Université Nice Sophia Antipolis

Processus

Présentation: Stéphane Lavirotte
Auteurs: ... et al*

(*) Cours réalisé grâce aux documents de :
Stéphane Lavirotte, Jean-Paul Rigault

Mail: Stephane.Lavirotte@unice.fr
Web: http://stephane.lavirotte.com/
Université de Nice - Sophia Antipolis

Définition d'un Processus

- ✓ Aspect dynamique d'un programme
- ✓ Unité de gestion d'activité
  - Exécution d'un programme
  - Unité d'allocation des ressources système
    - Fichiers, événements, périphériques, mémoire...
  - Espace d'adressage
- ✓ Unité d'ordonnancement
  - Flot de contrôle séquentiel
  - Entité affectable à un processeur

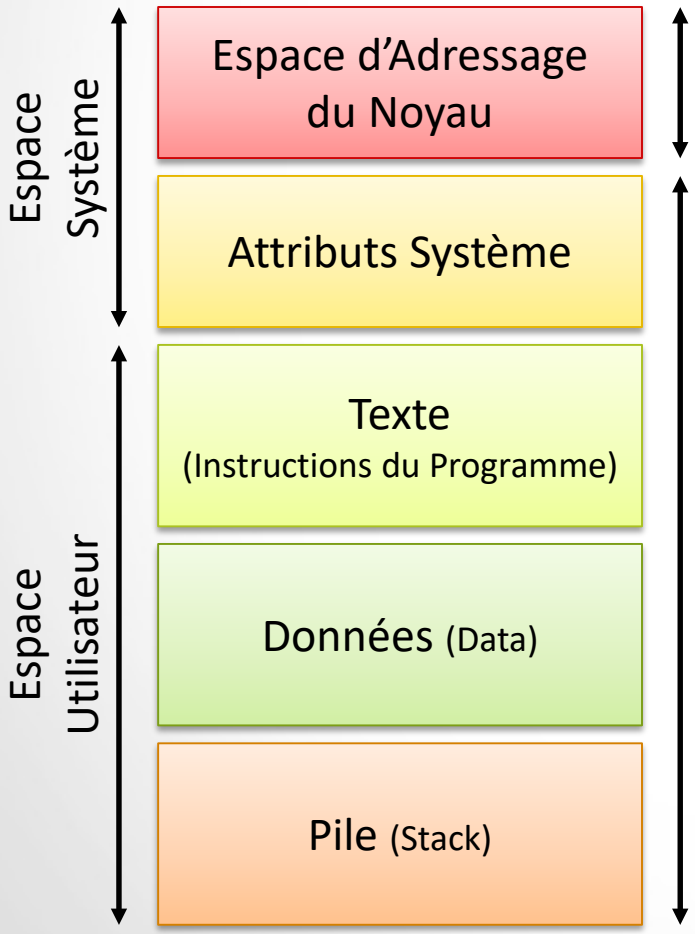
Gestion de Processus

La norme Posix.1

Processus sous Unix

- ✓ Unix traditionnels
  - Processus = unité gestion activité + unité d'ordonnancement
    - Un processus correspond à l'exécution séquentielle d'un programme, mais ce programme peut changer
- ✓ Unix modernes
  - Processus « lourd » = unité gestion activité
  - Processus « léger » (« thread ») = unité d'ordonnancement
    - Nous détaillerons cela dans un prochain cours

Espace d'Adressage d'un processus



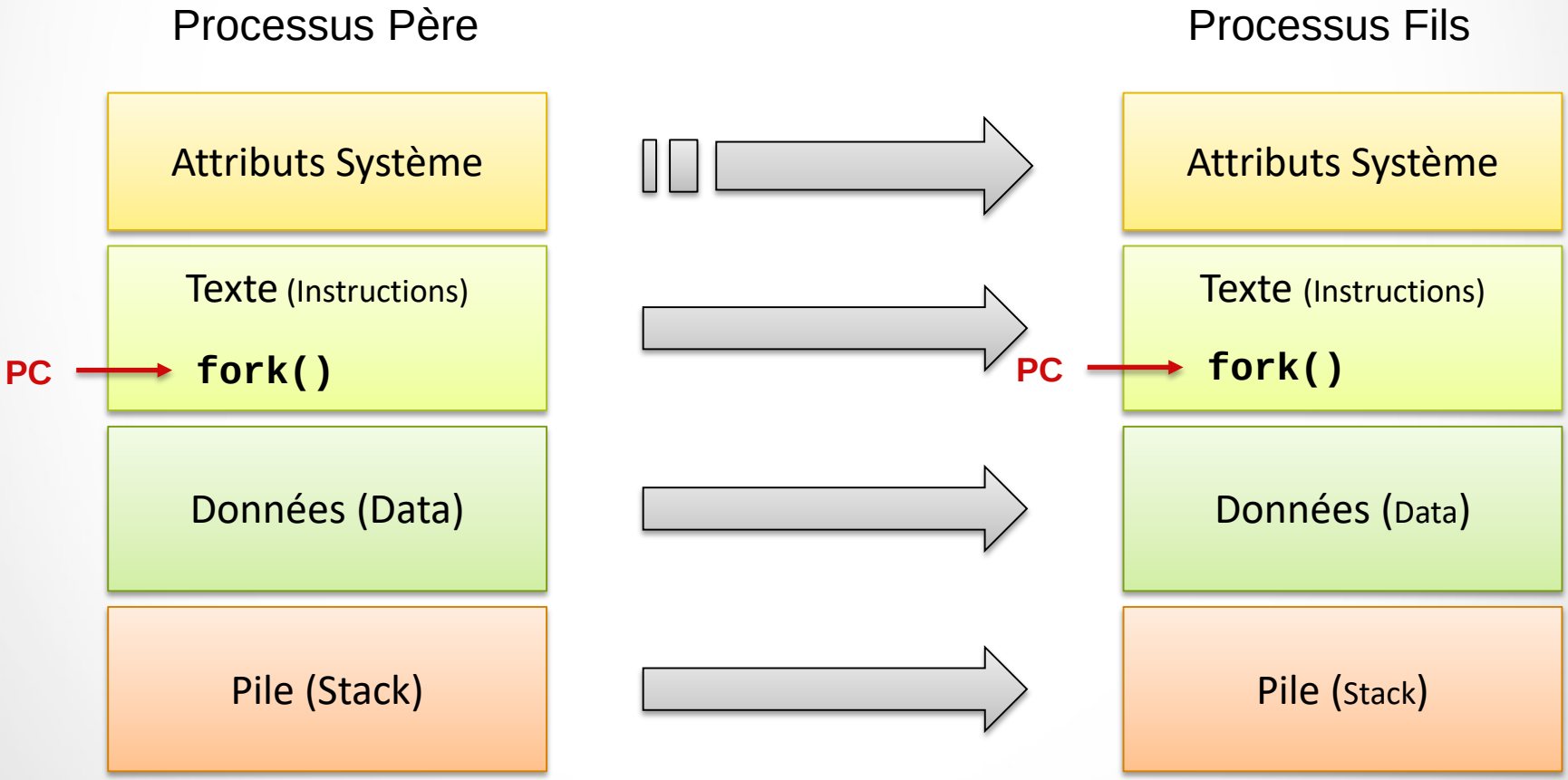
- ✓ Espace d'adressage
  - Zone de données système (attributs)
  - Zone de texte : instructions
    - non inscriptible, partageable
  - Zone de données
    - variables statiques et externes
  - Zone de pile
    - variables locales automatiques
    - gestion des sous-programmes
- ✓ Binaire exécutable
  - État initial de l'espace d'adressage

Université Nice Sophia Antipolis

Attributs d'un processus

- ✓ Attributs
  - Informations nécessaires à la gestion par le système
  - Sauvegardé et restauré à chaque changement de contexte
  - Contenu
    - Identification du processus
    - Identification de l'utilisateur
    - Ressources possédées (fichiers...)
    - Informations statistiques et comptables...
    - Contexte matériel
- ✓ 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() 1/3



Création d'un Processus `fork()` 2/3

```
#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
- ✓ `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.

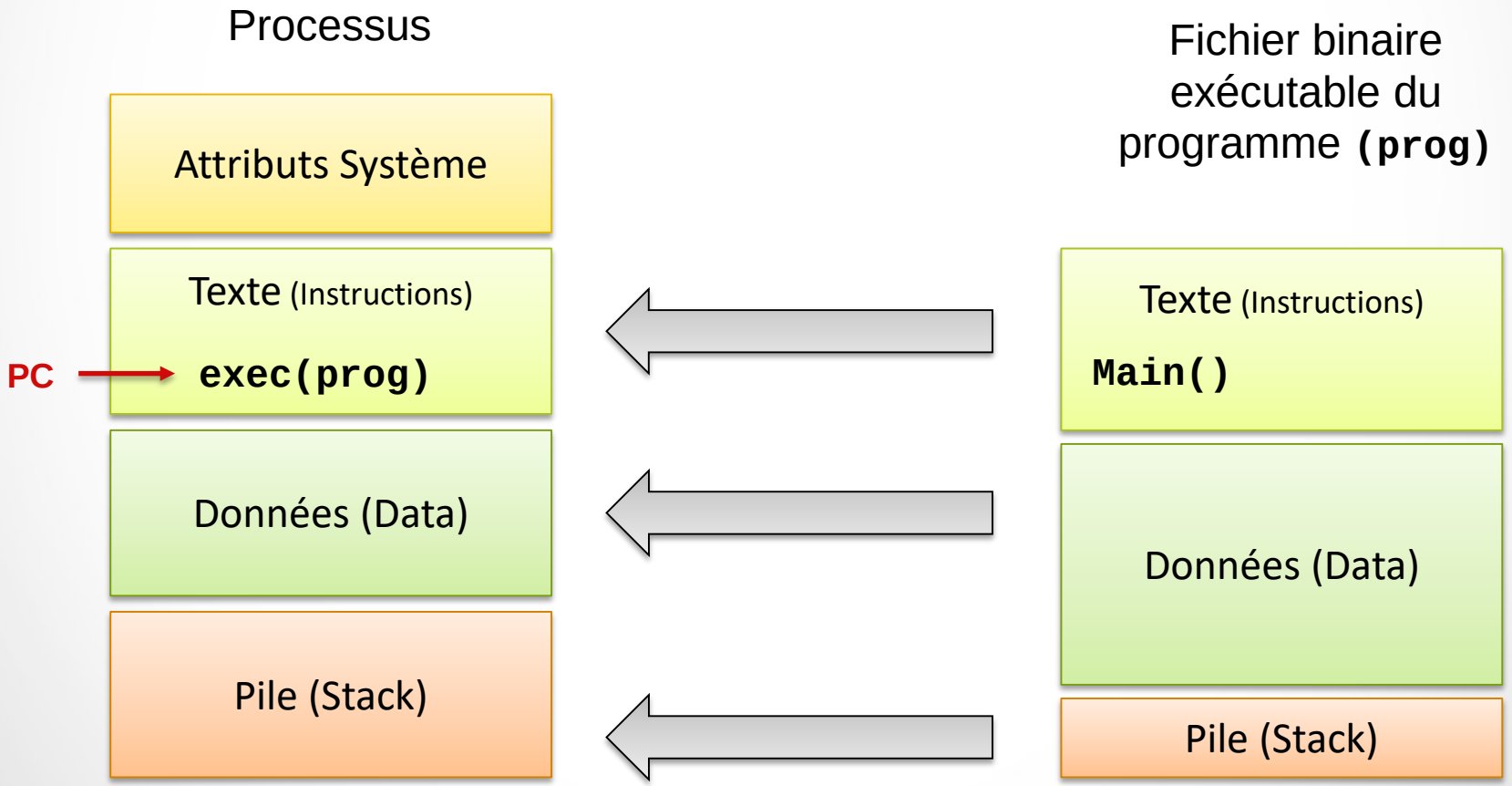
Création d'un Processus fork() 3/3

```
int i = 0;
switch (fork()) {
    case -1 :
        perror("fork");
        exit(1);
    case 0 :
        ++i;
        printf(”fils: %d\n”, i);
        break;
    default :
        i += 2;
        printf(“père: %d\n”, i);
        break;
}
++i;
printf("père+fils: %d\n”, i);
```

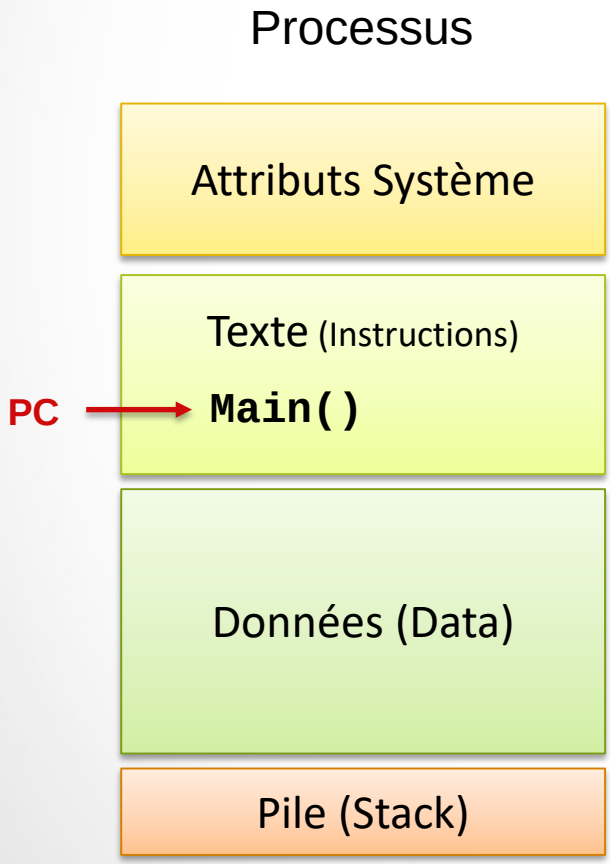
```
% test-fork
père: 2
fils: 1
père+fils: 3
père+fils: 2
%
```

- ✓ *L'ordre d'exécution entre le père et le fils est quelconque*

Association Programme/Processus
exec( ) 1/5



Association Programme/Processus
exec() 2/5



- ✓ 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() 3/5

- ✓ 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
- ✓ `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() 4/5

```
#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[]);
int execve(const char *path, char *const argv[],
           char * const envp[]);
```

Association Programme/Processus
exec() 5/5

- ✓ **Exemple d'appel:**

```
printf("début\n");

execlp("ls", "ls", "-l", "-R", "/usr", NULL);

/* On ne passe ici qu'en cas d'erreur de exec */

perror("exec");
```

Nom du programme a exécuter (avec le chemin si pas dans le PATH)

Nom donné au programme (ce sera la valeur de `argv[0]`)

Passage des différents paramètres

Terminaison volontaire d'un processus `exit()` 1/2

```
#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`**)**

Université Nice Sophia Antipolis

Terminaison volontaire d'un processus `exit()` 2/2

- ✓ **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.

Université Nice Sophia Antipolis

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 est déjà terminé**

Version simplifiée de `system()`

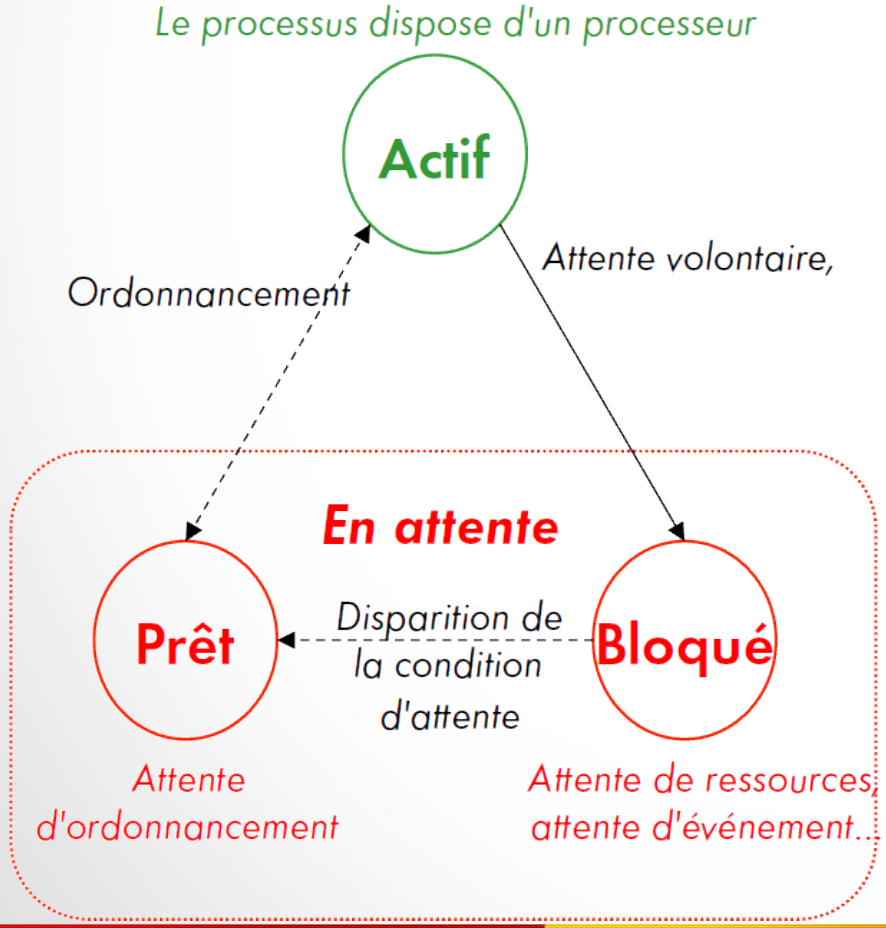
```
#define BAD 1
int System(const char *cmd) {
    int status;

    switch (fork()) {
        case -1 :
            perror("fork");
            exit(1);
        case 0 :
            execl("/bin/sh", "sh", "-c", cmd, NULL);
            perror("exec");
            exit(BAD); // et non return BAD;
        default:
            wait(&status);
            return status;
    }
}
```

✓ **Exemple**

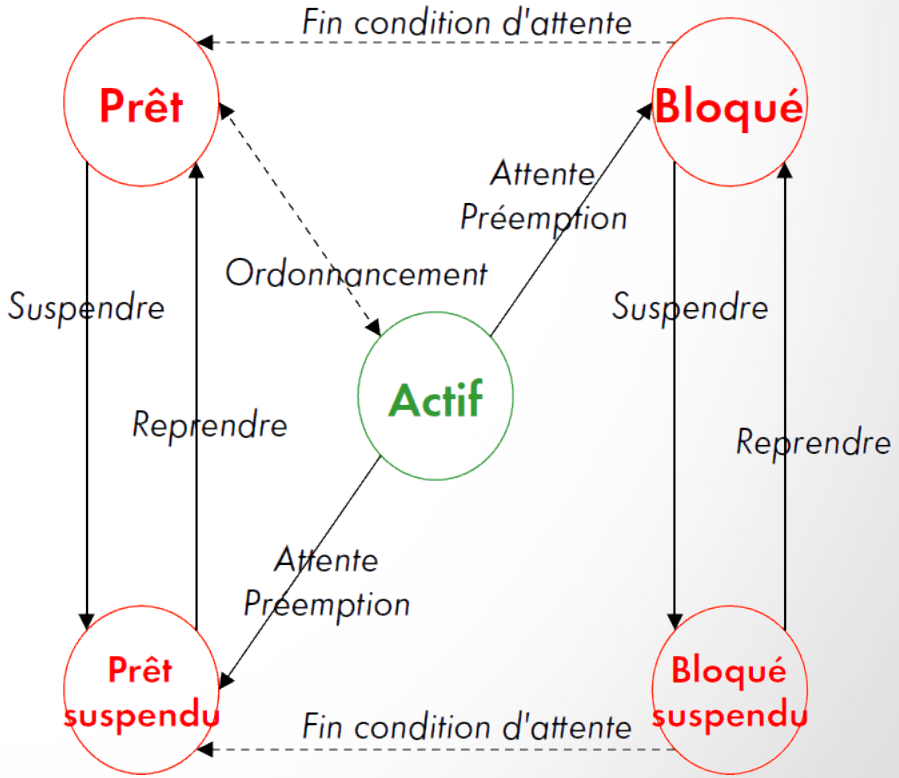
```
int status = System("ls –la –R /usr > foo");
```

Etat d'un Processus

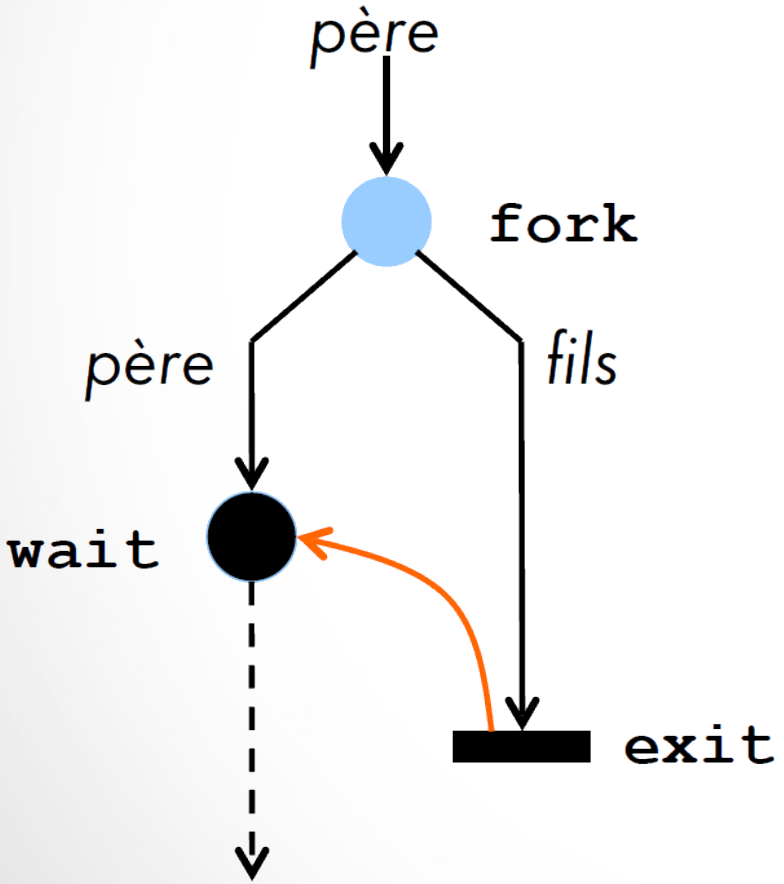
Etats Fondamentaux

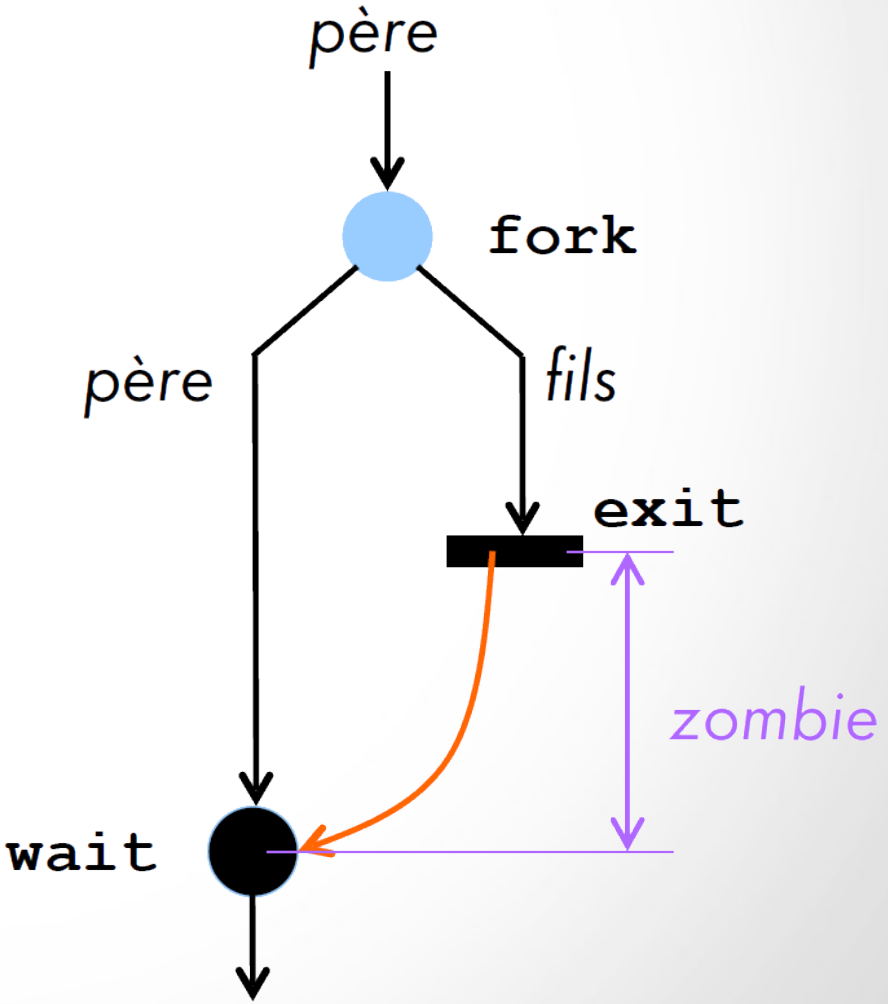


Un Graphe plus Réaliste

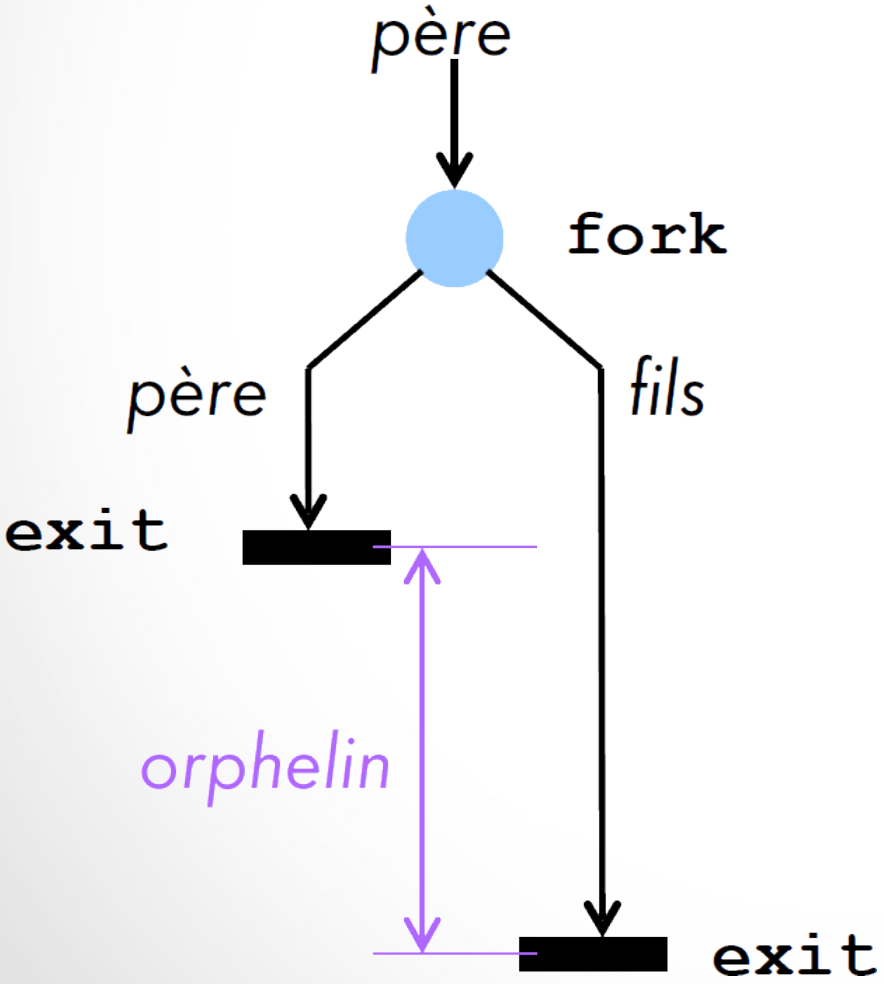


Processus père et fils: « Zombie »





Processus père et fils: Orphelin



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

Identification des Processus

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

- ✓ **Processus courant**

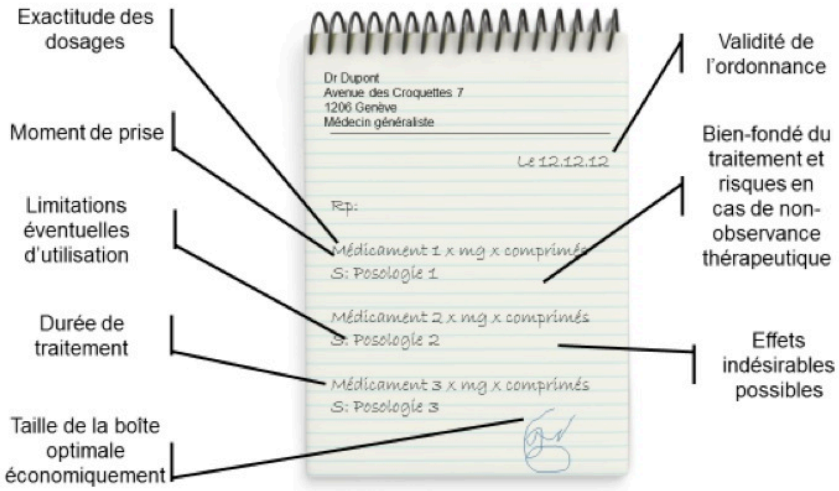
```
pid_t getpid();
```

- ✓ **Processus père**

```
pid_t getppid();
```

- ✓ **Groupe de processus**

```
int setpgrp();
pid_t getgrp();
```



Ordonnancement

Quelques notions sur l'ordonnancement

Ordonnancement Généralités 1/2

- ✓ Choix du (des) processus actif(s)
  - Quand ? Lequel (parmi les processus prêts) ?
- ✓ Mode de préemption
  - Ordonnancement préemptif
    - Le système peut retirer l'UC au processus actif à tout moment
    - Meilleure réponse
    - Garantie que le processus « le plus prioritaire » sera actif dès qu'il sera prêt
  - Ordonnancement non-préemptif
    - Le processus actif doit abandonner volontairement l'UC
    - Plus simple à programmer

Ordonnancement Généralités 2/2

- ✓ Objectifs de l'ordonnancement
  - Équité
    - Ne pas (trop) favoriser une classe particulière de processus
  - Efficacité, rendement
    - Maximiser l'activité (« throughput »)
    - Rentabiliser l'unité centrale
  - Temps de réponse, interactivité
  - Prédictibilité
  - Dégradation « gracieuse » sous forte charge
  - etc.

Ordonnancement Algorithmes Divers

- ✓ FIFO
  - Non préemptif
  - Formellement équitable, pratiquement inéquitable
- ✓ À tourniquet (« round robin »)
  - Préemption par tranche de temps (quantum)
- ✓ Le plus court d'abord (SJF)
  - Non préemptif
  - Imprévisible (gros travaux)
  - Connaissance a priori du temps d'exécution
- ✓ Le temps restant le plus court d'abord (SRT)
  - Amélioration de SJF, préemptif
  - Plus grand coût que SJF
  - Connaissance a priori du temps d'exécution
- ✓ Loterie (!)
- ✓ Ordonnancement temps réel
  - Taux monotone (« rate monotonic »)
  - Plus courte échéance d'abord
- ✓ Etc.

Ordonnancement à Priorité

- ✓ On associe une priorité (un entier) à chaque processus
  - En général, 0 désigne le plus haute priorité
- ✓ L'ordonnanceur choisit le processus de plus haute priorité parmi les processus prêts
  - Si l'ordonnancement est préemptif, le processus actif est toujours le (l'un des) processus prêt(s) de plus haute priorité
- ✓ Priorités fixes
  - La priorité est associée de manière fixe au processus
  - Elle n'est modifiable que par la volonté du programmeur
    - c'est-à-dire de ce processus, ou d'un autre processus coopérant
  - Très utilisé en temps réel
- ✓ Priorités variables
  - Le système peut « adapter » la priorité à des fins d'optimisation
  - Le programmeur peut éventuellement proposer une priorité (initiale)

Ordonnancement à Priorité / Tranche de Temps

Exemple: Unix BSD