus a un père, à l'exception du premier processus créé (structure arborescente). S'il perd son père (se termine), il est tout de suite adopté par le processus **init** de PID 1*. L'appel système **getppid**() permet de récupérer le PID de son processus père.
- Un processus fils peut partager certaines ressources (**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 !
- 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**).

- Un processus est un programme en cours d'exécution
- Un système multi-utilisateurs peut exécuter le même programme sous plusieurs processus, pour plusieurs utilisateurs
- 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**)
  - du Descripteur du processus (informations relatives au processus, pile d'exécution,...)

Types d'informations : `<unistd.h>`	**Primitives**
`pid_t PID` : numéro de processus, **`pid_t PPID`**: numéro de processus parent, **`uid_t UID`** : numéro de l'utilisateur réel, etc.	**`pid_t getpid(void)`** **`pid_t getppid(void)`** **`uid_t getuid(void)`**

•Création ou 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)

•Terminaison de processus

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

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

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.

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).

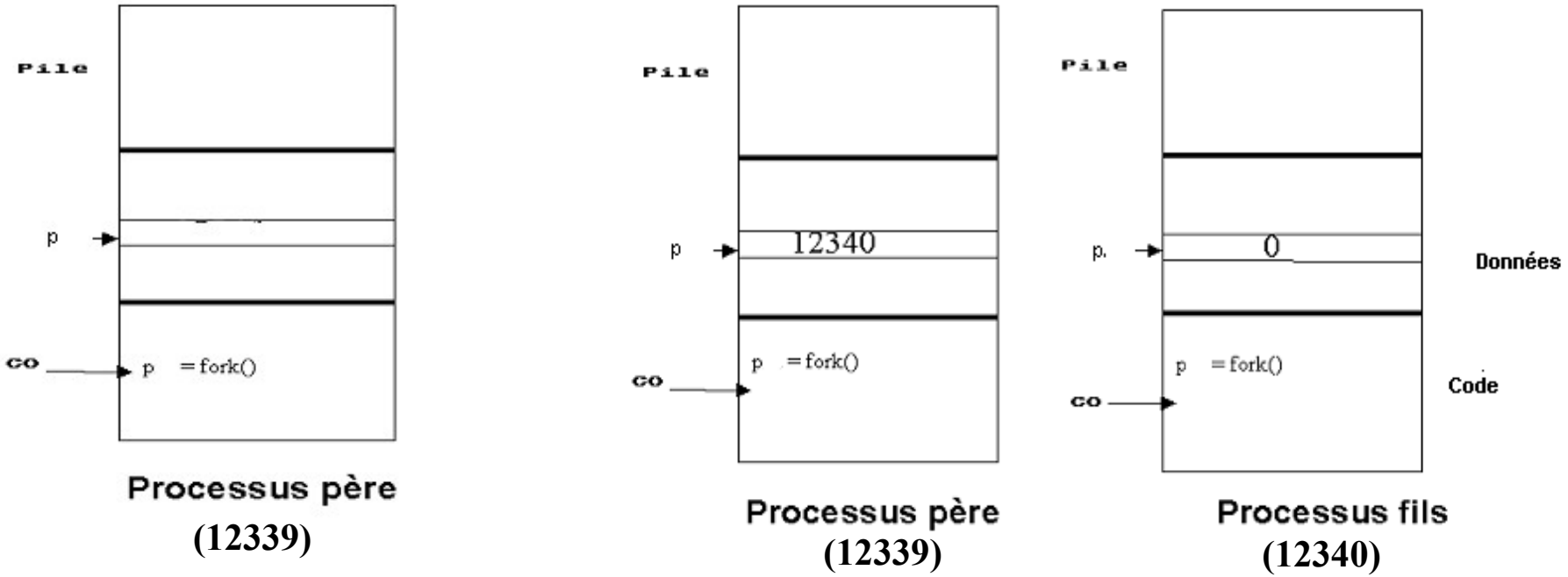
- Au retour de la fonction fork, l'exécution des processus père et fils se poursuit, en temps partagé, à partir de l'instruction qui suit fork.

- 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.

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

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

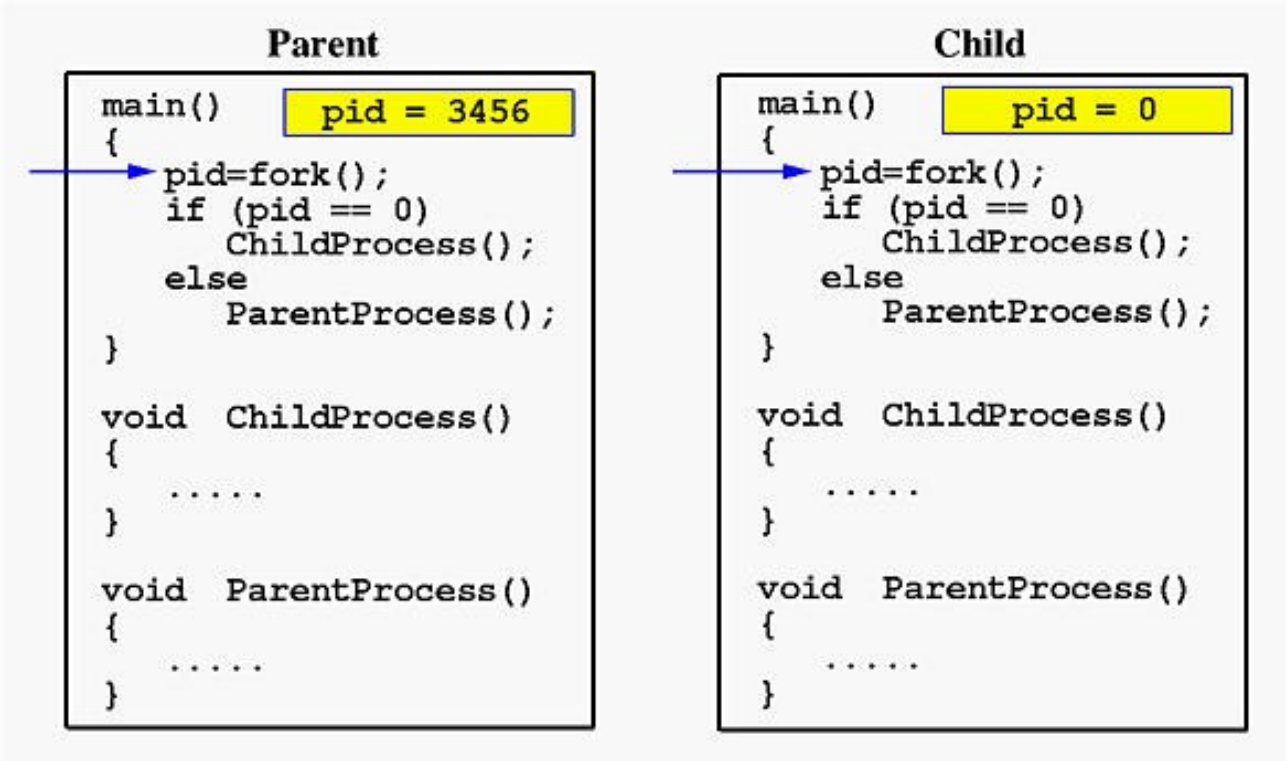
Le fils récupérera dans sa variable p la valeur 0.

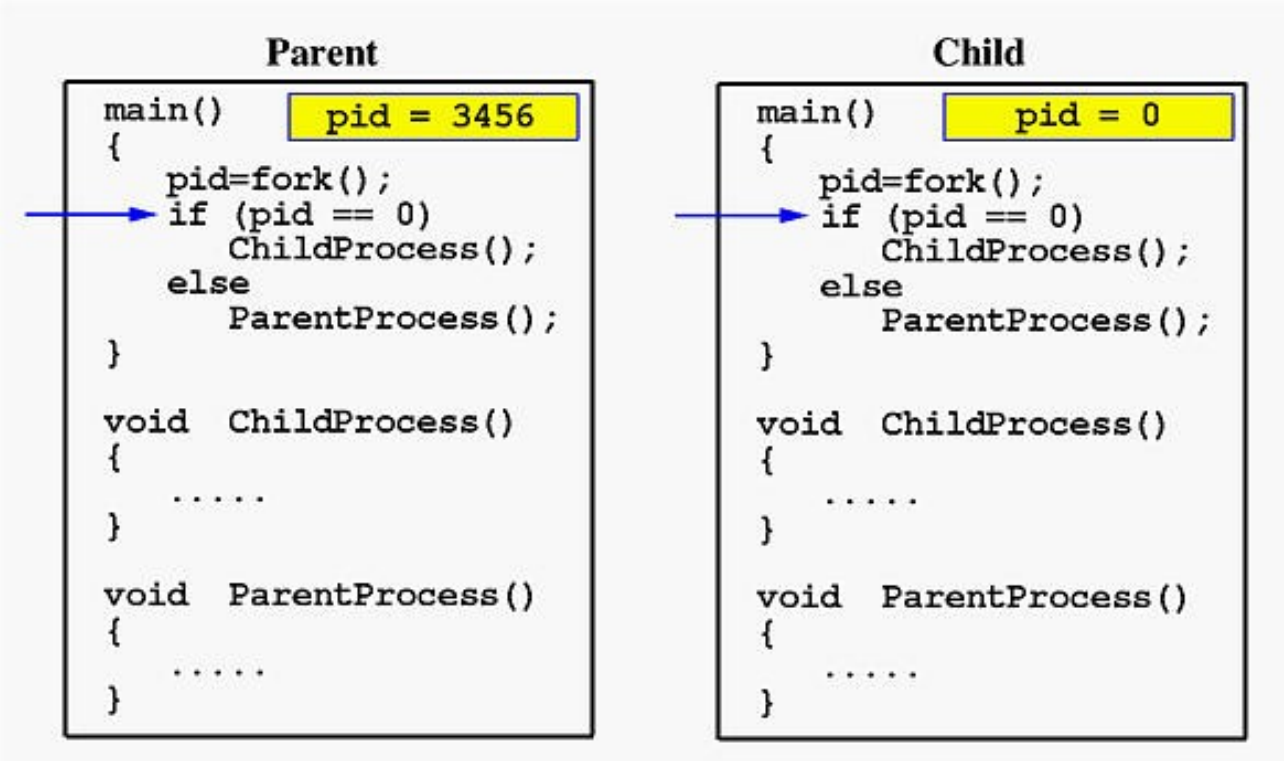


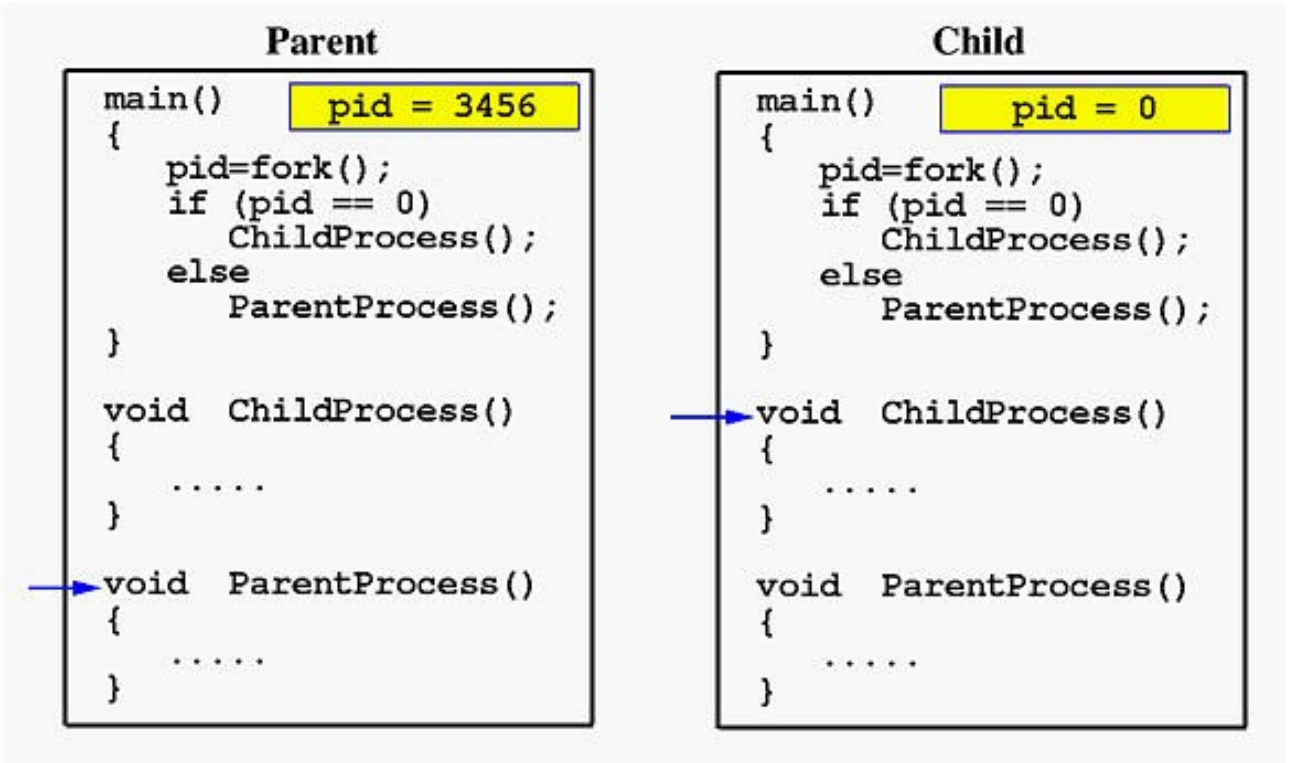
```
main()
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

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()
{
.....
}

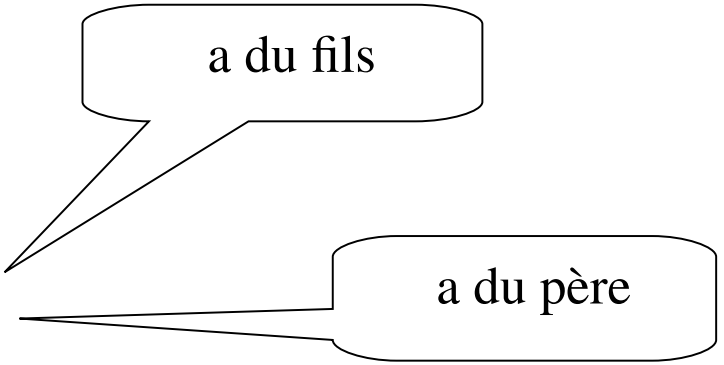

Parent
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
Child
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....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

Exemple: Création de processus

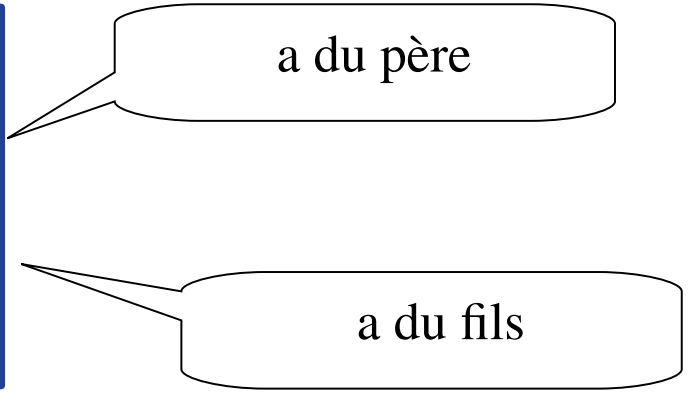
```c
// programme tfork.c : appel système fork()
#include <sys/types.h>  /* typedef pid_t */
#include <unistd.h>    /* fork() */
#include <stdio.h>     /* pour perror, printf */
int a=20;
int main(int argc, char *argv) {
    pid_t x;
    // création d'un fils
    switch (x = fork()) {
    case -1: /* 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 (2)

```
jupiter% gcc -o tfork tfork.c
jupiter% 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.
```



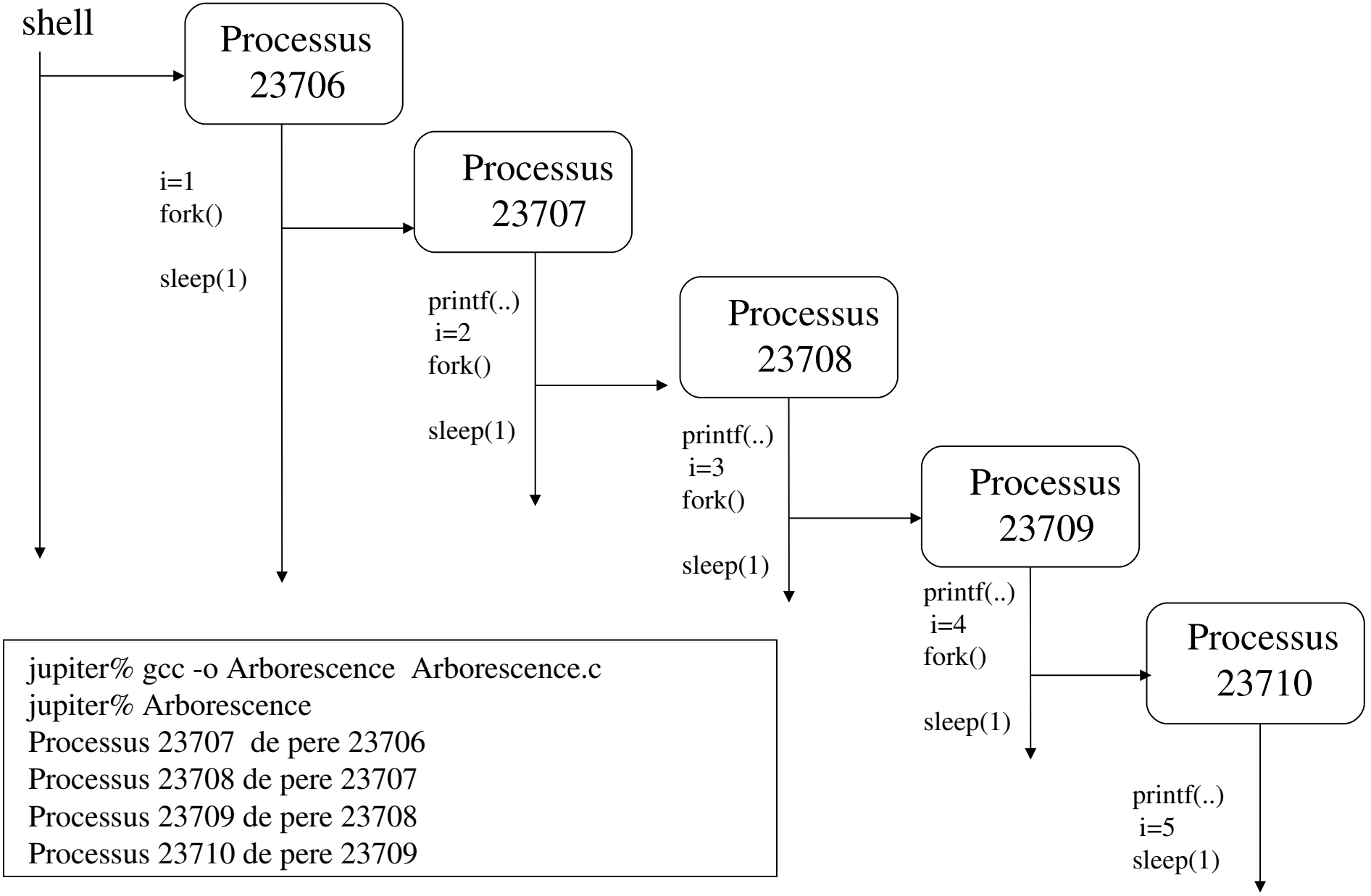

```
jupiter% tfork
ici processus père, le PID 15301.
Fin du Process 15301 avec a = 120.
ici processus fils, le PID 15302.
Fin du Process 15302 avec a = 30.
jupiter%
```



•**Exemple: Création de processus imbriqués**

```
// 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;
}
```

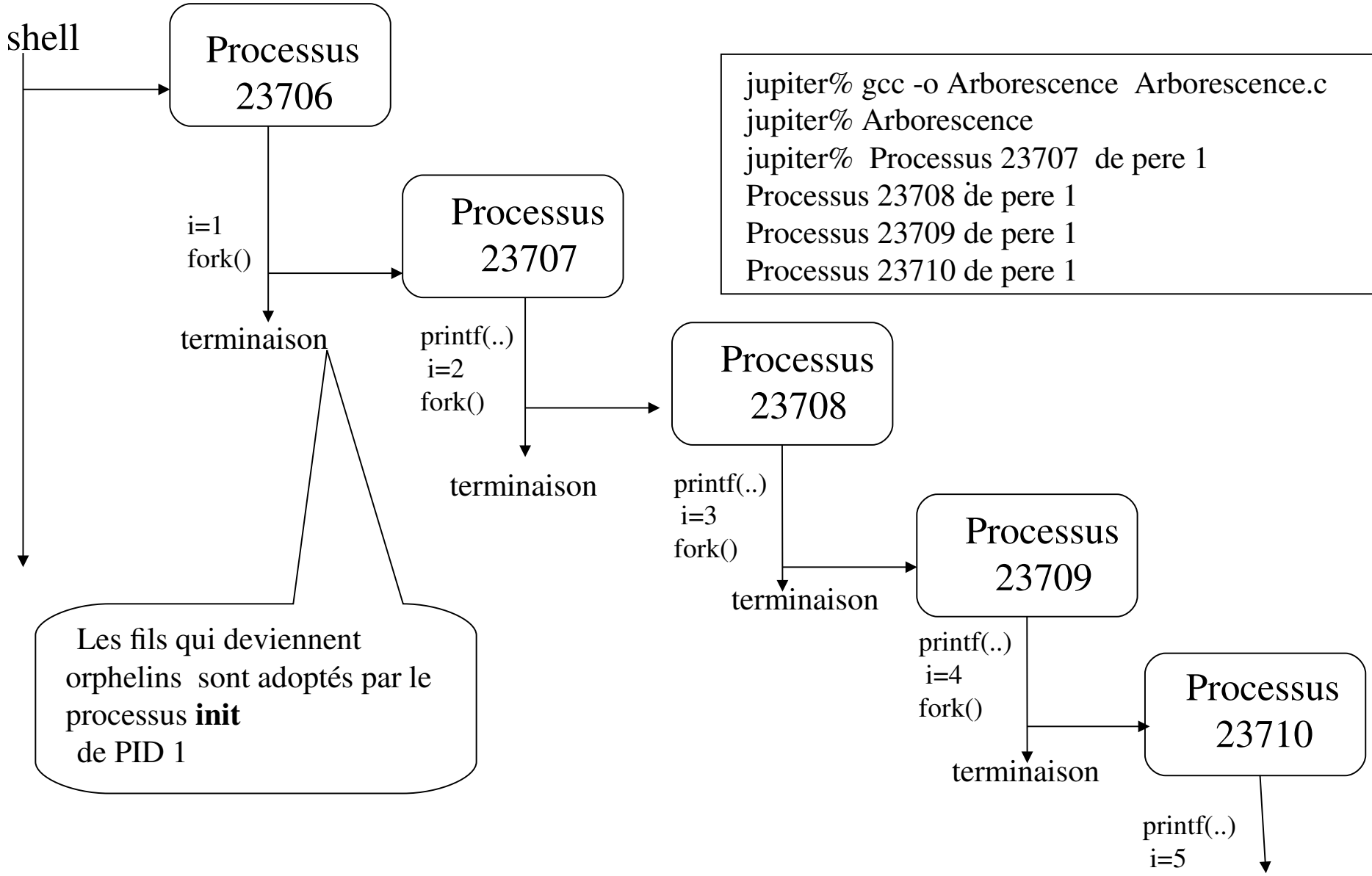
Exemple: Création de processus imbriqués (2)




```
jupiter% gcc -o Arborescence  Arborescence.c
jupiter% 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



“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

- Le système UNIX 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.

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 paramètre fixe (liste)
– execv. . . : nombre de paramètre variable (tableau comme argv)

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[]);

- 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 complet.

Appel système exec sous Unix

- 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).

Exemple: appel système exec

```
// 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;
}
```

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).

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 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;
}
```

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

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;
}
```

```
jupiter% gcc –-o texecvp texecvp.c
jupiter% texecvp date
mercredi,  6 septembre 2000,
16:12:49 EDT
Le père détecte la fin du fils

jupiter% texecvp ugg+kjù
invalide ugg+kjù
Le père détecte la fin du fils

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

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.

  void exit(int vstatus);

- 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?
- 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.

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, qui se charge à la place du père original d'appeler wait sur ces derniers.

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) > 0)
        printf("fin du fils%d\n", WEXITSTATUS(status));
    if (wait(&status) > 0)
        printf("fin du fils%d\n", WEXITSTATUS(status));
    return 0;
}

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

```
jupiter% gcc -o deuxfils deuxfils.c
jupiter% deuxfils
fin du fils1
fin du fils2
jupiter% deuxfils
fin du fils2
fin du fils1
```

Exemple: Terminaison de processus

Les liens père /fils sont maintenus

Après le fork, le père remplace son espace d'adressage par celui du programme wait_child.c

```c
// programme exec_wait.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
int main() {
    pid_t p = fork();
    if (p != 0) {
        execlp("./a.out", "./a.out", NULL);
        printf("execlp a échoué\n");
        exit(-1);
    } else {
        sleep(5);
        printf("Je suis le fils\n");
        exit(0);
    }
}
```

```c
// programme wait_child.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    printf("J'attends le fils\n");
    wait(NULL);
    printf("Le fils a terminé \n");
    exit(0);
}
```

```
$./exec_wait
J'attends le fils
Je suis le fils
Le fils a terminé
```