Programmation système sous Unix

Appels au système

Unix: un peu d'histoire

- 1ère version: 1969 (!) sur PDP-7 (DEC); par Ken Thompson et Denis Ritchie aux Bell Labs
- développé en réaction au système MULTICS du MIT (MULTICS → UNICS → UNIX)

  Unix a repris certaines idées de MULTICS:

  - système de fichiers hiérarchique

  - *shell* réalisé par un processus usager
- 1971: Unix porté sur PDP-11
- 1973: Unix réécrit en C (MULTICS était écrit en PL/1, les premières versions de Unix en assembleur)
- fin des années 70: Unix est utilisé dans de nombreuses universités (surtout aux USA)
- 1978: BSD 3 (Berkeley System Development): mémoire virtuelle ajoutée à Unix (sur DEC-VAX)
- années 80: Unix se répand partout (fin du système d'exploitation lié au constructeur)

Unix: un peu d'histoire

- 1ère version: 1969 (!) sur PDP-7 (DEC); par Ken Thompson et Denis Ritchie aux Bell Labs
- développé en réaction au système MULTICS du MIT (MULTICS → UNICS → UNIX)

  Unix a repris certaines idées de MULTICS:

  - système de fichiers hiérarchique

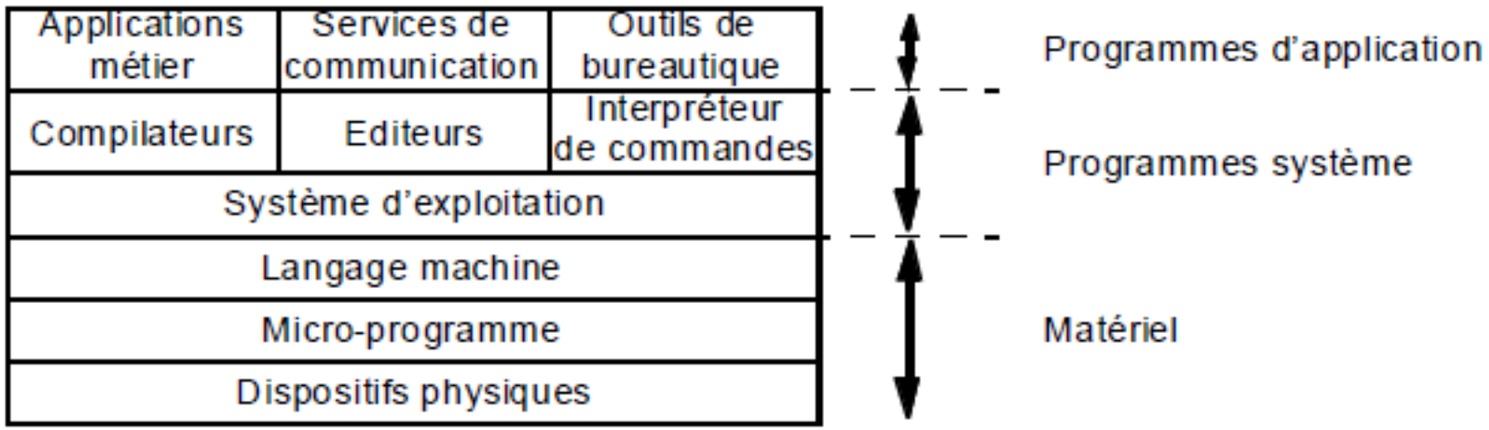
  - *shell* réalisé par un processus usager
- 1971: Unix porté sur PDP-11
- 1973: Unix réécrit en C (MULTICS était écrit en PL/1, les premières versions de Unix en assembleur)
- fin des années 70: Unix est utilisé dans de nombreuses universités (surtout aux USA)
- 1978: BSD 3 (Berkeley System Development): mémoire virtuelle ajoutée à Unix (sur DEC-VAX)
- années 80: Unix se répand partout (fin du système d'exploitation lié au constructeur)

- 1988: POSIX.1 est publié (Portable Operating System Interface)
- 1989: SVR4 (System V Release 4) unifie SVR3, Xenix, BSD et SunOS
- 1991: version 0.01 de Linux: développé par Linus Torvalds, en réaction au système Minix de Tanenbaum, qui était pour Torvalds « *juste un système d'exploitation pour étudiants* »
- 1992: Sun introduit Solaris 2.0, basé sur un nouveau noyau permettant l'exécution de threads concurrents (Solaris est basé sur SVR4)
- 1994: version 1.0 de Linux (inclut TCP/IP)
- depuis la fin des années 1990, forte expansion de Linux dans un contexte open source
- depuis 2003, expansion de Linux dans le domaine de l'industrie et des services

Plan du chapitre

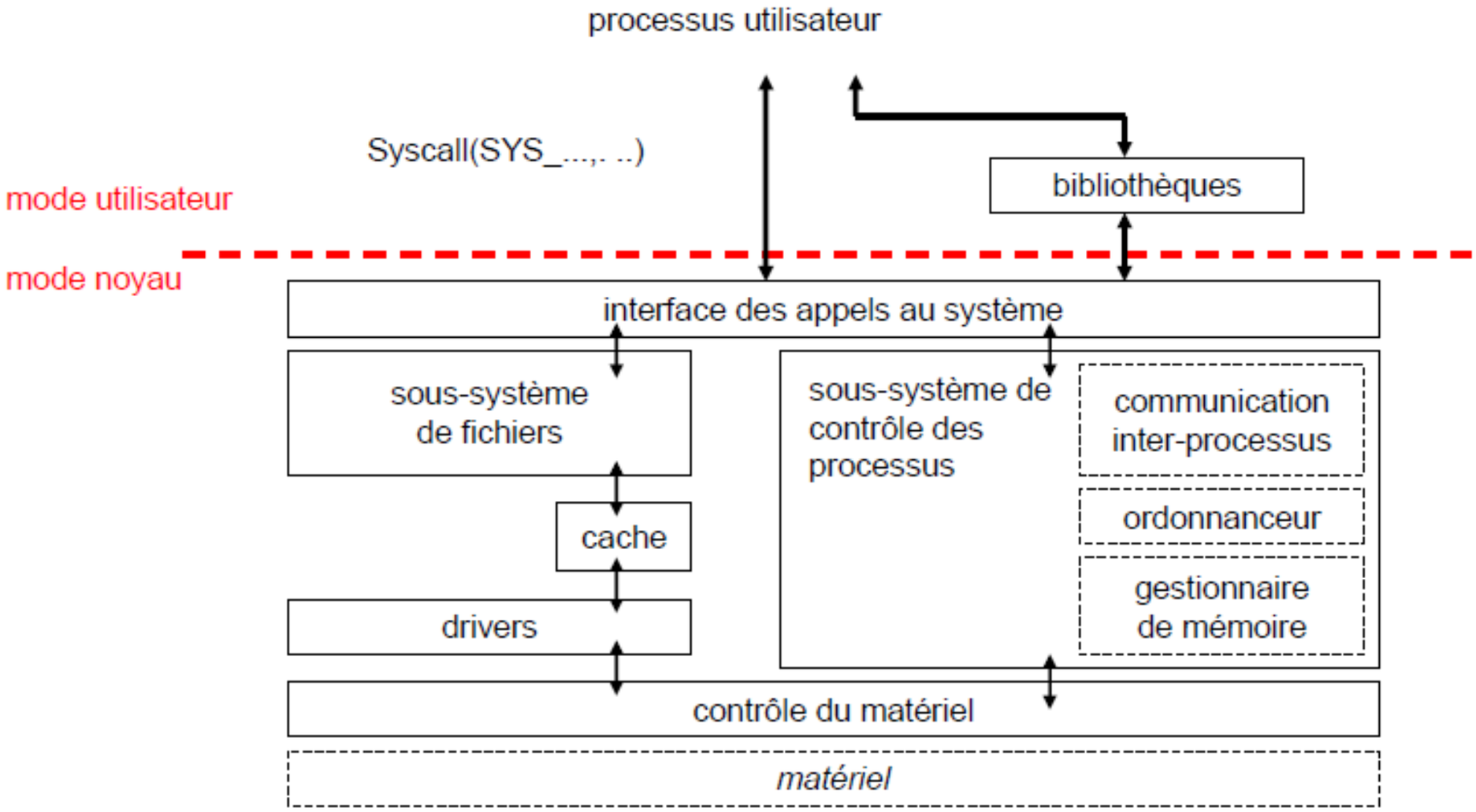
- Introduction
  - Architecture du système
  - Processus Unix
  - Fonctions sur les processus
- Synchronisation et communication
  - Signaux
  - Verrous
  - Sémaphores
  - Tubes
  - Messages
  - Mémoire partagée
  - Mappage de fichiers en mémoire principale

Les deux fonctions d'un système d'exploitation



- fournir au programmeur une abstraction (une interface de programmation commode) de la machine sur laquelle il travaille
- fournir les outils permettant de gérer les ressources de l'ordinateur et partager et protéger l'information qui y réside

Unix: architecture du noyau



Unix: rôle du noyau

- création et ordonnancement des processus (gestion du processeur)
- gestion de la mémoire principle
- gestion du système de fichiers
- gestion des périphériques
- contrôle des interruptions
- communication entre processus

Fonctions de bibliothèque

Convention:

(la plupart des fonctions de bibliothèque sont des fonctions entières)

- pas d'erreur: la fonction retourne 0
- erreur: la fonction retourne –1 et la variable `errno` contient un code d'erreur (la variable externe `errno` n'est pas affectée par un appel qui a réussi)

```
#include <errno.h>
extern int errno

#define  EINTR      // Interrupted system call
#define  EACCESS    // Permission denied
```

Appel au système

Exemple d'application de la convention:

- le registre D0 contient le numéro de la fonction du système appelée
- au retour de l'appel système le bit C (Carry) du registre d'état du processeur indique si l'appel a réussi (C = 0) ou échoué (C = 1)
- si C = 1, le registre D0 contient un numéro d'erreur

(Ex. suite) création d'un fichier (le fichier n'est pas ouvert)

Programme C:

```
...
int fd;
fd = create (name, 0666);
...
```

Assembleur:

...

- empile 1er paramètre
- empile 2ème paramètre
- JSR .. (appel de la fonction *create)*

...

Code de la fonction de bibliothèque *create*:

charger *0* dans la variable *errno*

charger code de la fonction système *create* dans registre *D0*

TRAP / SYSCALL (SYS_creat,..)

if bit *C* = 0 (pas d'erreur) **goto** *fin*

erreur:

copie *D0* (no erreur) dans la variable *errno*

résultat de la fct *create* mis à *–1*

RTS (return from subroutine)

fin:

résultat de la fct *create* mis à *0*

RTS (return from subroutine)

Notion de processus sous Unix

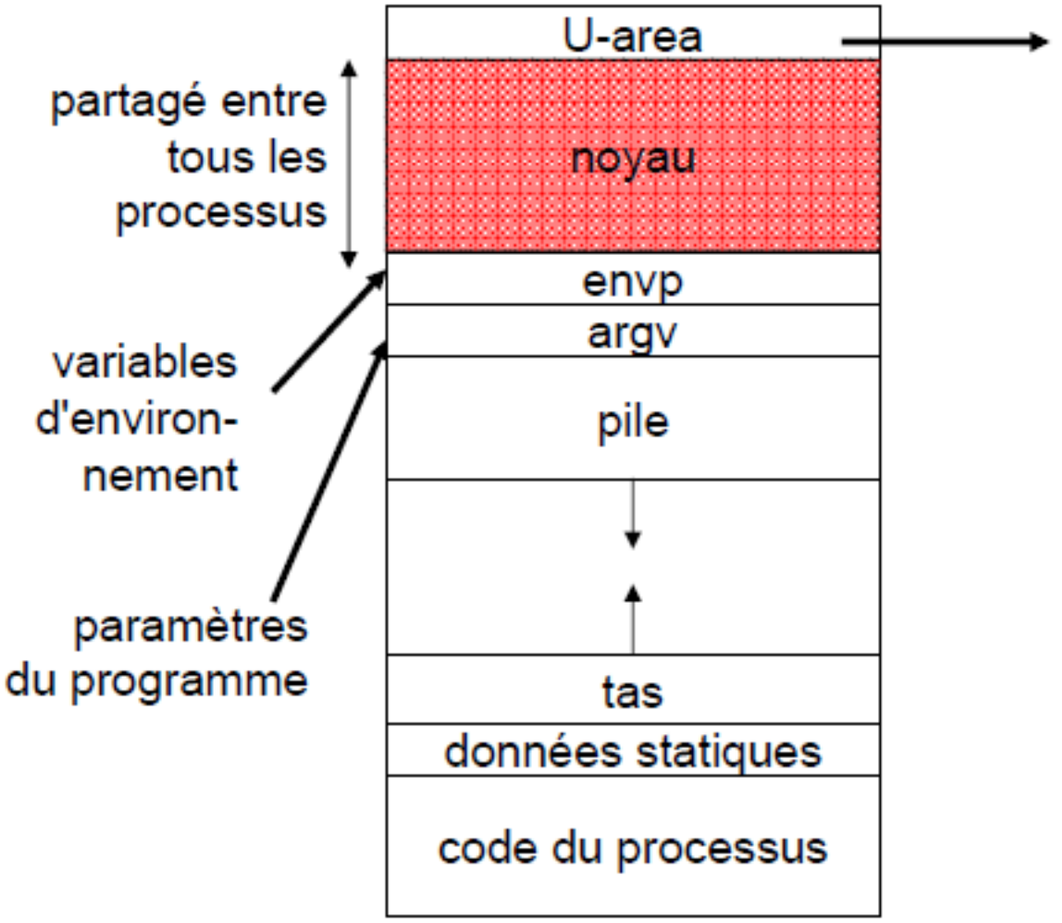
Un processus sous Unix est composé de:

- un espace d'adressage
- un (ou plusieurs) threads (flots de contrôle)
- un ensemble de ressources (fichiers ouverts, etc.)

Deux structures fournissent l'information de contrôle d'un processus:

- **U-area** (user area): information nécessaire uniquement lorsque le processus s'exécute
- **descripteur de processus:** information nécessaire même lorsque le processus ne s'exécute pas. Structure de donnée identifiée sous le nom `proc` (prédéfinie dans `<sys/proc.h>`). C'est une entrée de la **table des processus** du noyau

Espace d'adressage d'un processus



- sauvegarde du contexte hardware (lorsque le prss ne s'exécute pas)
- pointeur sur la structure `proc`
- UID et GID réels et effectifs
- "handlers" des signaux
- descripteurs des fichiers ouverts
- répertoire courant et terminal de contrôle
- utilisation CPU, quotas disque, etc.
- pile "noyau" du processus

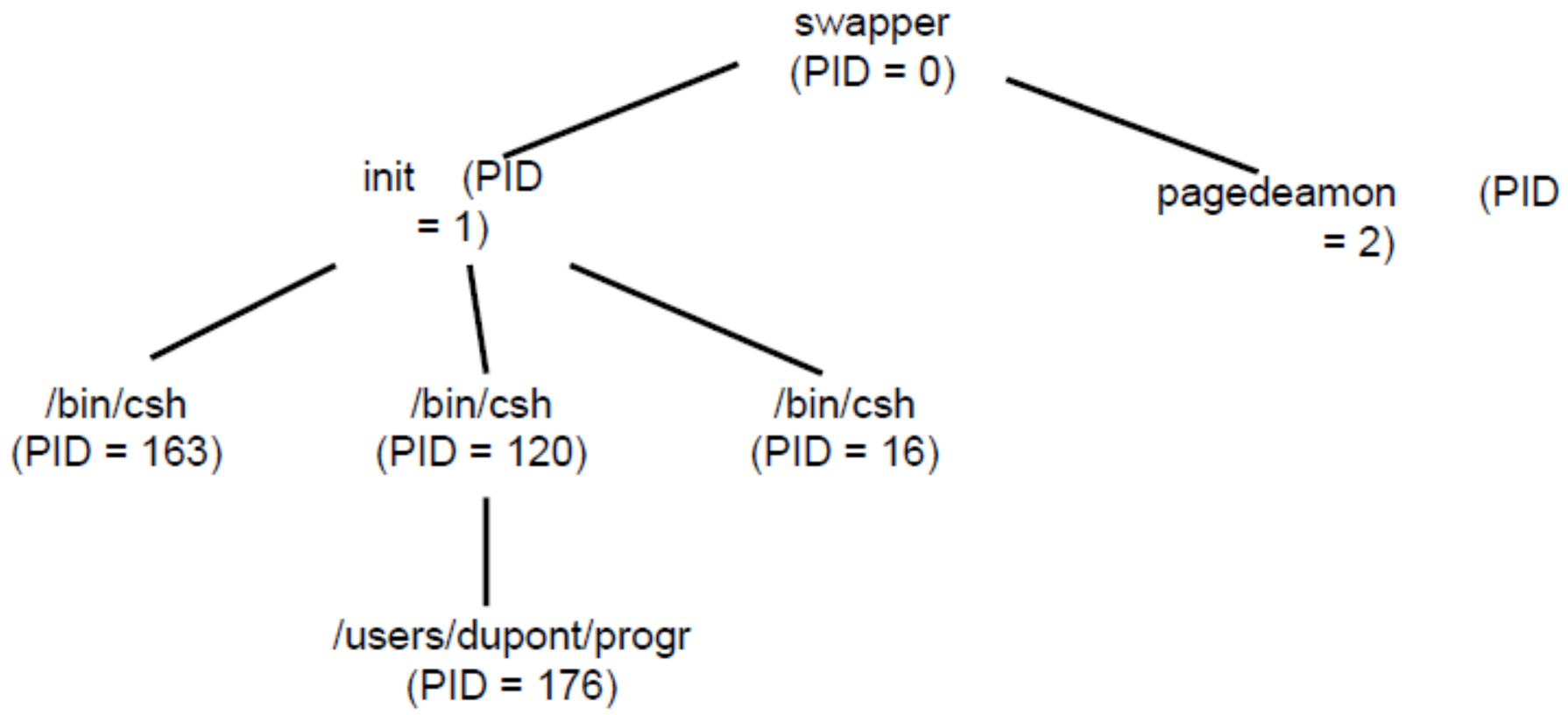
Structure `proc`

- identificateur du processus (PID)
- pointeur sur U-area du processus
- liens avant et arrière de chaînage (pour inclure la structure `proc` dans une des listes du noyau)
- priorité du processus
- informations liées à la gestion des signaux (signaux bloqués, à ignorer, etc.)
- information de gestion de la mémoire
- liens pour inclure la structure dans une des listes de processus *actifs*, *libres* ou *zombies*
- informations indiquant la relation du processus par rapport à d'autres processus (pointeur sur `proc` du père, pointeur sur `proc` du 1er fils, etc.)
- …

Identification d'un processus (liée à la gestion)

- PID: identificateur unique du processus (nombre entier)
- PPID: identificateur du processus père
- PGRP: identificateur du groupe auquel appartient le processus (nombre entier)
  - par défaut, hérité du processus-père
  - permet de désigner un ensemble de processus

Hiérarchie de processus



Identification de l'utilisateur (liée à la protection)

Contrôle de l'accès au système de fichiers:

- UID: identificateur réel (hérité du père, p.ex: du shell qui a lancé un programme)
- EUID: identificateur effectif (p.ex: exécution d'un programme avec le bit S mis à 1)
- GID: identificateur de groupe réel
- EGID: identificateur de groupe effectif

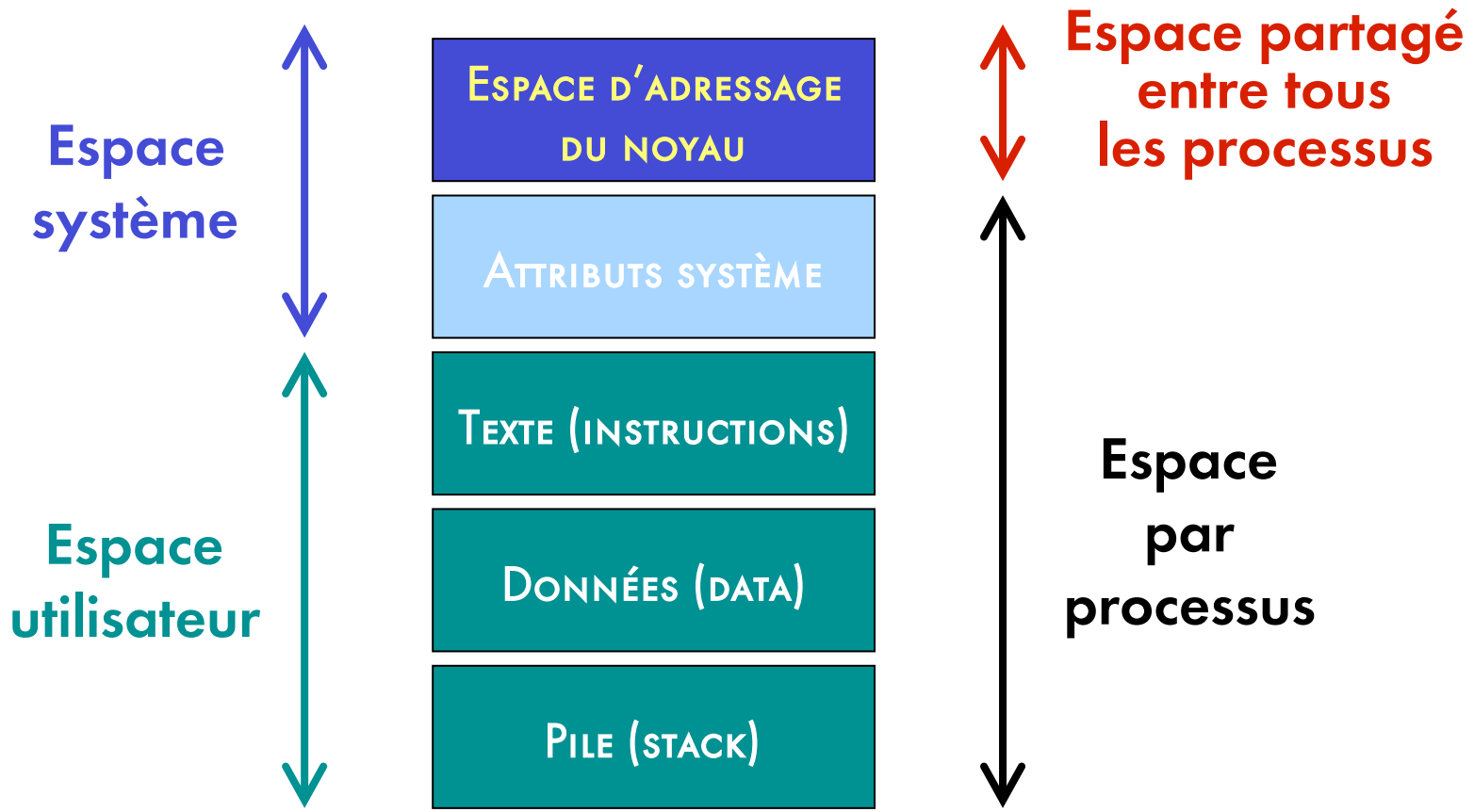
Appels au système liés à l'identification

- PID, PPID, PGRP
  - `getpid( ), getppid( ), getpgrp( )`
  - `setpgrp( )`
- UID, EUID
  - `getuid( ), getruid( ), geteuid( )`
  - `setuid( ), setruid( ), seteuid( ), setreuid( )`
- GID, EGID
  - `getgid( ), getrgid( ), getegid( )`
  - `setgid( ), setrgid( ), setegid( ), setregid( )`

Fonctions sur les processus: création et terminaison

- `fork`
- `exit`
- `wait`
- `exec`

Espace d'adressage d'un processus (1)

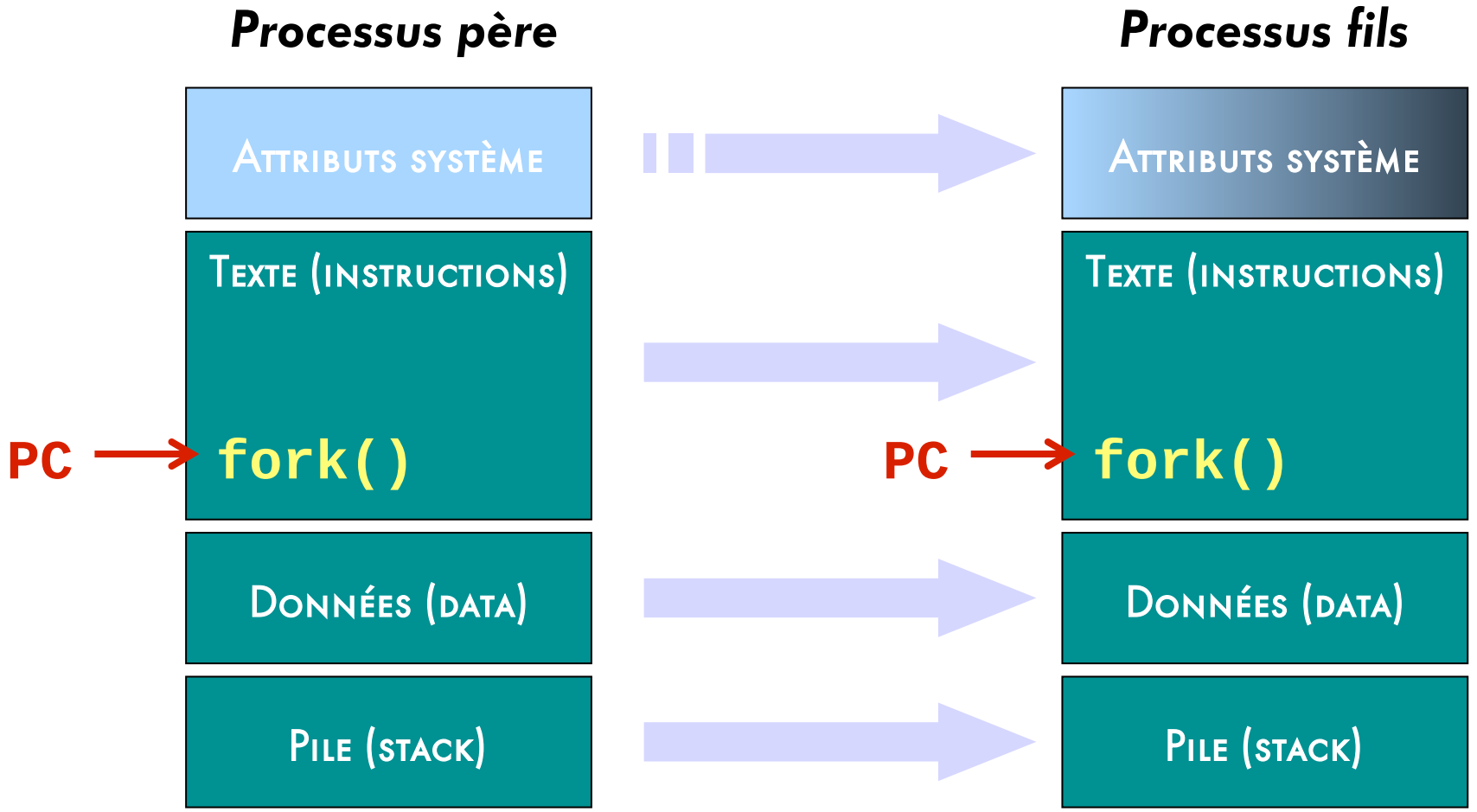


Espace d'adressage d'un processus (2)

- **Attributs système d'un processus**
  - identification (**`pid`**) : unique à un instant donné
  - **`uid`**, **`gid`** effectifs et réels
  - descripteurs de fichiers ouverts
  - racine et répertoire courants
  - états des signaux
  - masque de création des fichiers (**`cmask`**)
  - *adresses (mémoire, disque), information de gestion de mémoire virtuelle, priorité, etc.*

Création d'un processus *fork()*

(1)



Création d'un processus *fork()*

(2)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

- **Création de processus par « clonage »**
  - Duplication des segments de texte, de données, de piles et de la plupart des attributs système
- **Après `fork()`, les deux processus exécutent le même programme, mais *indépendamment***

Création d'un processus *fork()* (3)

- **`fork()` est donc appelé une fois mais a deux retours**
  - un dans le fils, avec la valeur 0
  - un dans le père avec comme valeur le pid du fils
- **Héritage des attributs système**
  - (descripteurs de) fichiers ouverts
    - **le pointeur d'E/S est partagé entre le père et le fils**
  - **`uid`**, **`gid`**, répertoire courant, terminal de contrôle, masque de création, état des signaux, etc.

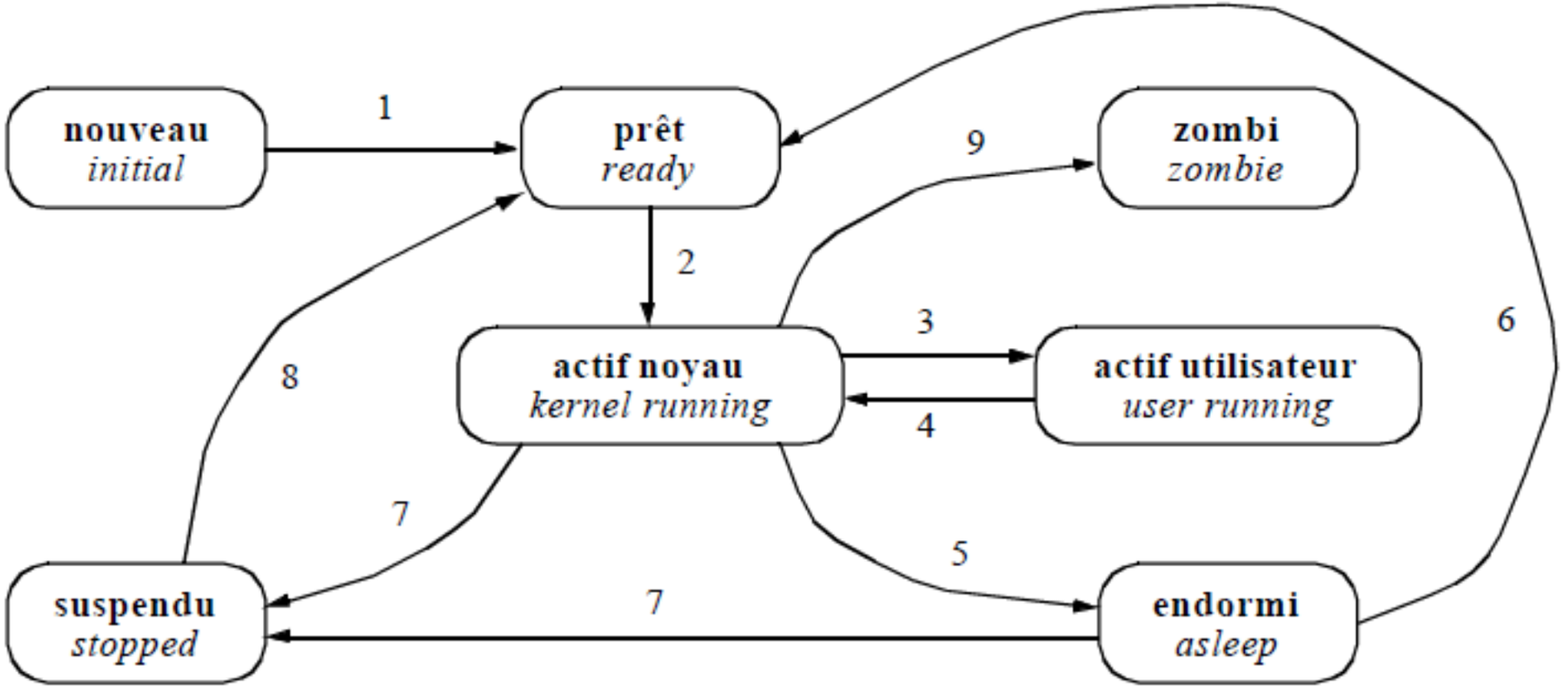
Création d'un processus *fork()* (4)

```
switch (fork()) {
case -1 :
  perror("fork");
  exit(1);
case 0 :
  printf("le fils\n");
  break;
default :
  printf("le père\n");
  break;
}
printf("père+fils\n");
```

```
% test-fork
père
fils
père+fils
père+fils
%
```

- L'ordre d'exécution entre le père et le fils peut être quelconque

Etats d'un processus Unix



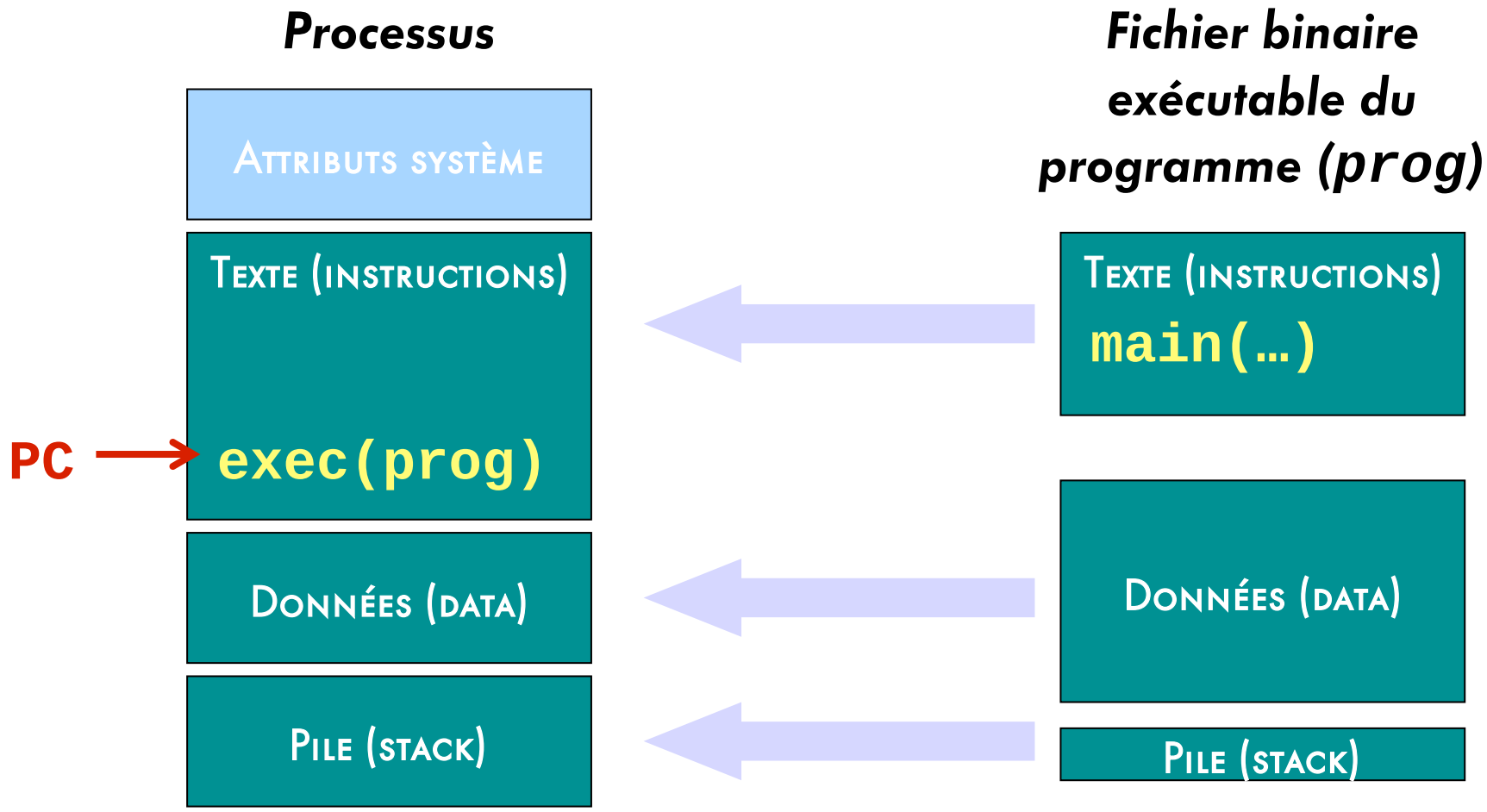
Conditions des transitions

1) le processus a acquis les ressources nécessaires à son exécution
2) le processus vient d'être élu par l'ordonnanceur : il y a alors changement de contexte
3) le processus revient d'un appel système ou d'une interruption (par exemple il vient d'être élu ou il a terminé l'exécution du handler d'une interruption)
4) le processus a réalisé un appel système ou une interruption est survenue (périphérique ou horloge)
5) le processus se met en attente d'un événement: appel système bloqué par un événement interne au système (libération de ressource ou terminaison d'un processus par exemple) ou extérieur (interruption)
6) l'événement attendu par le processus s'est produit
7) délivrance d'un signal particulier (SIGSTOP ou SIGTSTP)
8) réveil du processus par le signal SIGCONT
9) le processus se termine

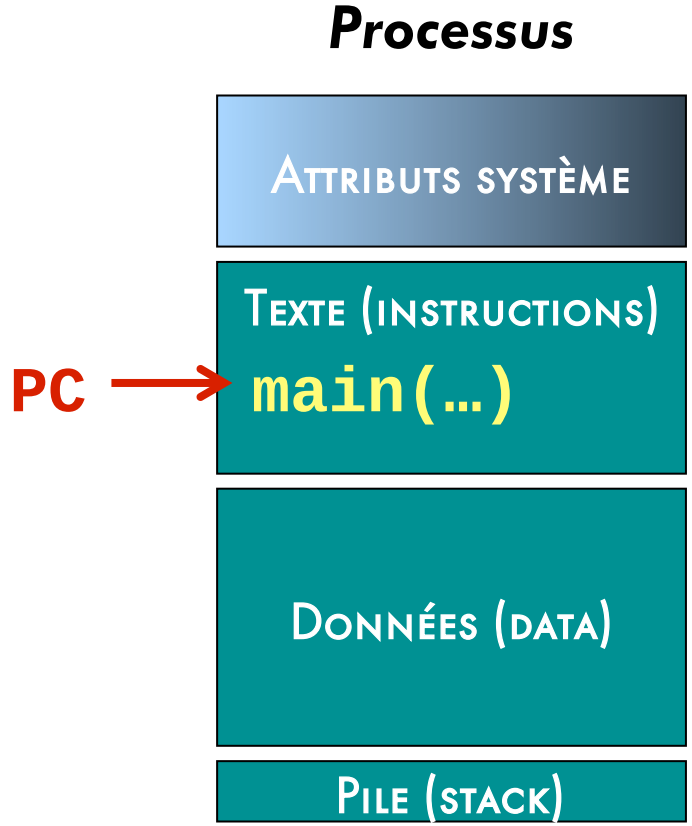
Lorsqu'il est endormi, un processus est en attente d'un événement. Les événements sont représentés par une adresse mémoire dont l'état est modifié par l'occurrence de l'événement associé. Une caractéristique importante d'Unix est que l'occurrence d'un événement réveille a priori (c'est-à-dire fait repasser de l'état endormi à l'état prêt) tous les processus en attente de cet événement.

Rifflet, Yunès 2003

Association programme/processus
exec( ) *(1)*



Association programme/processus
exec( ) (2)



- **Le pid du processus n'a pas changé**
  - c'est le même processus
- **Le code a changé**
  - il exécute un autre programme
  - ce programme démarre au début (`main()`)
- **L'état de l'ancien programme est oublié**
  - on ne peut revenir d'un **exec** réussi !

Association programme/processus
exec( ) (3)

```
#include <unistd.h>
int execl(const char *path,
          const char *arg0, ..., NULL);
int execv(const char *path, char *const argv[]);

int execle(const char *path, const char *arg0,..., NULL,
           char **envp);
int execve(const char *path, char *const argv[],
           char **envp);

int execlp(const char *path,
           const char *arg0,... , NULL);
int execvp(const char *path, char *const argv[]);
```

Association programme/processus
exec( ) (4)

- **Remplacement des segments d'un processus par ceux d'un programme pris dans leur état initial**
- **Argument**
  - Le fichier à exécuter (**`path`**)
  - les arguments du **`main`**
    - **`arg0`** ou **`argv[0]`** est le nom (de base) du fichier à exécuter
    - La liste (ou le tableau **`argv[]`**) se termine avec un pointeur **`NULL`**

Association programme/processus
exec( ) (5)

- **exec[lv]p utilisent la variable PATH**
- **exec[lv]e passent l'environnement en dernier paramètre**
- **Conservation de la plupart des attributs système**
  - (descripteurs) de fichiers ouverts
    - avec la même valeur du pointeur d'E/S qu'avant **exec( )**
  - **uid**, **gid**, répertoire courant, terminal de contrôle, masque de création, certains états des signaux, etc.

Association programme/processus
exec() (6)

```
printf("début\n");
execlp("ls", "ls", "-l", "-R",
       "/usr", NULL);
/* On ne passe ici
   qu'en cas d'erreur de exec
*/
perror("exec");
```

Terminaison volontaire d'un processus *exit()* *(1)*

```
#include <stdlib.h>
void exit(int status);
void abort();

#include <unistd.h>
void _exit(int status);
```

- Toutes ces fonctions terminent le processus courant
- **_exit()** et **exit()** transmettent le code de retour **status** au processus père
- **abort()** produit un fichier **core** (signal **SIGABRT**)

Terminaison volontaire d'un processus *exit()* (2)

- **Terminaison normale : `exit()`**
  - appelle les fonctions enregistrées par **`atexit()`**
  - « flushe » tous les fichiers de **stdio**
  - détruit les fichiers temporaire (**`tempfile()`**)
  - appelle **`_exit()`**
- **Terminaison forcée : `_exit()`**
  - ferme tous les fichiers et répertoires
  - *réveille le processus père (si nécessaire)*
  - *provoque éventuellement l'adoption du processus, etc.*

Attente d'un processus fils
wait()

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

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

- **Attente de la terminaison d'un fils**
  - **wait()** est réveillé par la fin d'un fils quelconque
  - **waitpid()** est réveillé par la fin du fils indiqué
- **Retour immédiat si un/le fils déjà terminé**

Version simplifiée de system() (1)

```
#define BAD 1

int System(const char *cmd) {
   int status;

   if (fork()) {
       wait(&status);
       return status;
   } else {
       execl("/bin/sh", "sh", "-c", cmd, NULL);
       perror("exec");
       return BAD;
   }
}
```

Version simplifiée de system() (1)

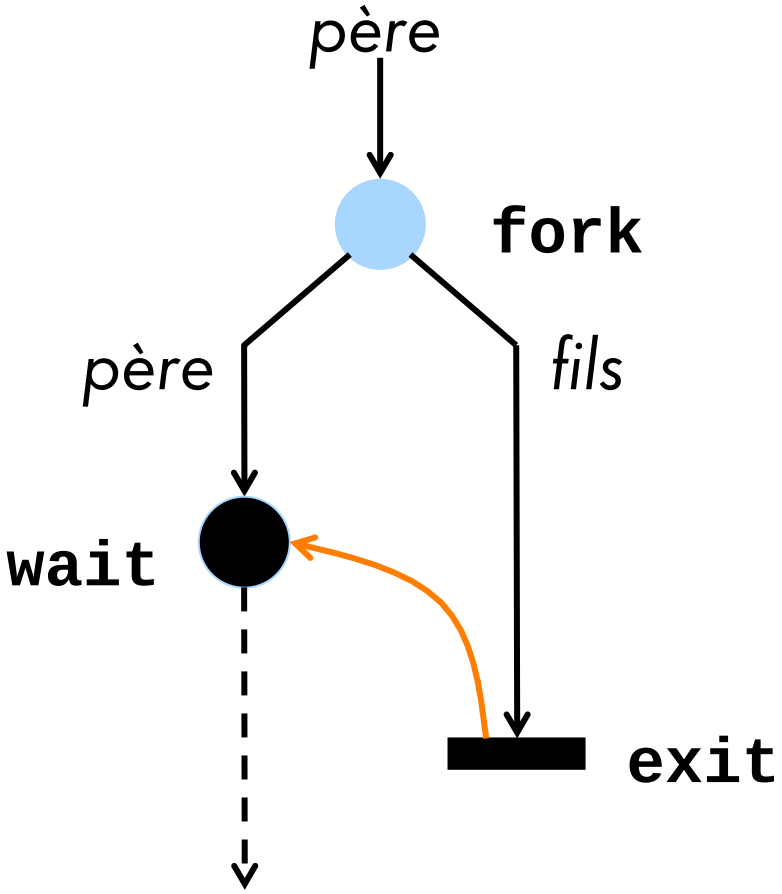
```
#define BAD 1

int System(const char *cmd) {
   int status;

   if (fork()) {
       wait(&status);
       return status;
   } else {
       execl("/bin/sh", "sh", "-c", cmd, NULL);
       perror("exec");
       exit(BAD);
   }
}
```

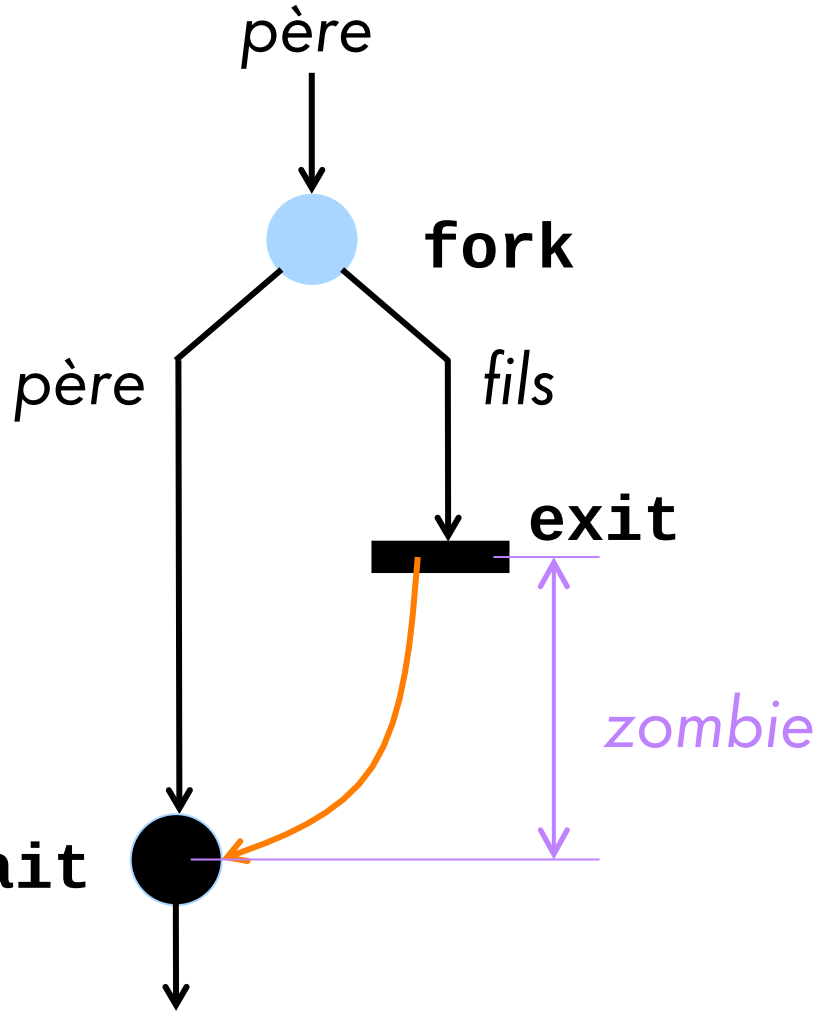
Version simplifiée de `system()` (2)

- **Exemple**

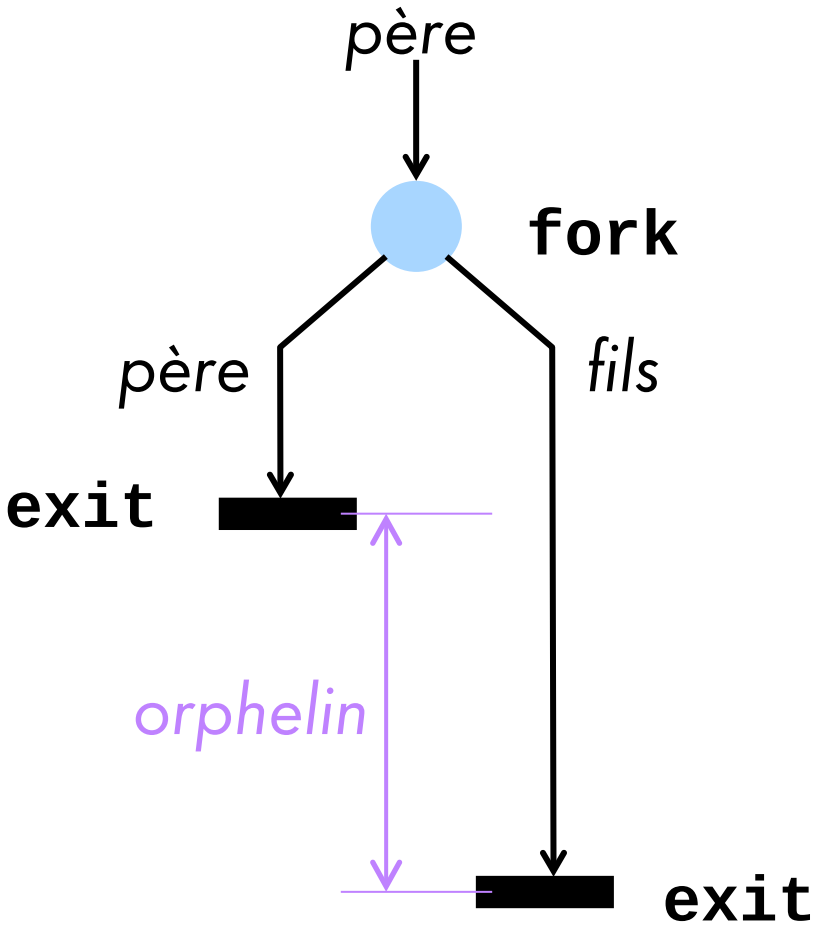
```
int status =
  System("ls -la -R /usr > foo");
```

Processus père et fils : « zombie » et orphelin (1)





Processus père et fils : « zombie » et orphelin (2)



- Les orphelins sont adoptés par le processus de pid 1
- Ce processus 1 est associé au programme `/etc/init`
- Ce processus est aussi le gestionnaire du temps partagé