4 – Processus et tubes

mickael.hoerdt@hesge.ch

Contenu

Introduction et rappels

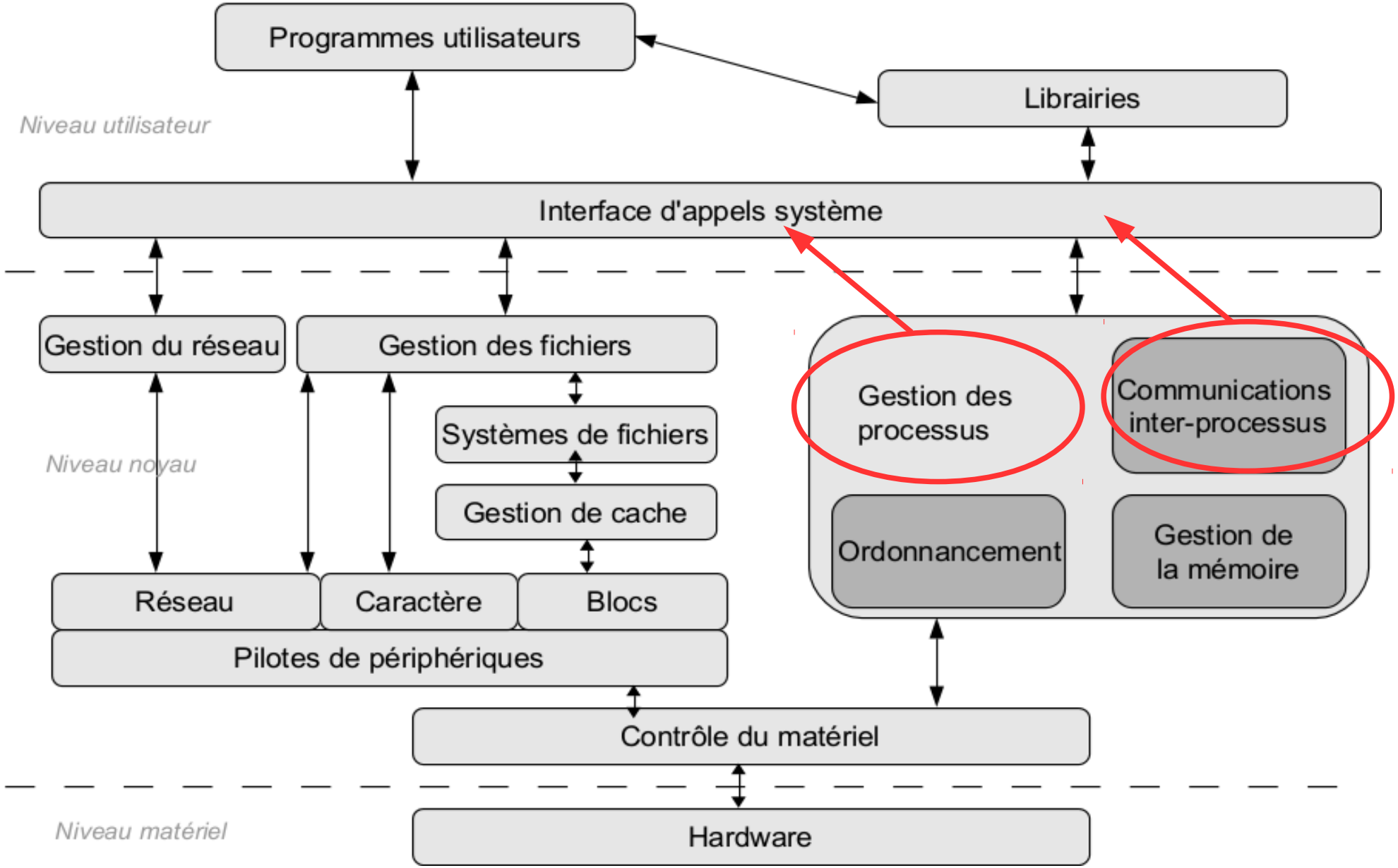
Gestion des processus sous Unix

Appels système pour la gestion des processus

Communications inter-processus : les tubes

Appels système pour la gestion des tubes

Rappels sur la vue d'ensemble



Introduction

Gérer efficacement les ressources matérielles par la définition de ressources logiques.

- Fichiers stocké = unité d'accès aux périphériques de stockage
  - Voir chapitre système de fichier - entrées/sortie
- Processus = unité d'accès aux CPUs.

Rappel : définition d'un processus

Un processus est une instance de programme en cours d'exécution (par un processeur)

- Un même programme peut très bien être exécuté par deux processus différents

```
mickael@computer:~$ echo $$
1792
mickael@computer:~$ bash
mickael@computer:~$ echo $$
2138
mickael@computer:~$ exit 10
exit
mickael@computer:~$ echo $?
10
mickael@computer:~$ echo $$
1792
mickael@computer:~$
```

- Pour gérer un processus, le système d'exploitation définit des attributs de processus

Rappel : processus UNIX

Attributs systèmes principaux des processus :

- Numéro unique sur le système : PID (Process Identifier)
- Identifiant d'utilisateur : UID
- Identifiant de groupe : GID
- PID du processus parent : PPID (Parent Process Identifier)
- Terminal d'attache utilisé pour les entrées/sorties/erreurs
- Descripteur de fichiers ouverts

Contenu

Introduction et rappels

Gestion des processus sous Unix

Appels système pour la gestion des processus

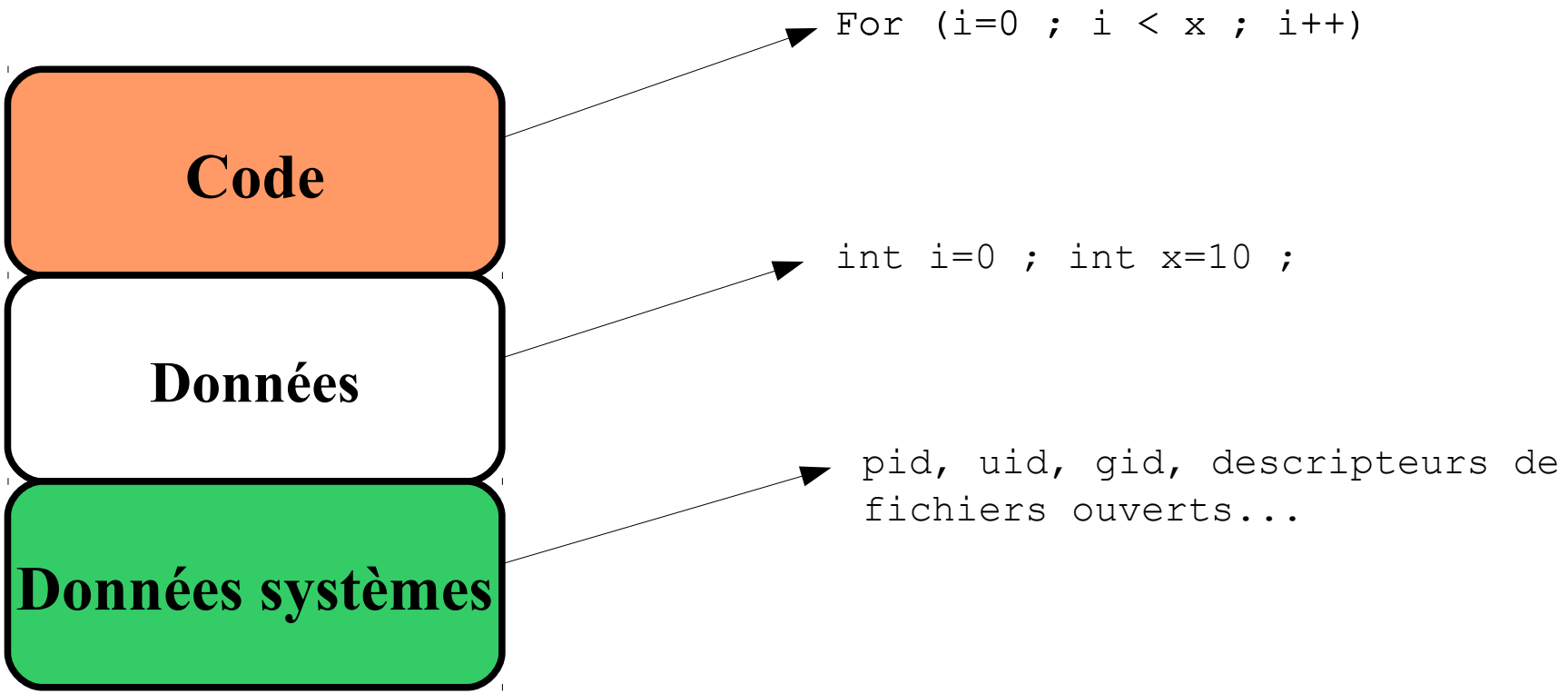
Communications inter-processus : les tubes

Appels système pour la gestion des tubes

Composition d'un processus

Trois parties :

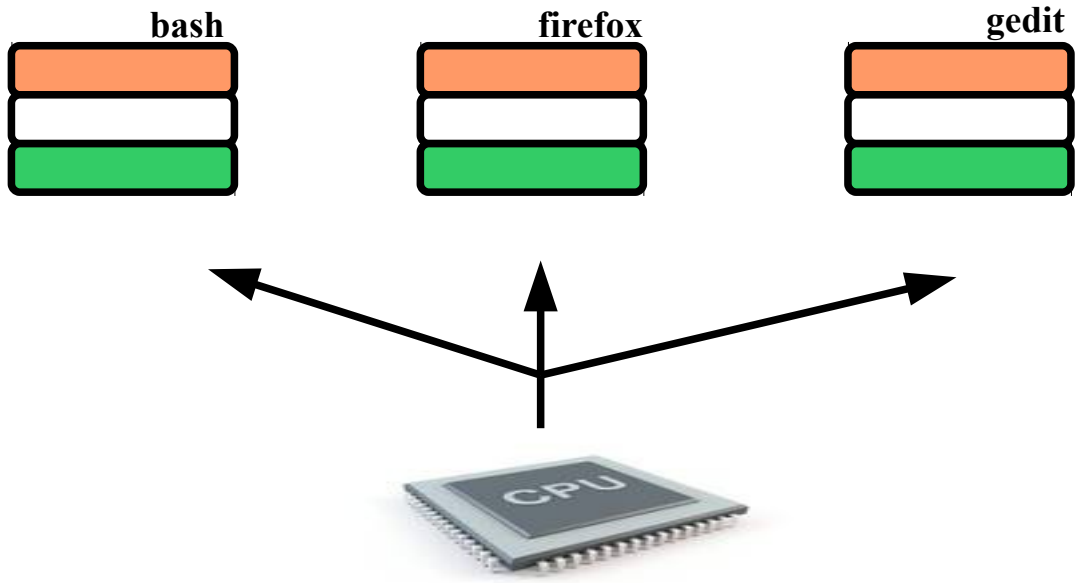
- Code
- Données utilisateur
- Attributs systèmes



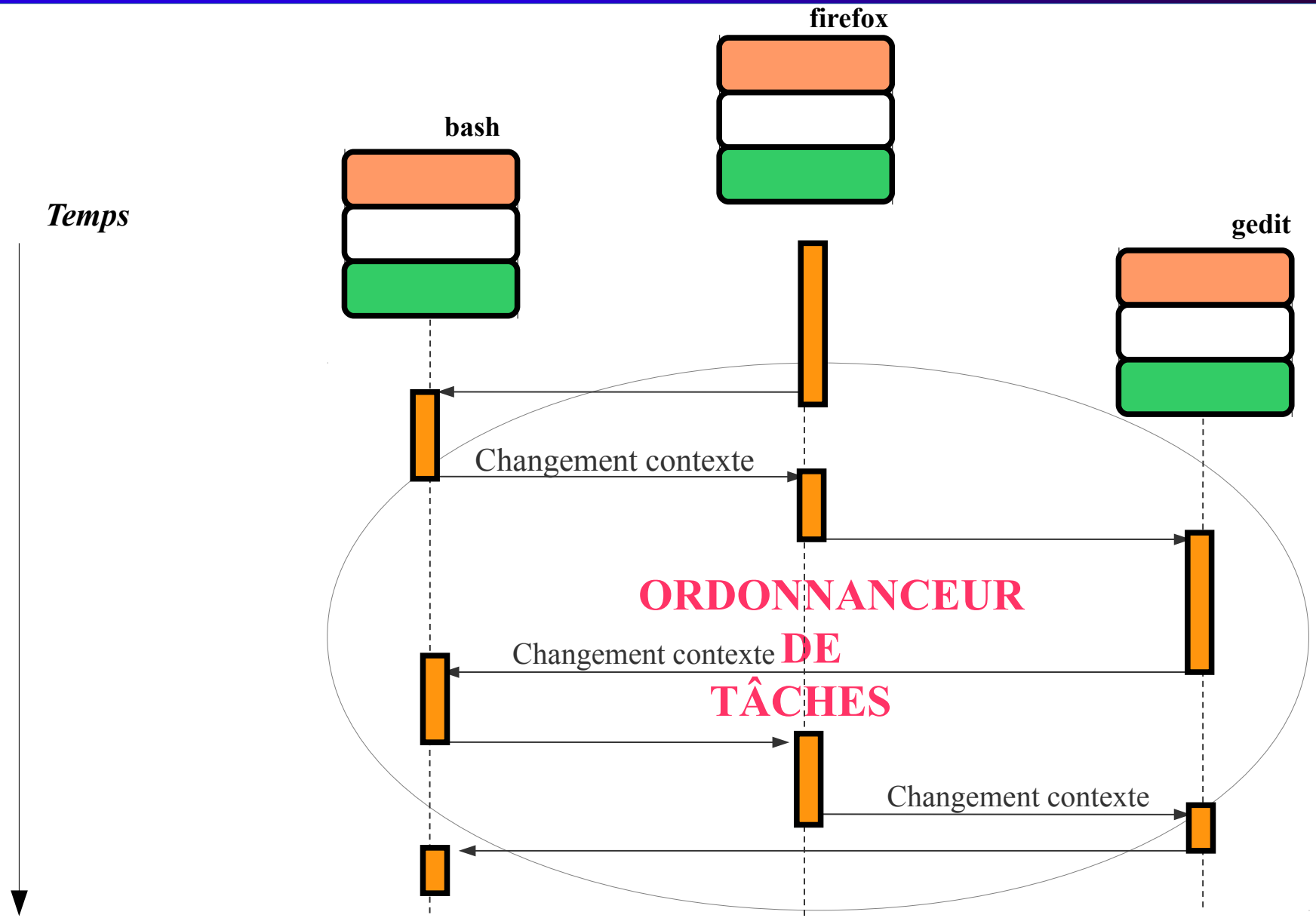
Systèmes UNIX : multitâche

Prendre un cliché des processus en cours sur la machine : voir la la commande unix `ps` (process snapshot) et `htop`

Comment gérer un nombre de processus largement supérieur au nombre de CPUs ?



Processus : pseudo-parallélisme



Changement de contexte d'exécution

L'ordonnanceur change de contexte d'un processus par une boucle qui comporte les séquences suivantes :

1. **Mise en attente** du processus actif (fin du quantum de temps)

2. **Sauvegarde** de son contexte d'exécution

3. **Recherche** du processus éligible ayant la plus haute priorité

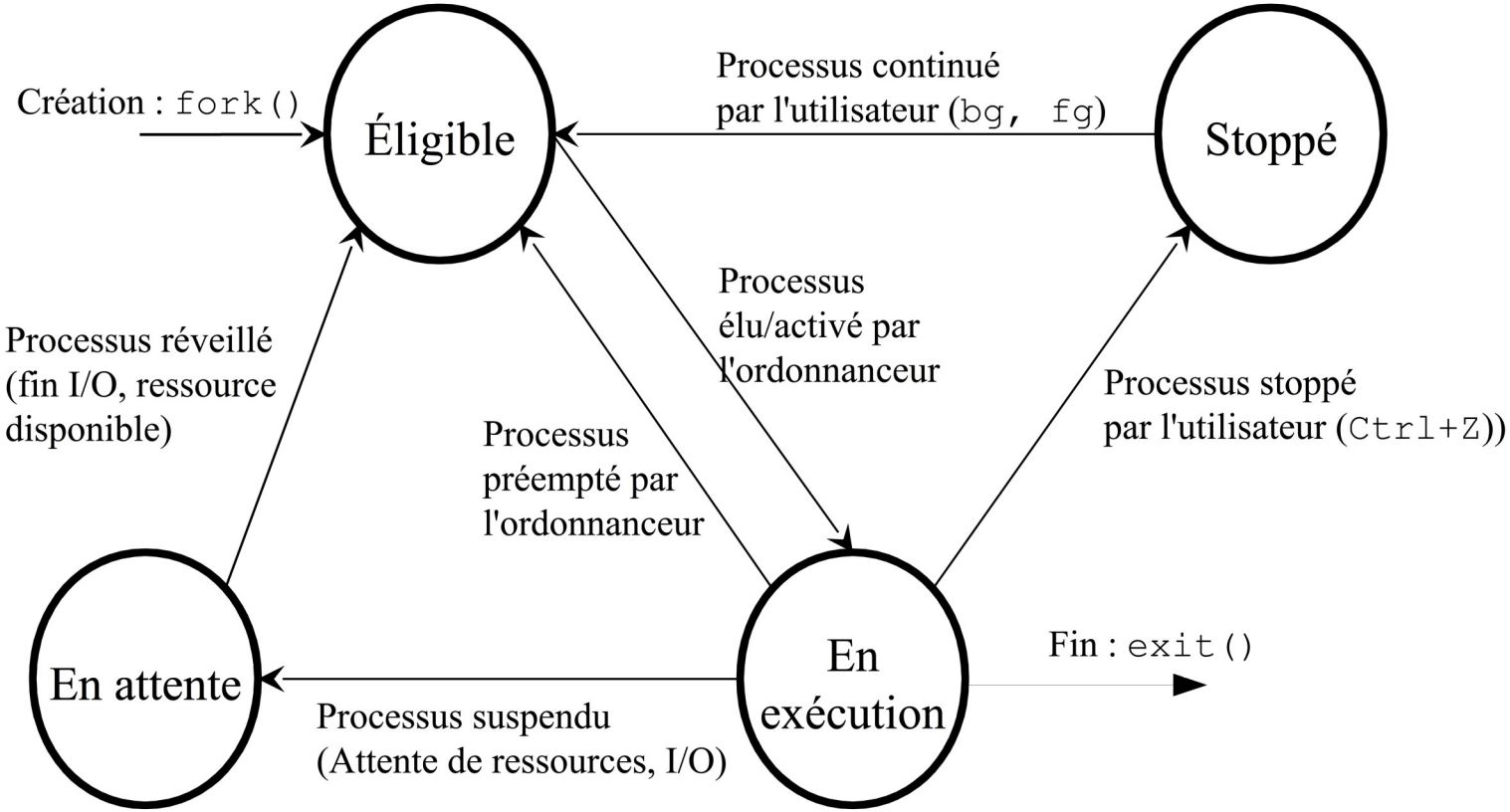
4. **Restauration** du contexte d'exécution du processus élu. L'opération restaure la valeur des registres processeur sauvés en **2**.

5. **Activation** du processus élu

Comment le processeur revient-t-il dans la boucle une fois le changement de contexte effectué ?

Tout se passe comme si le processus préalablement interrompu n'avait pas cessé de s'exécuter.

Etats d'un processus UNIX



On peut voir l'état courant d'un processus avec la commande `ps -l` colonne « `S` » (State)

`R` : éligible ou en Exécution
`D` ou `S` : En attente
`T` : Stoppé

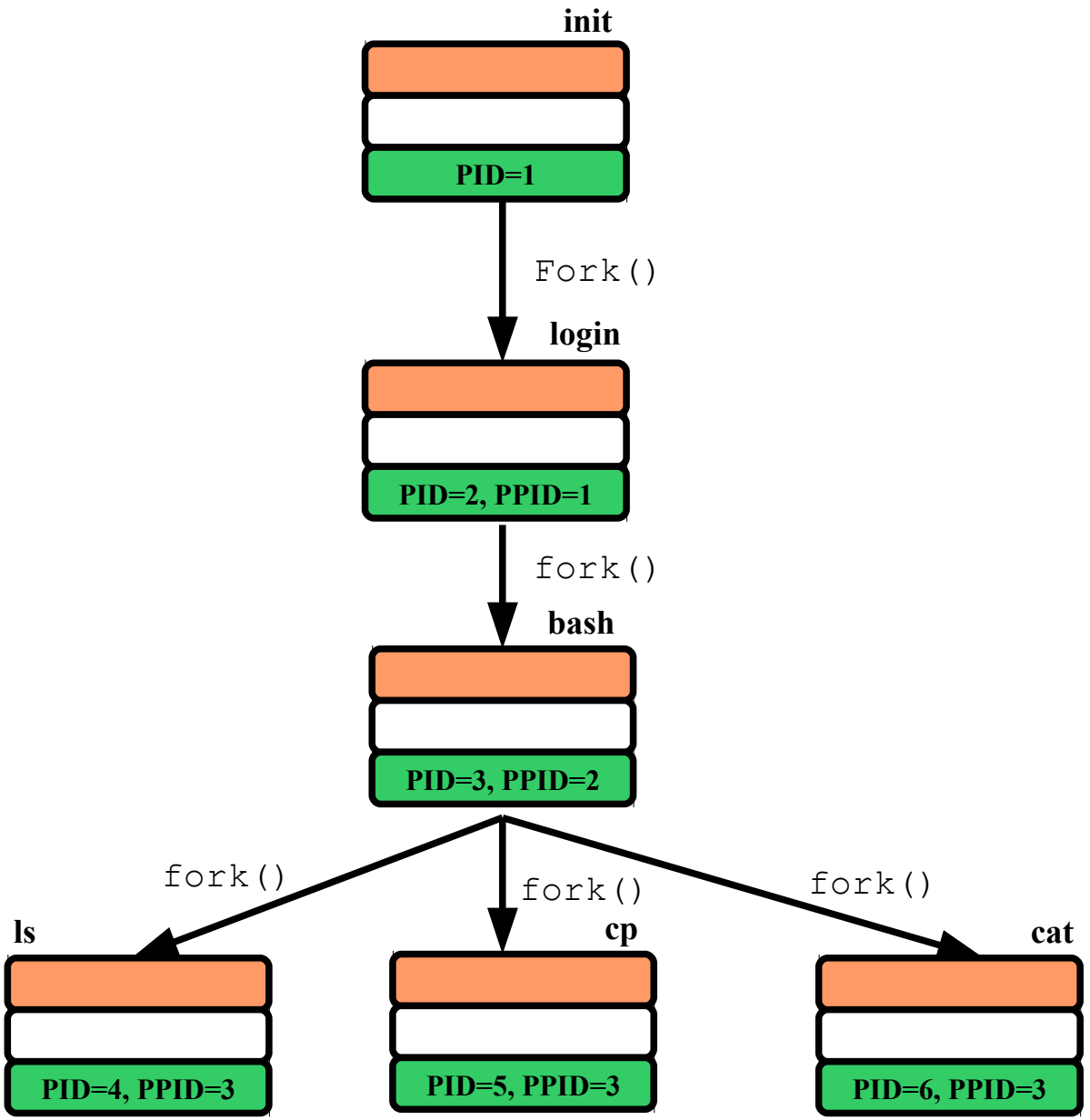
Arborescence de processus

À part pour tout le tout premier (init), un processus a un seul processus parent depuis lequel il a été instancié.

Un processus peut instancier un ou plusieurs processus à l'aide de l'appel système `fork()`

Conséquence : la structure d'organisation des processus est une arborescence dont la racine est le processus init. Les fils de init correspondent aux processus instanciés depuis init par `fork()`. Ces fils peuvent eux-même instancier d'autres processus et ainsi de suite.

Arborescence de processus : exemple



Contenu

Introduction et rappels

Gestion des processus sous Unix

Appels système pour la gestion des processus

Communications inter-processus : les tubes

Appels système pour la gestion des tubes

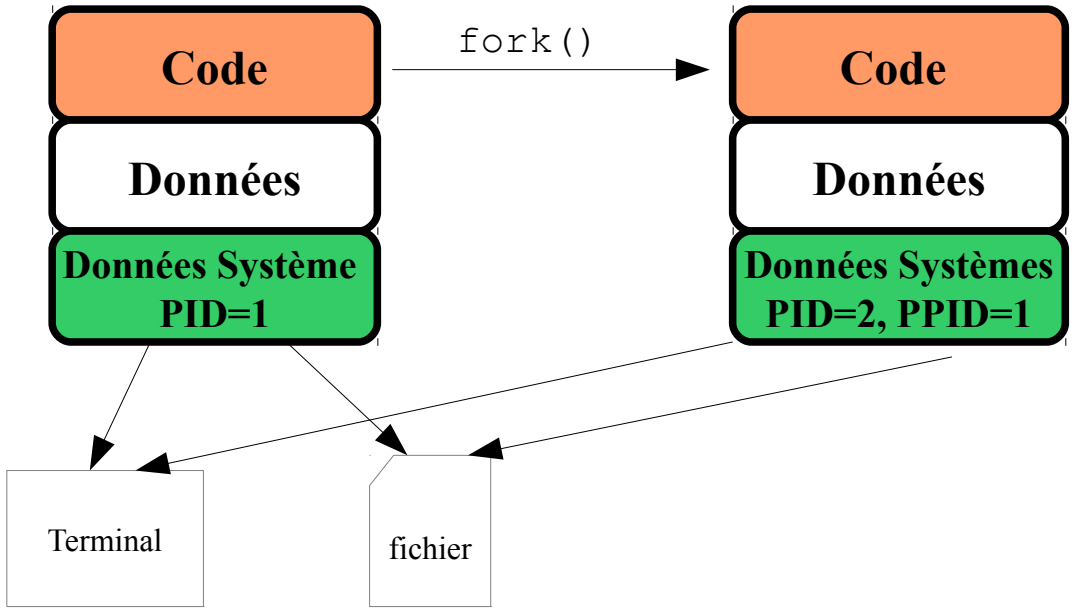
fork()

`fork()` duplique l'ensemble d'un processus qui en fait l'appel :

- Code
- Données utilisateur
- Données systèmes
  - Descripteurs de fichiers
  - Terminal d'exécution
  - UID, GID

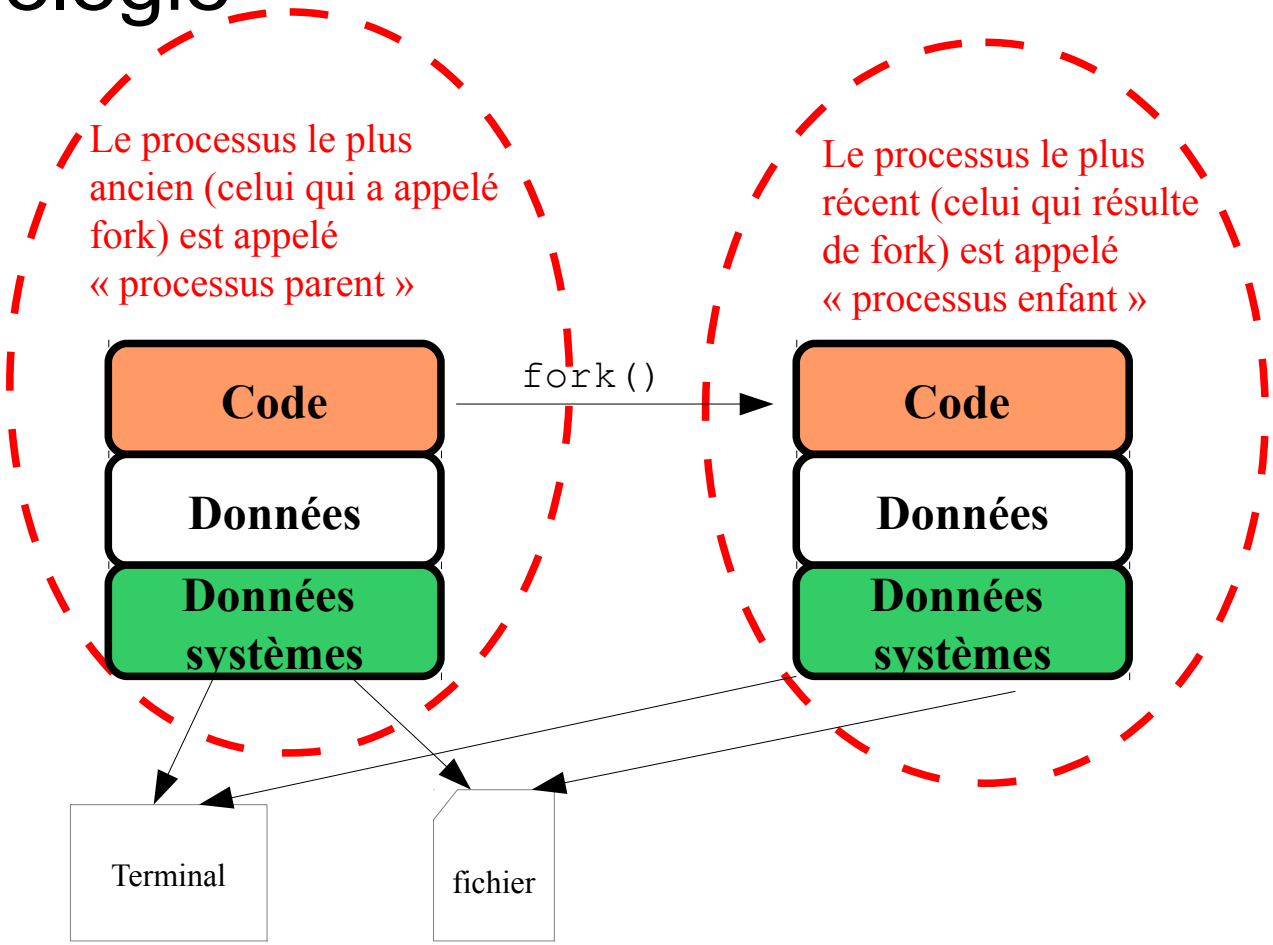
Mais :

- Une nouvelle valeur de PID est allouée.
  - Allocation unique et séquentielle sur l'ensemble du système.
- PPID est mis à jour
- Terminal et fichiers ouverts sont partagés



fork() - Suite

Terminologie



L'appel système fork()

```
pid_t p = fork() ;
```

Si l'appel réussit :

- Un nouveau processus est créé.
- Pour différencier le processus parent du processus enfant :
  - Renvoie 0 dans le processus enfant
  - Renvoie le PID du processus enfant dans le processus parent.
- Si on veut connaître le PID du processus parent :
  - voir l'appel système `getppid()`

Si l'appel échoue :

- Renvoie une valeur négative

Fork : exécution

Processus parent

```
int i=3, c_pid=-1 ;
c_pid=fork() ;
if(c_pid==0)
    printf("enfant\n") ;
else if (c_pid > 0)
    printf("parent\n") ;
else
    perror("fork") ;
```

```
i=3
c_pid=1313
```

PID=1212

Processus enfant

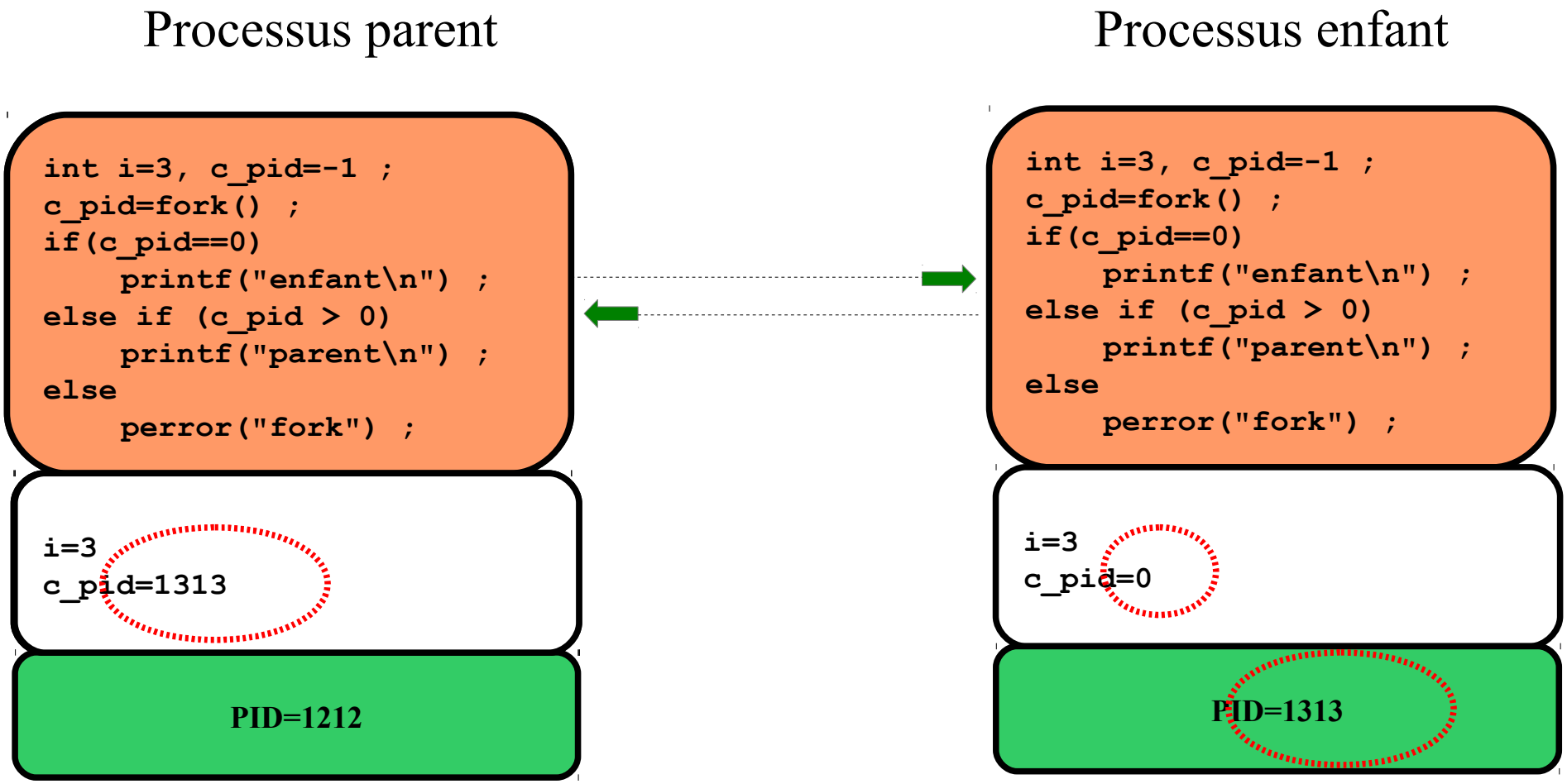
```
int i=3, c_pid=-1 ;
c_pid=fork() ;
if(c_pid==0)
    printf("enfant\n") ;
else if (c_pid > 0)
    printf("parent\n") ;
else
    perror("fork") ;
```

```
i=3
c_pid=0
```

PID=1313

Après `fork()`, les deux processus restent identiques,
mis à part la valeur de retour de `fork()`

Fork : exécution



Parce que les données sont différentes, l'exécution du programme diffère aussi : le comportement du parent et de l'enfant peuvent diverger

L'appel système execve()

```
execve(char *prog,char *argv[],char *envp[]) ;
```

Remplace le processus appelant par `prog`

Ne retourne jamais sauf en cas d'erreur

`envp` est un tableau de chaînes de caractères du type env=valeur définissant les variables d'environnement à passer au programme.
- dernier élément à NULL.

`argv` est passé à la fonction "main" du programme.
- Il faut au minimum argv[0]
- dernier élélment à NULL.

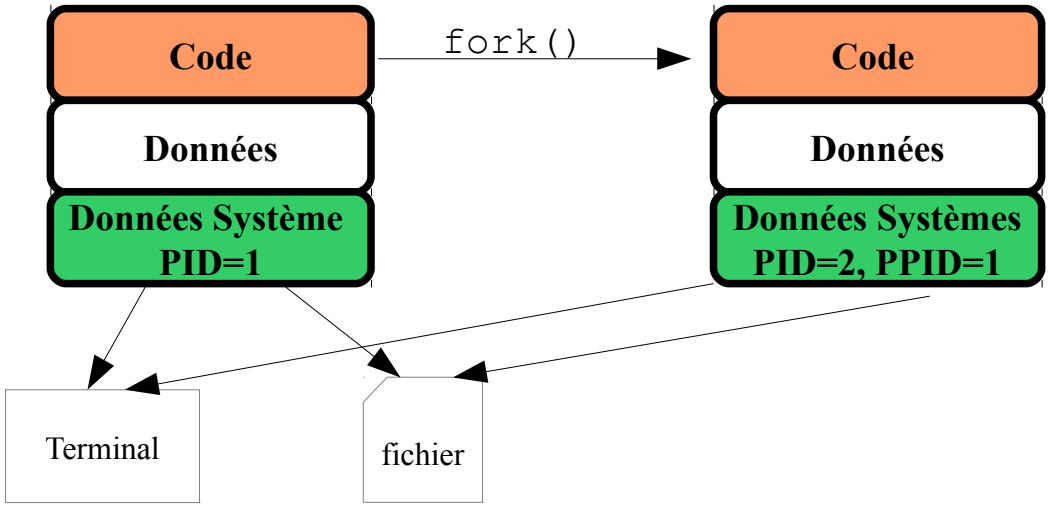
Nouveau processus :
- Code : remplacé par `prog`
- Données utilisateurs : réinitialisé
- Données systèmes : conservées en partie (descripteursde fichiers)

Plusieurs variantes existent dans la libc : `execl`, `execvp`, ...

Execve versus fork

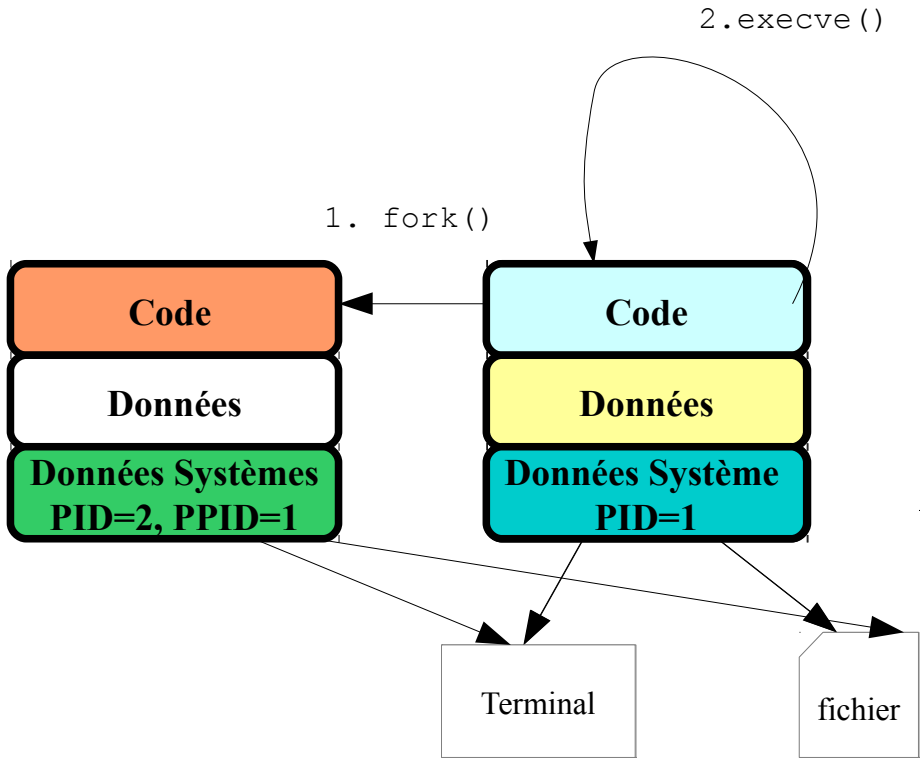
`fork()` duplique un processus

- Les descripteurs de fichiers sont <u>partagés</u>



`execve()` remplace un processus

- Les descripteurs de fichiers sont <u>conservés</u> et donc utilisables entre deux processus qui ne contiennent **pas** le même programme.



Mort d'un processus créé via fork()

Lorsqu'un processus parent se termine
- Les processus enfants deviennent orphelins !
- Le processus init les "adopte"

Lorsqu'un processus enfant se termine :
- Le parent ou init est informé par un signal système.
- Le processus enfant est détruit en partie.
- La partie résiduelle reste jusqu'à ce que le parent « récolte » le résultat de ses enfants
  - En utilisant l'appel système `wait()`
- Tant que le parent n'a pas récolté les résultats, les processus enfants sont dans l'état `Zombie`
  - La commande `ps` indique "exiting" ou <defunct>

L'état Zombie est utilisé pour permettre au parent de savoir quel processus s'est terminé.

L'appel système exit()

```
void exit(int status)
```

Termine l'exécution d'un processus

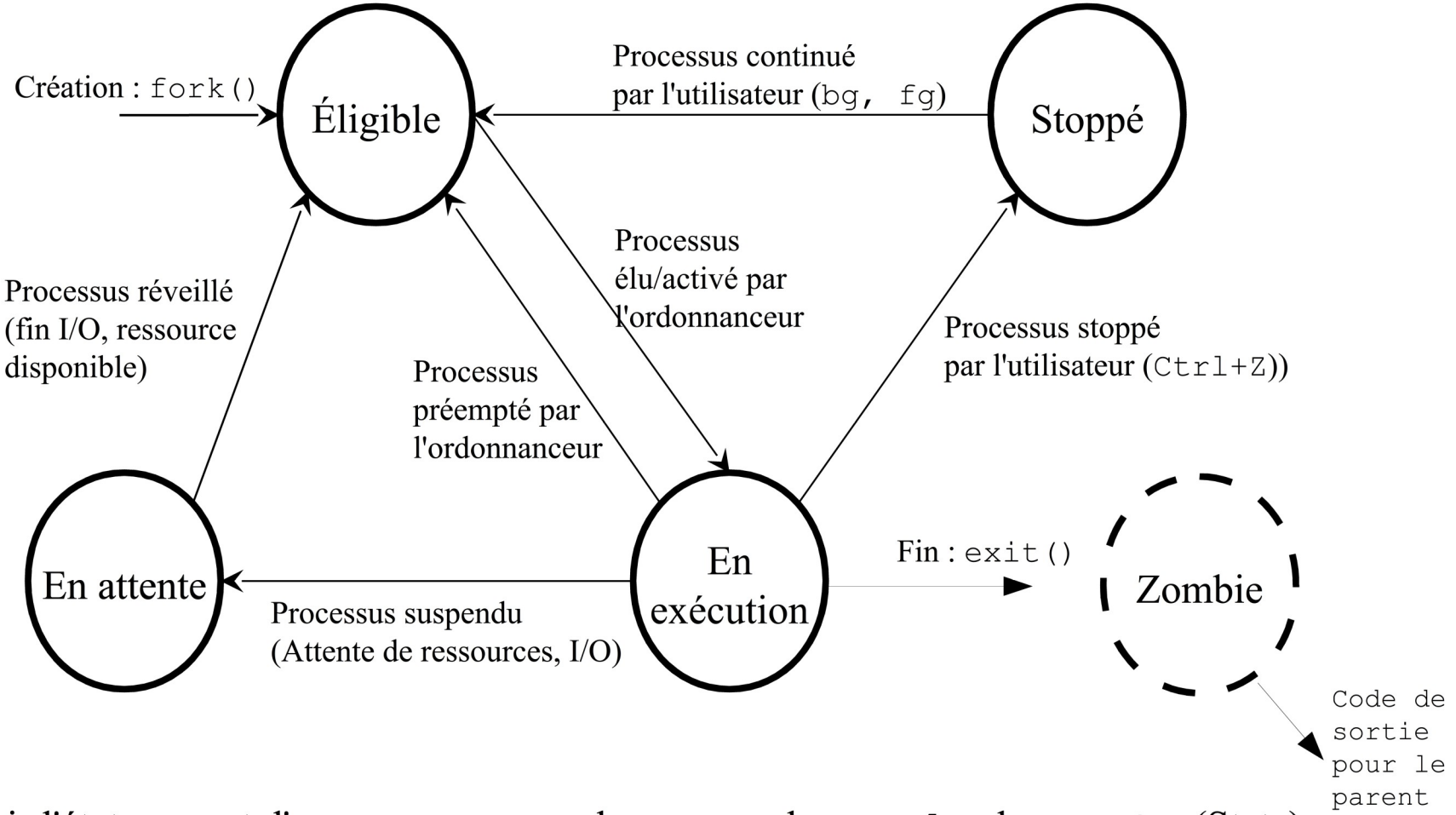
Status est un paramètre transmis au processus parent

L'appel `exit()` fait passer le processus à l'état zombie

- Il restera dans cet état jusqu'à ce que son père ait lu status. A ce moment le processus disparaît du système .

Si un processus parent termine avant ses processus enfants, ils sont adoptés par le processus init

Etats d'un processus UNIX : l'état Z



On peut voir l'état courant d'un processus avec la commande `ps -l` colonne « `S` » (State)

`R` : éligible ou en Exécution
`D` ou `S` : En attente
`T` : Stoppé
`Z` : Zombie

L'appel système wait()

```
pid_t = wait (int *status) ;
```

L'appel est bloquant jusqu'à ce que le processus fils ait terminé

Renvoie le `pid` et le `code de sortie` du processus fils terminé

Si un processus parent n'appele pas `wait()`

- Ses fils restent zombies pour toujours
- Les ressources système risque de s'épuiser

L'appel système wait() : Exemple

```
int run_prog(charg *prog, char *argv[])
{
   int status ;
   int pid = fork() ;
   if (pid < 0 ) return -1 ;
   if (pid == 0) execvp (prog, argv) ;
   if (wait (&status)!= pid ) return -1 ;
   return 0 ;
}
```

Entrées/sorties et fork()

Entrées/sorties système

- Descripteurs de fichiers partagé
  - L'écriture et la lecture sur ces descripteurs est partagée par l'ordonnanceur du système
  - Le premier processus ordonnancé au moment de la lecture/l'écriture lira ou écrira les données => Pas prévisible sans contrôle d'accès par le processus lui-même.
  - La fermeture est reportée jusqu'à ce que tous les processus ferment le descripteur
    - Toujours fermer tous les descripteurs de fichiers non utilisés

Entrées/sorties par la libc

- Bufferisé
  - Buffer dupliqué après `fork()`
  - Le buffer peut-être flushé après `fork()`
    - Affichage double !
  - `Stderr` non bufferisé.

Contenu

Introduction et rappels

Gestion des processus sous Unix

Appels système pour la gestion des processus

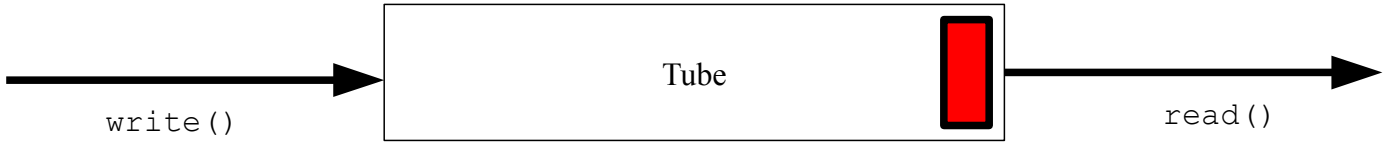
Communications inter-processus : les tubes

Appels système pour la gestion des tubes

Les tubes

Les tubes classiques permettent aux processus de communiquer des données selon les contraintes suivantes :

- Seulement de manière unidirectionnelle et en FIFO: toute donnée écrite dans le tube est retirée du tube par le système à la première lecture

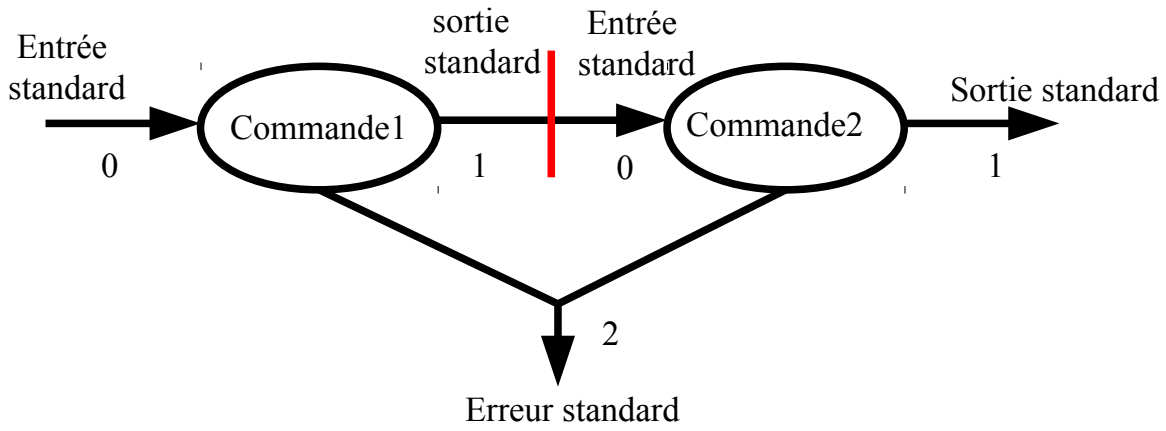


- Ne sont pas présents sur le système de fichier (anonymes par défaut => pathless).
- Ne peuvent contenir qu'un nombre limité d'octets (ordre de grandeur : 4Kb, max 4Mb)
- Ont une de vie limitée à celle du processus les ayant créés.
- Seuls les processus ayant un ancêtre commun ayant crée un tube peuvent communiquer (ils héritent du tube)

Introduction : les tubes et le shell

Le shell utilise les tubes pour relier la sortie standard d'une commande à l'entrée standard d'une autre :

Commande1 | commande2



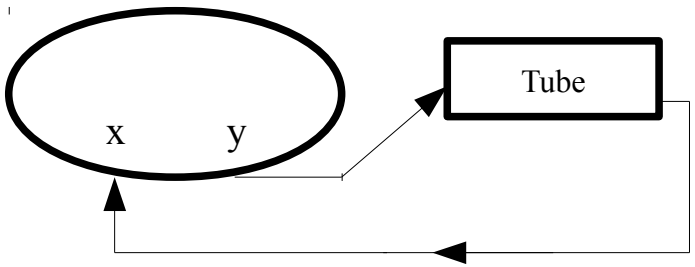
Commande1 et Commande2 s'exécutent en même temps

Dans cet usage particulier le shell n'utilise qu'un seul tube entre les processus, en fait on peut en avoir un nombre arbitraire

Le descripteur de fichier de l'erreur standard est partagé

Les tubes : principe et utilisation

Au départ un tube est toujours crée au sein d'un seul processus par un appel système : `pipe()`

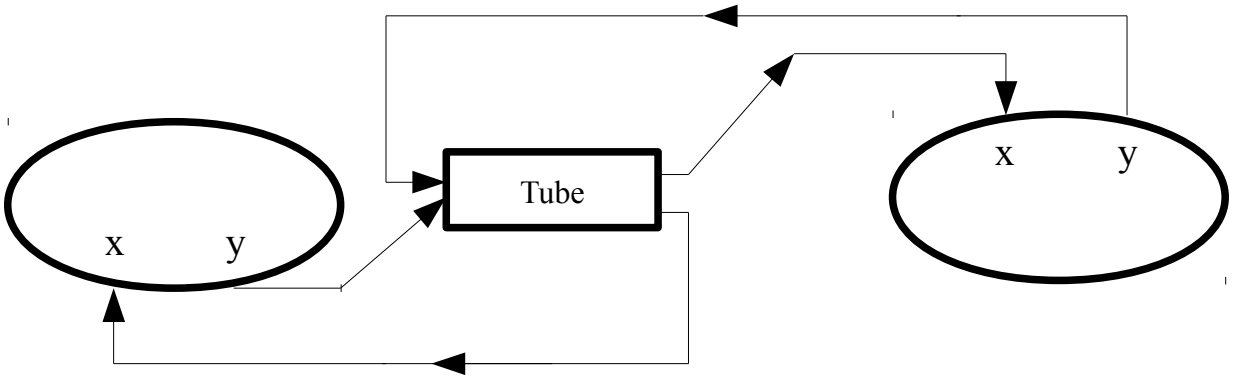


Le processus ne peut que communiquer vers lui-même et que dans un seul sens.

Ce qui est écrit dans le descripteur de fichier `y` (`write()`) peut être lu dans le descripteur de fichier `x` (`read()`): pas très utile !

Les tubes : principe et utilisation

Heureusement, `fork()` permet de faire mieux que ça !

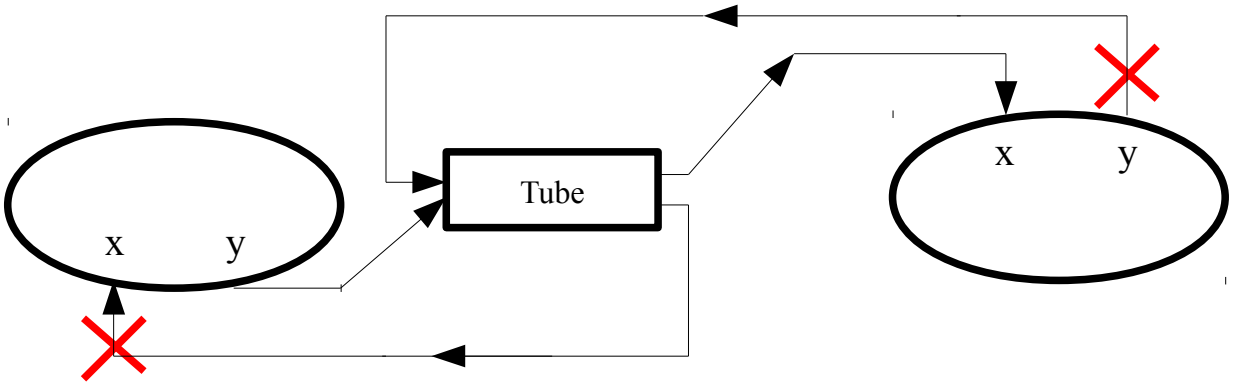


Parce qu'un processus enfant hérite des descripteurs de fichiers de ces ancêtres (s'ils n'ont pas été fermés) :

- Le parent et l'enfant peuvent tous les deux écrire dans l'entrée du tube
- Le parent et l'enfant peuvent tous les deux lire la sortie du pipe
- Mais il n'est pas possible de prédire qui le fera en premier !!! [sauf si on utilise des mécanismes pour contrôler l'accès partagé au tube (verroux, mutex, etc)]

Les tubes : principe et utilisation

Pour qu'un tube soit utilisable entre deux processus :



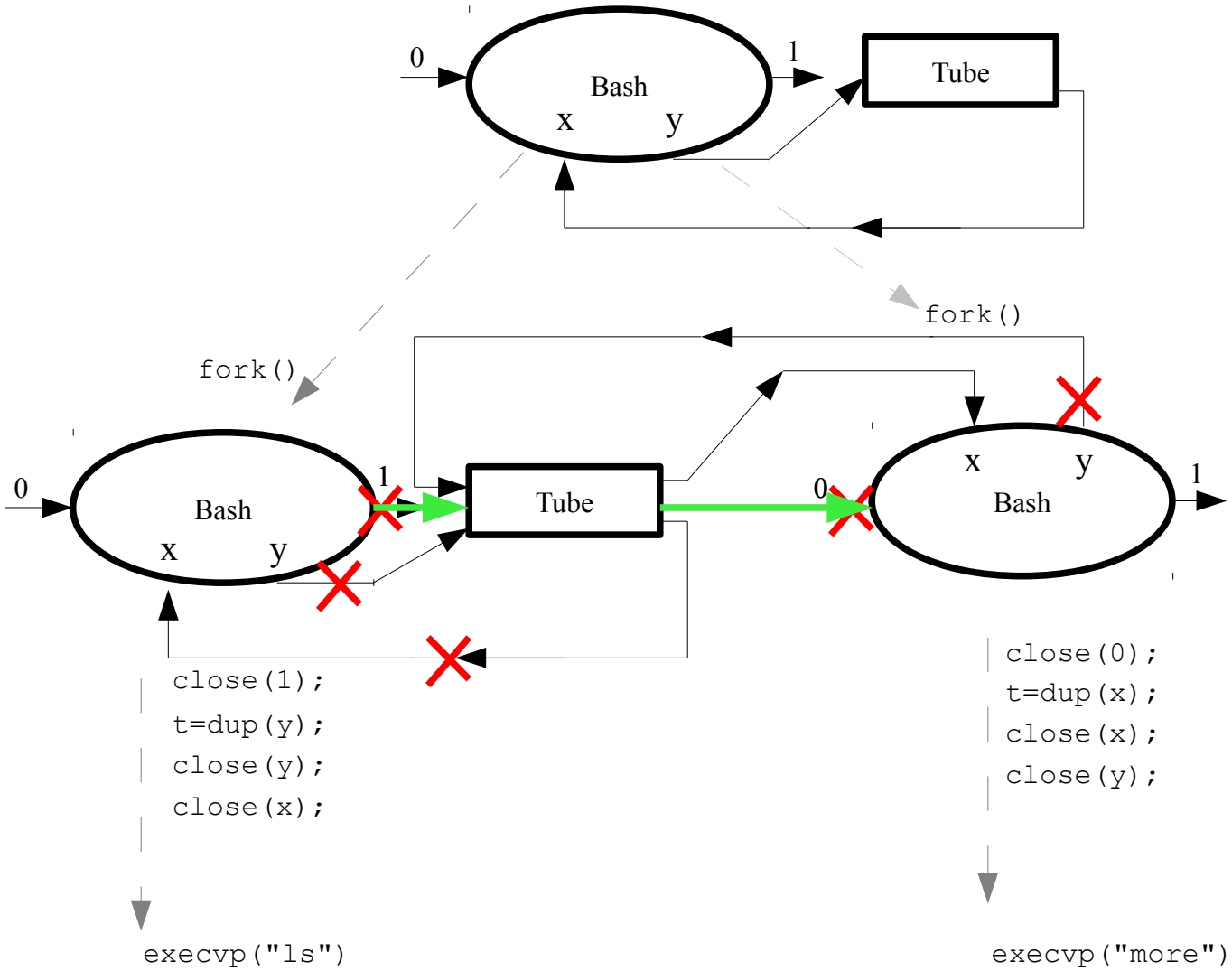
Il faut fermer l'accès en lecture du processus émetteur

- Appel système `close()`

Il faut fermer l'accès en écriture du processus récepteur

Une communication prévisible devient alors possible du processus émetteur vers le processus récepteur : tout ce qui est écrit dans `y` peut être lu dans `x`

Les tubes : Pipes et processus du shell



Contenu

Introduction et rappels

Gestion des processus sous Unix

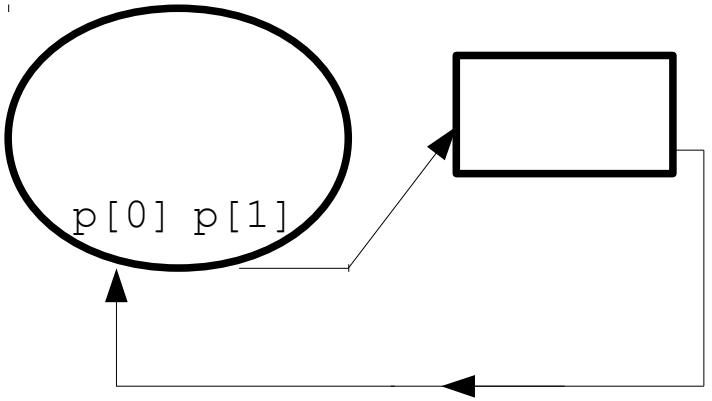
Appels système pour la gestion des processus

Communications inter-processus : les pipes

Appels système pour la gestion des pipes

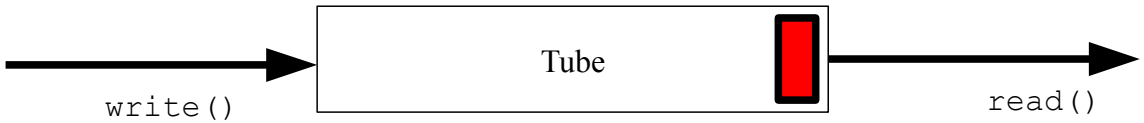
L'appel système pipe()

```
int p[2];
int res = pipe(p);
```



Si l'appel réussit :

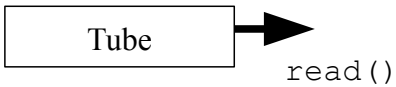
- un nouveau tube est crée.
- renvoie une paire de descripteurs de fichiers dans `p`.
- Cette paire est connectée de `p[1]` vers `p[0]`.
- Toute donnée écrite dans `p[1]` apparait dans `p[0]` en lecture



L'appel système pipe()

Remarques importantes :

- Lorsque le tube est plein 

l'appel à `write()` est bloquant, dès le premier appel
- La capacité d'un tube varie d'un système à l'autre et d'une version à l'autre
- Lorsque le tube est vide 

l'appel à `read()` est bloquant.
- Le signal `SIGPIPE` est envoyé à l'émetteur lorsque aucun processus ne peut lire `p[0]`: évite que `write` ne bloque indéfiniment !
- Lorsque tous les processus ont fermé l'accés en écriture `p[1]`, le prochain appel à `read` renvoie `0` (fin de fichier).