Processus, partie 1

LIFSE - Systèmes d'exploitation

L. Gonnord, N. Louvet, G. Pichon, M. Moy, F. Zara

8 février 2024

Introduction

On travaille sur un ordinateur avec un seul processeur séquentiel : à chaque cycle d'instruction, le processeur exécute une et une seule instruction.

Cet ordinateur a une mémoire de masse et une mémoire vive.
Un programme est une suite d'instructions en langage machine.

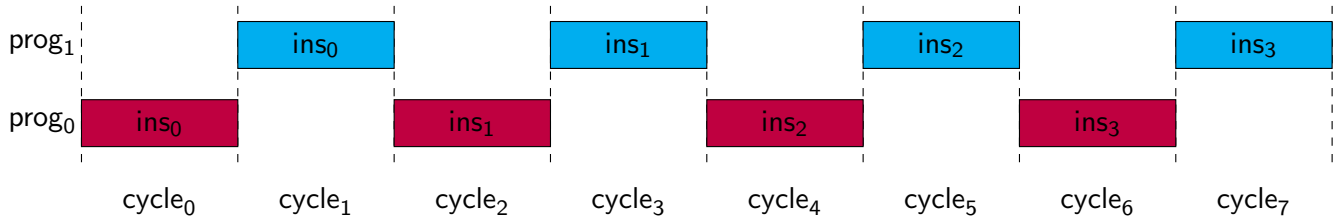
Avant le début de son exécution, un programme est stocké dans la mémoire de masse ; lors de son lancement, il est chargé dans la mémoire vive, puis son exécution commence, instruction par instruction.

Question : avec ces hypothèses, on ne peut exécuter qu'un seul programme à la fois sur notre ordinateur ; comment permettre l'exécution simultanée de plusieurs programmes ?

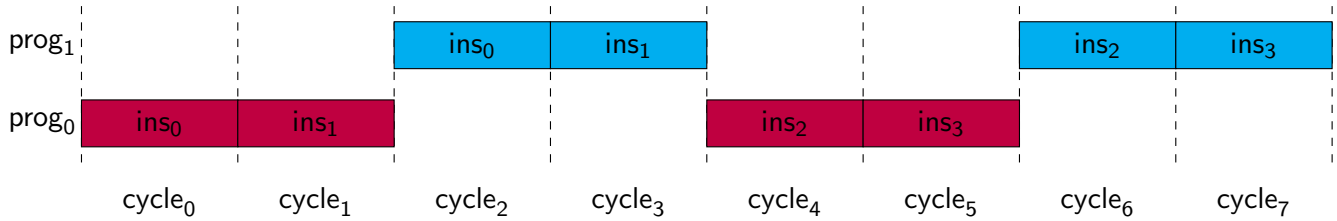
Supposons que l'on souhaite exécuter deux programmes simultanément :

- $\texttt{prog}_1$ avec les instructions $\texttt{ins}_0$, $\texttt{ins}_1$, $\texttt{ins}_2$, ...
- $\texttt{prog}_0$ avec les instructions $\texttt{ins}_0$, $\texttt{ins}_1$, $\texttt{ins}_2$, ...

On peut envisager différentes exécutions des deux programmes, par ex. :

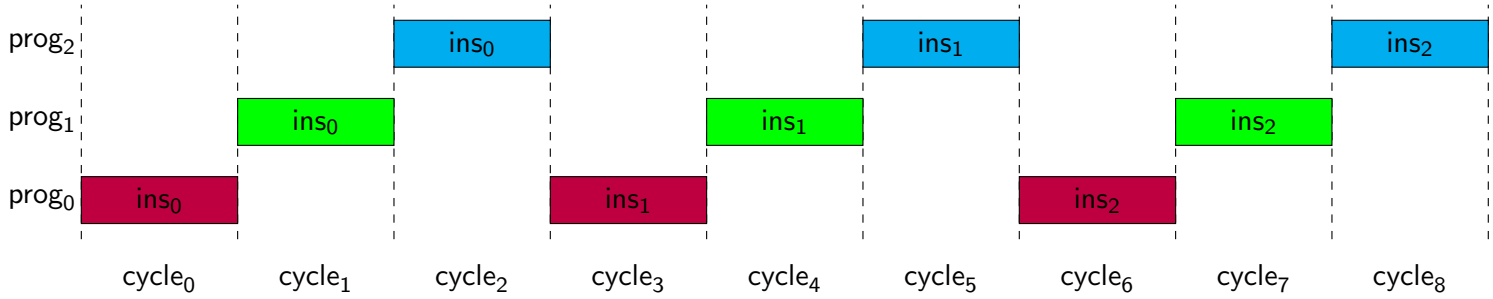


Ou encore :



On fait progresser *en même temps* l'exécution des deux programmes.

Bien-entendu, on peut exécuter ainsi plus de deux programmes :



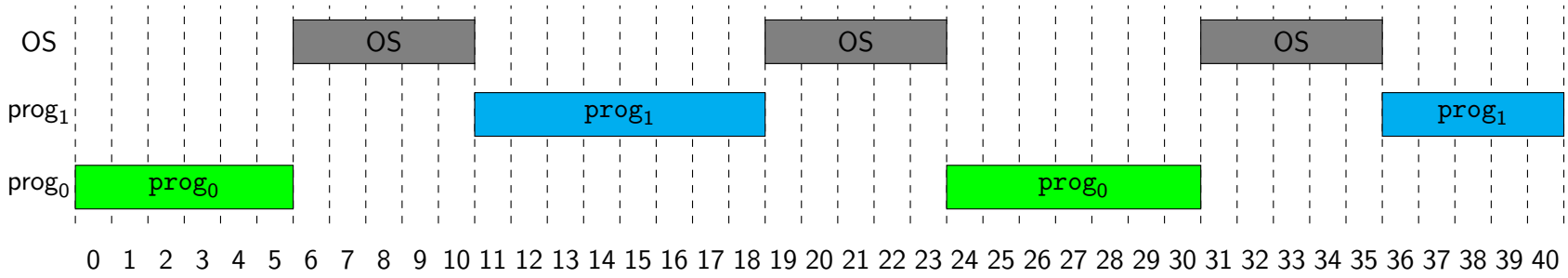
Si la durée du cycle d'instruction est suffisamment petite, l'utilisateur aura l'impression que les deux programmes s'exécutent en même temps.

Problème : ce mode d'exécution ne peut être gérée que par le processeur ; il faudrait avoir dans le processeur, pour chaque programme :

- un compteur de programme spécifique,
- une version spécifique de chaque registre.

$\hookrightarrow$ impossible pour un nombre de programmes un peu grand !

L'idée reste de faire progresser l'exécution de chaque programme, pour donner l'impression qu'ils s'exécutent en même temps.

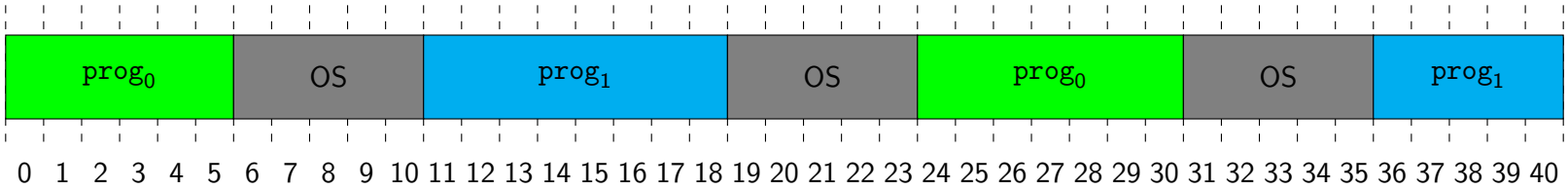


On suppose que le processeur dispose de mécanismes permettant de

- sauvegarder l'état d'exécution d'un programme en mémoire,
- restaurer l'état d'exécution d'un programme depuis la mémoire,
- passer de l'exécution d'un programme à celle du système,
- inversement, de passer du système à l'exécution d'un programme.

Le système d'exploitation se charge d'utiliser ces mécanismes pour partager le temps d'exécution sur le processeur entre chaque programme.

Pour simplifier, on se contente de représenter quel programme est exécuté à chaque cycle d'instruction sur le processeur :



Important : $\texttt{prog}_0$ et $\texttt{prog}_1$ peuvent être l'exécution d'un même code, provenant du même programme exécutable, même si, à chaque cycle, ils en sont à des stades différents de leur exécution.
exemple : il a déjà dû vous arriver de lancer plusieurs fois `gedit`...

Il faut donc pouvoir distinguer l'état d'exécution de chaque programme lancé sur l'ordinateur ; on introduit pour cela la notion de processus.

Notion de processus

Système d'exploitation multitâche

C'est un système qui permet d'exécuter, de façon apparemment simultanée, plusieurs programmes.

Rappel du problème :

- un processeur est prévu pour exécuter un seul programme à la fois,
- on veut exécuter plusieurs programmes sur un seul processeur,
- l'exécution d'un programme ne doit pas perturber celle des autres.

Exécution d'un programme

- Le système doit charger le programme en mémoire,
- trouver son point d'entrée (fonction `main()`),
- suivre son déroulement (pointeur d'instruction),
- à la fin du programme, libérer les ressources qu'il occupait.

Définition (Processus)

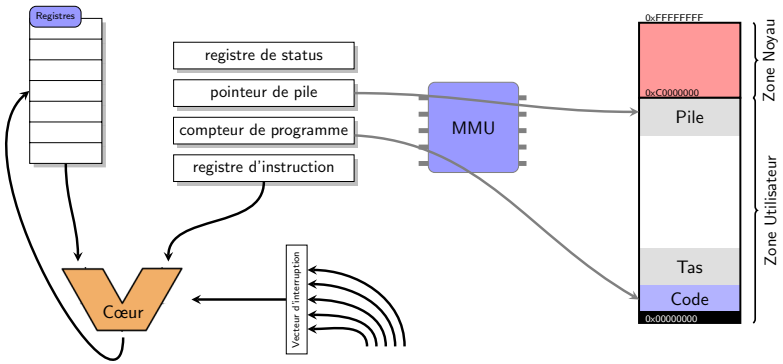
Un processus est un programme en cours d'exécution. Il est géré par le système d'exploitation comme une structure de donnée qui contient toutes les informations nécessaires afin de suivre le déroulement du programme, d'en stopper ou d'en reprendre l'exécution.

Cette structure de données doit permettre :

- d'identifier le programme dont elle est une instance ;
- d'isoler ce programme de tous les autres ;
- d'accéder aux fichiers et aux ressources matérielles ;
- contrôler l'accès aux ressources du système ;
- de libérer les ressources (fermer les fichiers, libérer la mémoire, *etc*).

Dans le processeur, à un instant donné, un processus est décrit par :

- les valeurs contenues dans les registres : compteur de programme, registres généraux, registre de statut, pointeur de pile...
- l'état de la zone mémoire qui lui a été attribuée par le système.



Un processus n'accède jamais directement aux adresses de la mémoire physique, mais à une mémoire virtuelle.

Chaque processus dispose ainsi de son propre espace mémoire, et ne peut pas interférer avec les autres processus.

Chaque processus est représenté par un contexte, qui est une structure de données qui regroupe (au moins) les informations précédentes :

contexte d'un processus
identifiant unique du processus
compteur de programme
registre d'instruction
pointeur de pile
contenu des registres généraux
...

Le contexte est un enregistrement de l'état courant du processus. L'idée est que le système peut :

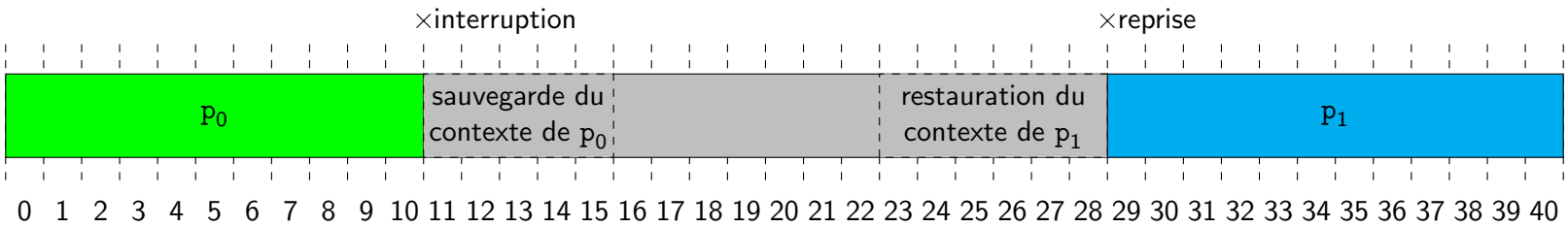
- à partir du contexte d'un processus, faire reprendre son exécution,
- sauvegarder le contexte d'un processus, et arrêter son exécution.

Ordonnancement des processus

Le système permet le passage de l'exécution d'un processus à celle d'un autre. Lors du passage de l'exécution d'un processus p_0 à un autre p_1 :

- l'exécution de p_0 est *interrompue*,
- le système reprend la main et sauvegarde le contexte de p_0,
- le système restaure le contexte p_1 dans le processeur,
- le processus p_1 *reprend* son exécution.

La même chose représentée de façon graphique :

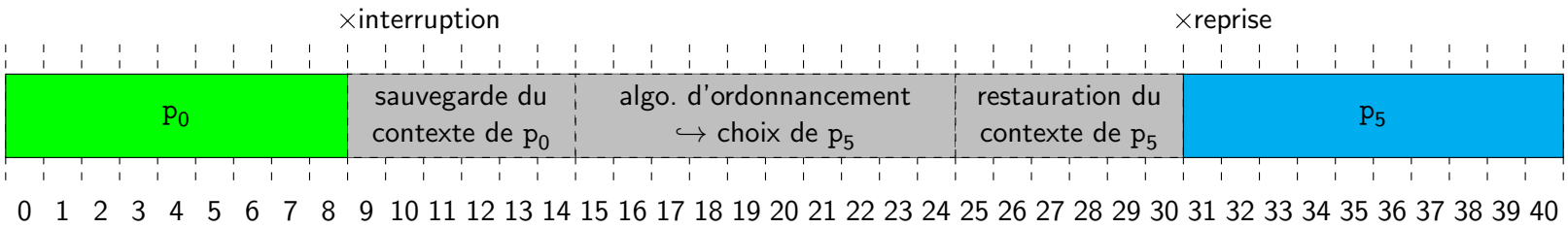


On parle de commutation de contexte pour désigner :

- soit la sauvegarde du contexte courant vers la mémoire (*interruption*),
- soit la restauration d'un contexte depuis la mémoire (*reprise*).

Le système doit permettre l'accès au processeur à de nombreux processus pour faire progresser leurs exécutions (selon certaines contraintes).

À chaque interruption d'un processus, le système exécute un algorithme ordonnancement pour choisir le prochain processus à accéder au processeur.

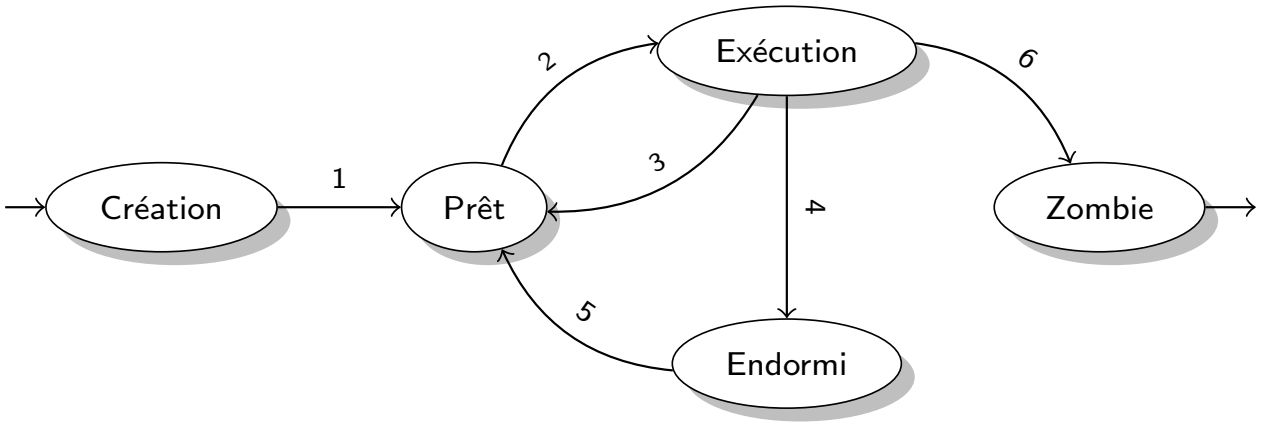


Cet algorithme se base sur différentes files d'attentes de processus :

- ceux qui peuvent s'exécuter,
- ceux qui sont en mémoire,
- ceux qui sont bloqués car ils demandent une ressource occupée,
- ...

On parle d'état du processus.

Exemple : évolutions possibles de l'état d'un processus dans l'ordonnanceur.



1 - le processus est prêt, il est pris en compte par l'ordonnanceur
2 - l'ordonnanceur choisit le processus pour être exécuter sur le processeur
3 - l'ordonnanceur choisit un autre processus pour être exécuter
4 - le processus décide de s'endormir pour une certaine durée (avec `sleep()`)
5 - l'ordonnanceur repasse la processus à l'état « prêt » à la fin de cette durée
6 - le processus « décide » de se terminer (avec `exit()` par exemple)

Rappel. Pour voir l'état des processus : `ps -l` ou `top` (`q` pour quitter).

Interruptions

Le système est un programme qui s'exécute sur le processeur :
comment peut-il reprendre la main sur un processus en cours d'exécution ?

Tous les processeurs disposent d'un mécanisme d'interruption. Une interruption est une requête adressée au processeur pour qu'il interrompe l'exécution du code courant.

Il existe différents types d'interruptions.

- interruptions matérielles : changement d'état d'un périphérique (disque, clavier, ...), échéance d'une alarme, ...
- interruptions logicielles : provoquées par des certaines instructions, ou par le comportement anormal d'un processus (segfault).

Chaque interruption est identifiée par un code d'interruption numérique.

Le processeur dispose d'un vecteur d'interruptions qui associe à chaque code d'interruption l'adresse d'une procédure gestionnaire d'interruption.

Quand le processeur reçoit une interruption :

- il sauvegarde le contexte du processus courant,
- il se met à exécuter la procédure gestionnaire de l'interruption.

Concrètement, le vecteur d'interruptions est un tableau d'adresses `V` : à la réception d'une interruption de code `k`, le processeur sauvegarde le contexte en mémoire vive, puis se met à exécuter le code à l'adresse `V[k]`.

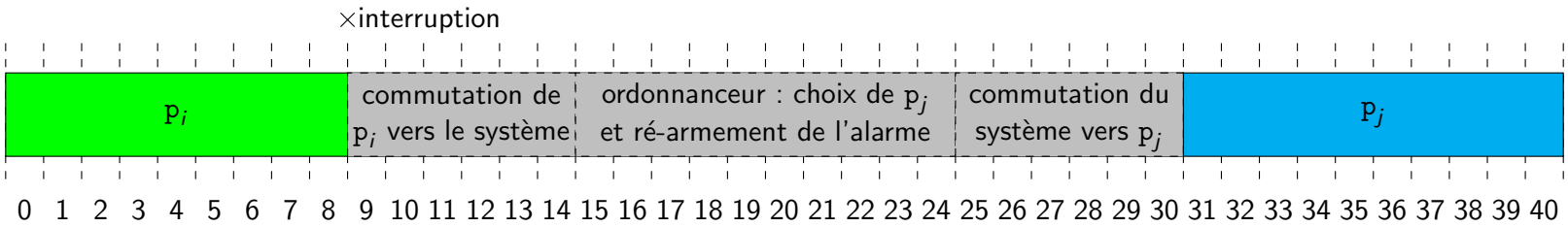
Ce mécanisme permet les commutations d'un processus vers le système.

Les interruptions matérielles sont utilisées quand le système doit reprendre la main sur un processus.

Par exemple, lors du passage d'un processus p_i à un autre p_j, le processeur :

- est interrompu par une alarme matérielle,
- sauvegarde le contexte du processus courant,
- exécute l'ordonnanceur qui choisit d'allouer t ms d'exécution à p_j,
- ré-arme l'alarme pour qu'elle envoie une interruption après t ms,
- restaure le contexte du processus p_j.

Ainsi p_j s'exécutera t ms avant la prochaine interruption.



Les interruptions logicielles sont utilisées quand un processus a besoin de rendre la main au système d'exploitation.

En effet, tout processus utilisateur doit faire appel au système pour avoir accès aux ressoources, par exemple pour lire ou écrire dans un fichier.

Lorsqu'un processus doit passer la main au système :

- le processus provoque une interruption logicielle,
- le processeur lance la procédure de gestion de l'interruption,
- la procédure d'interruption invoque le code adéquat du système.

C'est ce qui se passe lorsque vous faites un appel système,
comme avec `read()` ou `write()` dans un programme C/C++.

Quand vous faites un appel système, vous rendez la main au système : il réalise cet appel, mais il en profite aussi pour effectuer l'ordonnancement.

Observation des processus

Sous Unix, les processus sont organisés de façon hiérarchique :

- chaque processus est identifié de façon unique par un entier : c'est son PID, pour Process Identifier.
- chaque processus peut créer des processus appelés fils.
- tous les processus ont accès à l'identifiant de leur père : c'est leur PPID, pour Parent Process Identifier.

A la racine de la hiérarchique il y a l'ancêtre de tous les processus :

- son PID est égal à 1 ;
- sous GNU/Linux, c'est `init` (souvent remplacé par `systemd`).

Deux commandes utiles pour observer les processus :

- `ps` avec notamment les options `-e` et `-l` ;
- `pstree` pour avoir une vue de l'arborescence des processus.

Exemple :

```
nlouvet:~$ xclock & xcalc
nlouvet:~$ bash
nlouvet:~$ xcalc &
nlouvet:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  5503  4261  0  80   0 -  6317 wait   pts/3    00:00:00 bash
0 S  1000  5555  5503  0  80   0 - 18513 poll_s pts/3    00:00:00 xclock
0 S  1000  5556  5503  0  80   0 - 13683 poll_s pts/3    00:00:00 xcalc
0 S  1000  5558  5503  0  80   0 -  6065 wait   pts/3    00:00:00 bash
0 S  1000  5566  5558  0  80   0 - 13683 poll_s pts/3    00:00:00 xcalc
4 R  1000  5567  5558  0  80   0 -  7558 -      pts/3    00:00:00 ps
nlouvetk:~$ pstree -p 5503
bash(5503)─┬─bash(5558)─┬─pstree(5568)
           │            └─xcalc(5566)
           ├─xcalc(5556)
           └─xclock(5555)
```

On retrouve bien :

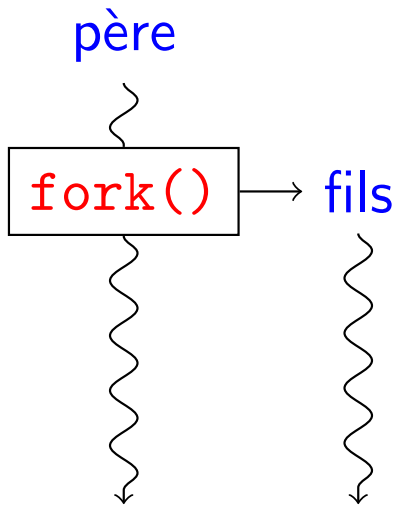
- les informations PID, PPID, état (colonne S)...
- l'organisation hiérarchique des processus

Création de processus

Sous Unix, un nouveau processus est forcément la copie d'un processus existant :

- le processus déjà existant est appelé le père,
- le nouveau processus créé est appelé le fils.

`fork()` est l'appel système utilisé pour créer un nouveau processus :



Au retour de `fork()`, le père et le fils exécutent tous les deux le même code : celui présent juste après l'appel à `fork()` dans le programme !

`pid_t fork(void)`

Que dit le manuel ?

- `fork()` creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following...
- On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, and errno is set appropriately.

Après un appel réussi à `fork()` :

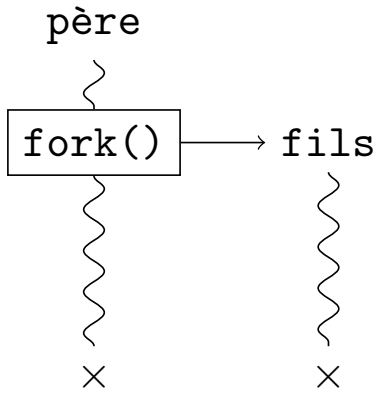
- le fils est une copie du père (même code, même mémoire),
- seule la valeur de retour de `fork()` permet de les distinguer.

Il faut tester la valeur de retour de `fork()` pour différentier leurs exécutions.

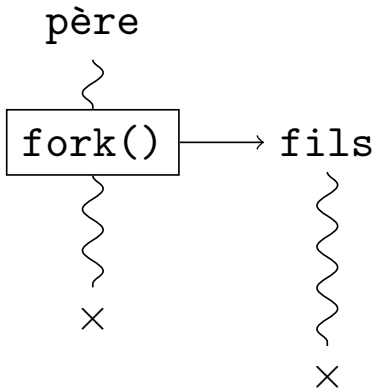
Un exemple :

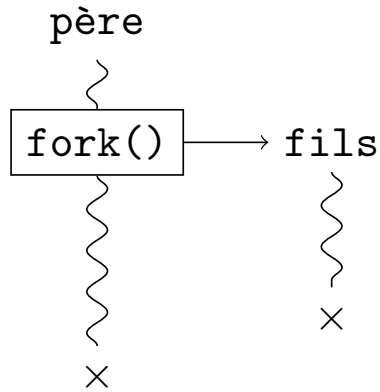
```
int main(void) {
    int ret = fork();

    // le test suivant est exécuté par les deux processus
    if(ret > 0) {
        // exécuté par le père
        cout << "(père) pid : " << getpid() << endl;
        cout << "(père) ppid : " << getppid() << endl;
    }
    else {
        // exécuté par le fils
        cout << "(fils) pid : " << getpid() << endl;
        cout << "(fils) ppid : " << getppid() << endl;
    }

    // executé par les deux processus
    cout << "Fin du processus de pid " << getpid() << endl;

    return 0;
}
```

On peut se représenter l'exécution du programme par le schéma suivant :



Mais il ne faut pas présumer du processus qui se terminera en premier !
On peut être dans l'un des deux cas suivants :





Pour être certain que le père se termine après le fils, il faut qu'il attende sa terminaison, grâce à un mécanisme que l'on va voir juste après.

Fin d'un processus

Dans le cas général, un processus se termine en

- exécutant un `return()` dans sa fonction `main()`,
- ou en faisant appel à la fonction `void exit(int status)`.

Lorsqu'il se termine, un processus passe forcément en état zombie :

- il n'est plus jamais prêt mais ses ressources ne sont pas toutes libérées : la valeur de retour de la fonction `main()` est en attente.
- le père doit lire ce résultat par l'appel `waitpid()` :
  le processus fils disparaît alors de la table des processus.

Un zombie occupe des ressources tant que son père n'a pas fait un appel à `waitpid()` : il ne faut pas laisser le système encombré de zombies.

Si le père se termine avant son fils, le fils est orphelin : il est adopté par `init`, qui se charge d'en libérer les ressources.

`pid_t waitpid(pid_t pid, int *wst, int opt);`

L'appel permet d'attendre qu'un fils se termine :

- `pid` est le PID du fils attendu, ou 0 si l'on attend n'importe lequel ;
- `wst` pour récupérer les infos de retour du fils (NULL pour ignorer) ;
- si `opt` = 0 l'attente est bloquante (le cas important pour nous) ;
- si `opt` = `WNOHANG` l'appel est non-bloquant.

Que dit le manuel ?

In the case of a terminated child, performing a wait allows the system to release the resources associated with the child ; if a wait is not performed, then the terminated child remains in a "zombie" state.

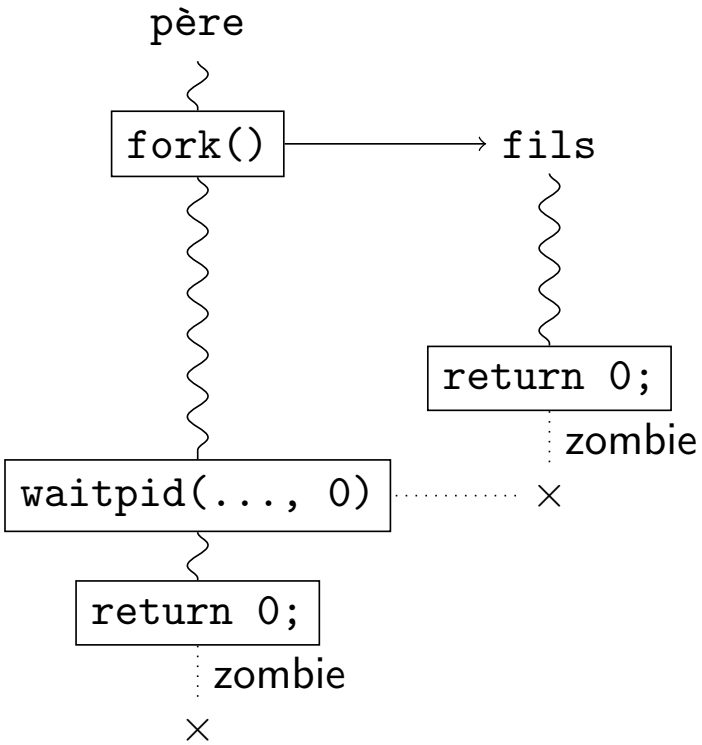
Valeur de retour :

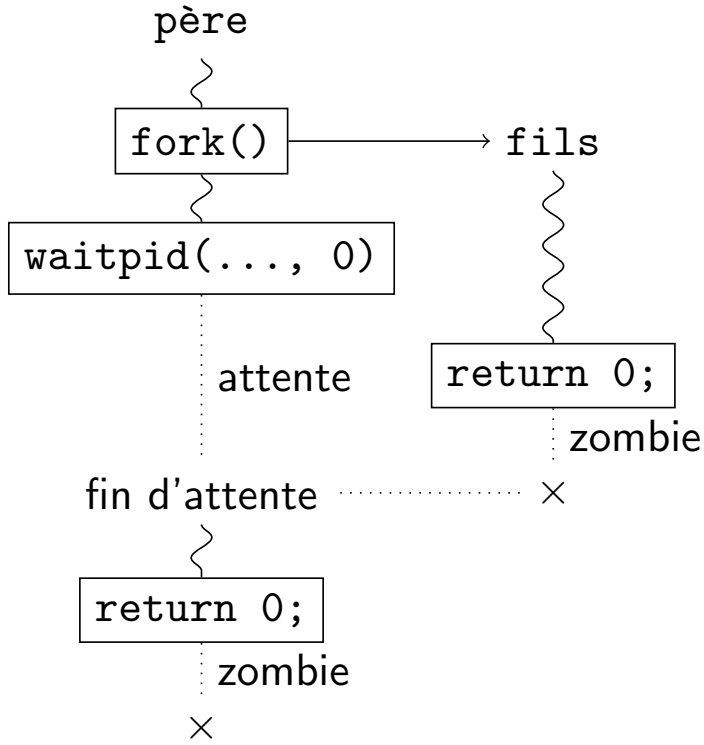
- en cas d'échec, -1 est retourné et `errno` mise à jour ;
- si un fils s'est terminé, le PID de ce fils ;
- si l'appel est non-bloquant et qu'aucun fils n'est terminé, alors 0.

Un exemple :

```
int main(void) {
    int ret = fork();

    if(ret > 0) { // processus père
        cout << "(père) j'attends..." << endl;
        waitpid(ret, NULL, 0); // le père attends son fils
        cout << "(père) pid : " << getpid() << endl;
        cout << "(père) ppid : " << getppid() << endl;
    }
    else { // processus fils
        cout << "(fils) pid : " << getpid() << endl;
        cout << "(fils) ppid : " << getppid() << endl;
    }

    cout << "Fin du processus de pid " << getpid() << endl;

    return 0;
}
```

On peut avoir les deux scénarios suivants, selon que le fils arrive au `return(0)` avant ou après le processus père :





En tout cas, le processus père attend la terminaison du processus fils : le `waitpid(..., 0)` ne retourne pas tant que le fils n'est pas terminé.

Question : quel processus attend la terminaison du père dans cet exemple ?

Recouvrement d'un processus

Comment faire pour, depuis un programme, lancer un autre programme ?

Une seule méthode prévue : le code d'un processus déjà créé doit être recouvert (remplacé) par celui du programme à lancer. On fait appel à l'une des fonctions de la famille `exec()` : le processus appelant est recouvert par la commande en argument d'`exec()`.

Par exemple `int execl(const char *path, const char *argv, ...);` recouvre le processus appelant par le commande dont le chemin est donné par `path`, en lui passant les arguments de qui sont dans `argv`.

Que dit le manuel ? (`man exec`)

The exec() family of functions replaces the current process image with a new process image. The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

Un exemple :

```
int main(void) {
    pid_t ret = fork();
    if(ret == 0) { // processus fils
        execl("/bin/ps", "ps", "-l", NULL);
        // suite du fils jamais executée, sauf en cas d'erreur
        cerr << "(fils) erreur : " << strerror(errno) << endl;
        return 1;
    }
    else { // processus père
        waitpid(ret, NULL, 0);
        cout << "(père) la commande est terminée." << endl;
    }
    return 0;
}
```

Dans cet exemple, le père commence par créer un fils puis :

- le fils est recouvert par la commande `ps -l`,
- le père attend la terminaison de la commande avant de se terminer.

Conclusion

Les processus sont un moyen pour le système de gérer l'exécution simultanée de nombreux programmes :

- chaque processus exécute son propre code,
- les processus ne partagent pas de mémoire entre eux,
- l'ordonnanceur gère l'accès au processeur de chaque processus.

Un seul moyen de créer un nouveau processus : avec la primitive `fork()`. Le processus fils créé est initialement une copie exacte de son père : seule la valeur de retour de `fork()` permet de distinguer le fils du père.

Un processus peut prendre en compte et attendre la terminaison de ses fils avec `waitpid()`. Cela permet une une forme de synchronisation entre processus. De plus, il est important de ne pas laisser proliférer les zombies.

Un seul moyen de lancer un exécutable : en recouvrant un processus existant par cet exécutable, avec l'une des fonctions de la famille `exec()`.