

1 Les processus Unix

1.1 Définition

Un processus est une entité d'exécution dynamique composée :

- un programme
- des données
- un CPU

Un processus est lancé par un autre processus avec un utilisateur associé. Par exemple une commande déclenche un processus si le nom de la commande est trouvé dans le PATH. A un instant, un seul processus est actif (sauf si l'ordinateur est un ordinateur multicoeur).

Le processus actif est changé (il est swappé) un grand nombre de fois par seconde. Un même utilisateur peut lancer plusieurs processus. Chaque processus est identifiable par un numéro, le PID (process Identification).

Par exemple, une des premières choses que fait le système d'exploitation UNIX lorsque le serveur est connecté est d'initier le processus **init** portant le numéro d'identification (pid) 1. Ce processus occupe la même position dans la hiérarchie des processus que la racine (**root**) occupe dans la hiérarchie des fichiers : Il est l'ancêtre de tous les autres processus utilisés par les utilisateurs. Si des terminaux sont attachés au système, **init** initie un processus **getty** pour chacun d'eux (rappelez-vous que UNIX est multi-tâches). Ce dernier processus attend qu'un utilisateur tente de se connecter. Le cas échéant, **getty** initie un processus **login** qui vérifie l'identité de l'utilisateur et la validité de son mot de passe. Si tout se passe bien, **login** effectue quelques tâches et initie un processus **bash** (cela peut être un autre **shell**).

1.2 La commande pstree.

La commande **pstree** affiche les processus sous forme d'arborescence et permet de les visualiser avec leurs liens de parenté. Ainsi, pour tuer une série de processus de la même famille, il suffira d'en découvrir l'ancêtre commun. On peut utiliser l'option **-p**, qui affiche le PID de chaque processus, ainsi que l'option **-u**, laquelle donnera le nom de l'utilisateur ayant lancé le processus. L'arborescence étant généralement longue, il est plus facile d'invoquer **pstree** de cette façon :

```
pstree -up | more
```

pour en avoir une vue d'ensemble.

```
init(1)--NetworkManager(773)
|-acpid(848)
|-anacron(847)---sh(1598)---run-parts(1600)---apt(1606)---sleep(1630)
|-atd(858,daemon)
|-avahi-autoipd(1506,avahi-autoipd)---avahi-autoipd(1507,root)
```

FIGURE 1 – la commande pstree.

On voit que le processus **init** de PID 1 est à l'origine de tous les autres !

1.3 La commande ps.

Cette commande sans argument montrera uniquement les processus dont vous êtes l'initiateur et qui sont rattachés au Terminal que vous utilisez :

```
File Edit View Terminal Help
serveur@ubuntu:~$ ps
  PID TTY          TIME CMD
 9509 pts/0    00:00:01 bash
16667 pts/0    00:00:00 ps
serveur@ubuntu:~$
```

FIGURE 2 – la commande ps.

Les options sont nombreuses. Voici les plus courantes :

- **a** : affiche aussi les processus lancés par les autres utilisateurs ;

- x : affiche aussi les processus n'ayant pas de terminal de contrôle ou un terminal de contrôle différent de celui que vous êtes en train d'utiliser ;
- u : affiche pour chaque processus le nom de l'utilisateur qui l'a lancé et l'heure de son lancement.

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.2	23684	1348	?	Ss	Nov11	0:01	/sbin/init
root	2	0.0	0.0	0	0	?	S	Nov11	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	Nov11	0:00	[migration/0]
root	4	0.0	0.0	0	0	?	S	Nov11	0:01	[ksoftirqd/0]
root	5	0.0	0.0	0	0	?	S	Nov11	0:00	[watchdog/0]
root	6	0.0	0.0	0	0	?	S	Nov11	0:06	[events/0]
root	7	0.0	0.0	0	0	?	S	Nov11	0:00	[cpuset]
root	8	0.0	0.0	0	0	?	S	Nov11	0:00	[khelper]
root	9	0.0	0.0	0	0	?	S	Nov11	0:00	[netns]
root	10	0.0	0.0	0	0	?	S	Nov11	0:00	[async/mgr]
root	11	0.0	0.0	0	0	?	S	Nov11	0:00	[pm]

FIGURE 3 – la commande ps aux.

La commande visualise d'autres attributs des processus :

- la date et heure de lancement :START TIME
- la mémoire utilisée en RAM par le processus : RSS
- la mémoire totale (RAM+swap) par le processus : VSZ
- le terminal utilisé s'il y en a un : TTY
- la quantité de CPU utilisée en % : CPU
- la quantité de mémoire RAM utilisée en % : MEM

1.4 Lancement d'un processus en arrière plan.

Il existe deux types de processus lancés dans un terminal : les processus en **arrière-plan** et le **processus en avant-plan**. Lorsqu'une commande est exécutée en avant-plan (foreground), tout ce que l'on écrit au terminal est envoyé à l'entrée standard (à moins celle-ci ait été redirigée). En conséquent, il est impossible de faire quoi que ce soit d'autre tant que le processus n'est pas terminé. En tout temps, il n'y a qu'un **seul processus en avant-plan**.

Exemple d'utilisation :

Si vous lancez la commande :

gedit fichier.txt

L'éditeur se lance en avant plan et vous n'avez plus la main dans le terminal !

Si vous rajoutez un & après la commande :

gedit fichier.txt &

vous lancez l'éditeur en arrière plan et vous pouvez taper sur la touche entrée dans le terminal et vous gardez la main ! La commande jobs permet de voir la liste des processus en arrière plan.

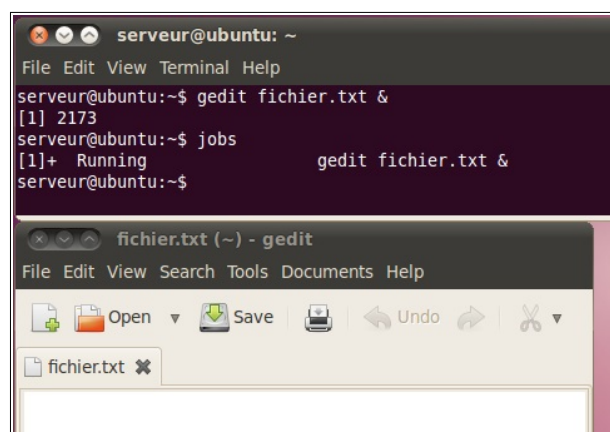


FIGURE 4 – Lancement d'un processus en arrière plan.

1.5 La Commande kill et les signaux

Un **signal** est un **message** envoyé à un processus. Le corps du message est en fait très simple puisqu'il est constitué en tout et pour tout d'un **entier** indiquant le type du signal. Pour la plupart des signaux, on peut redéfinir la procédure de réponse par défaut en installant un **gestionnaire de signal**. Toutefois certains signaux ne peuvent être déviés de leur signification première. Tout processus peut envoyer un signal à un autre processus, à partir du moment où il connaît son **PID**.

La liste des signaux peut être obtenue avec la commande : **kill -l**

```

serveur@ubuntu:~$ kill -l
1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL    5) SIGTRAP
6) SIGABRT   7) SIGBUS    8) SIGFPE    9) SIGKILL   10) SIGUSR1
11) SIGSEGV  12) SIGUSR2  13) SIGPIPE  14) SIGALRM   15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD  18) SIGCONT  19) SIGSTOP   20) SIGTSTP
21) SIGTTOU  22) SIGURG   23) SIGXCPU  24) SIGXFSZ   25) SIGVFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO     30) SIGPWR
31) SIGSYS   34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
serveur@ubuntu:~$

```

FIGURE 5 – Liste des signaux.

Voici une description de quelques signaux :

- SIGINT : Interruption du clavier (Ctrl+C dans un terminal). Par défaut arrête le processus. Peut être intercepté ou ignoré.
- SIGQUIT : Arrêt du processus. Un processus en avant plan peut être arrêté par son terminal avec CTRL-\. Peut être intercepté ou ignoré.
- SIGKILL : Terminer un processus. Ne peut être intercepté.
- SIGUSR1, SIGUSR2 : Signaux utilisateurs. L'action par défaut est la mise à mort du processus. Est en général intercepté et modifié par programme.

Pour envoyer un signal en ligne de commande on utilise la syntaxe :

kill -<num signal> <pid>

Le numéro de signal peut donné symboliquement ou numériquement (par exemple SIGKILL ou 9!) Exemple :

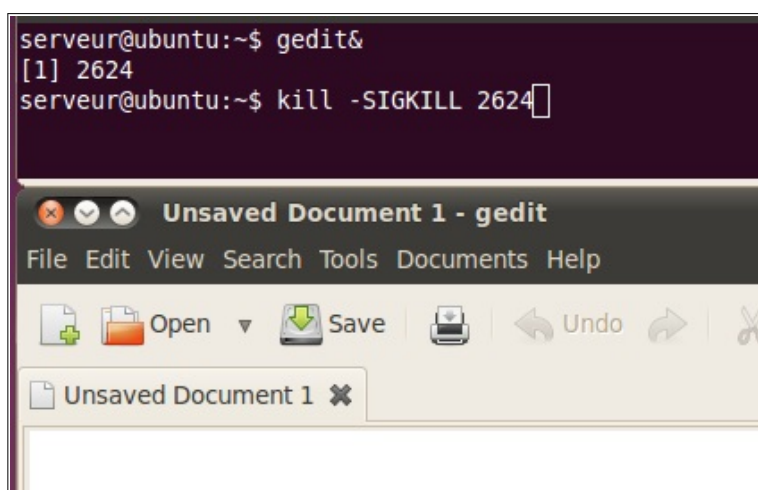


FIGURE 6 – Le kill -9.

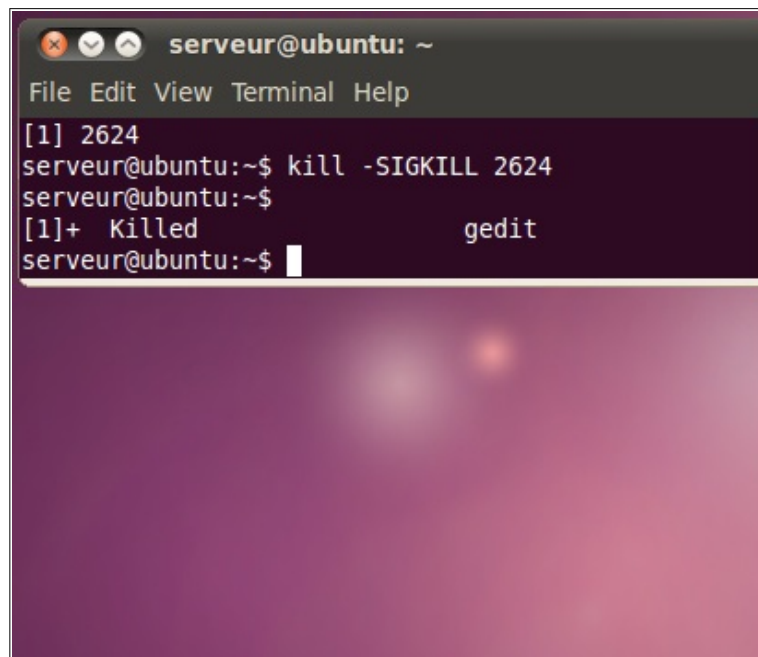


FIGURE 7 – Fin de la commande kill.

2 Programmation système autour des processus

Nous allons nous intéresser aux appels système qui permettent de créer des processus d'une part, puis aux mécanismes permettant aux processus de communiquer entre eux.

2.1 Appel getpid()

L'appel système `getpid()` permet d'obtenir le PID du processus qui tourne. Voici son prototype :

```
#include <sys/types.h>
#include <unistd.h>
int getpid();
```

La valeur de retour est le PID du processus qui s'exécute.

2.2 Appel système fork()

L'appel système `fork()` est le seul moyen de créer des processus, par duplication d'un processus existant. L'appel système `fork()` crée une copie exacte du processus original, comme illustré à la figure suivante :

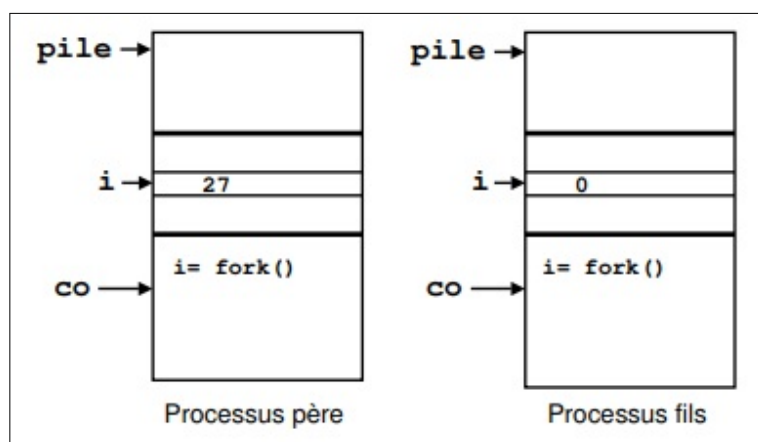


FIGURE 8 – processus cloné par fork.

Lorsque la fonction `fork()` s'exécute il y a duplication de la zone de données (variables globales et locales) et les deux processus père et fils exécutent le même code. Comment distinguer alors le processus père du processus fils ?

En fait la valeur de retour de `fork()` prend 3 valeurs possibles :

- 0 pour le processus fils ;
- >0 pour le processus père et qui correspond au **PID** du processus **fils** ;
- <0 si la création de processus a échoué, s'il n'y a pas suffisamment d'espace mémoire ou si bien le nombre maximal de créations autorisées est atteint.

C'est ce que montre la figure suivante :

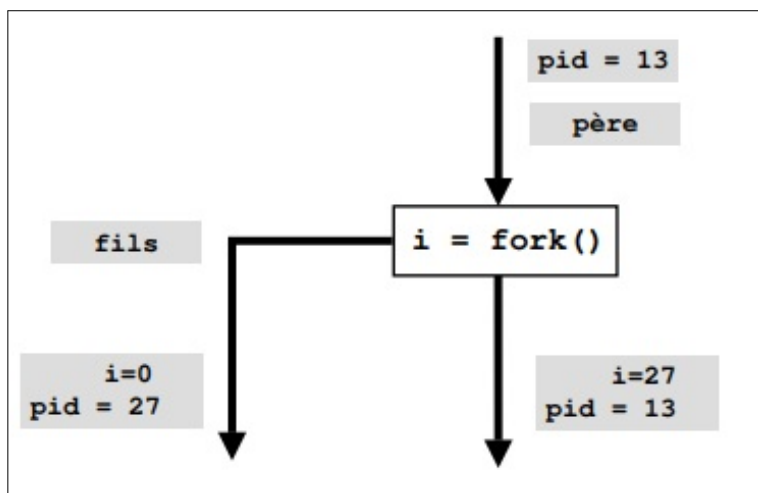


FIGURE 9 – processus cloné par fork (suite).

Voici un exemple de programme qui montre le principe de fork :

```
fork.c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main()
{
    int pid;
    printf("Avant fork mon pid vaut pid=%d\n",getpid());
    pid=fork();
    if(pid==0)
    {
        printf("Je suis le fils avec pid=%d\n",getpid());
    }
    else if(pid>0)
    {
        printf("Je suis le père avec pid=%d\n",getpid());
    }
    else
    {
        printf("Erreur de fork, pid=%d\n",getpid());
    }
}
```

FIGURE 10 – Exemple de fork.

Voici ce que ça donne :

```
File Edit View Terminal Help
serveur@ubuntu:~$ ./fork
Avant fork mon pid vaut pid=3570
Je suis le père avec pid=3570
serveur@ubuntu:~$ Je suis le fils avec pid=3571
```

FIGURE 11 – Exemple de fork : exécution.

On remarque une chose étrange : l'invite de commande de bash s'intercale avant l'exécution de la branche fils ! C'est parce que le code du père se termine avant que le fils ait eu le temps de s'exécuter.

2.3 Gestion de la fin de processus clonés

En général il vaut mieux que le processus père attende que le processus fils se termine avant de se terminer lui-même. Pensez à pstree qui affiche la hiérarchie des processus !

L'appel **wait()** est un des moyens pour le père d'attendre la fin du fils. le prototype est :

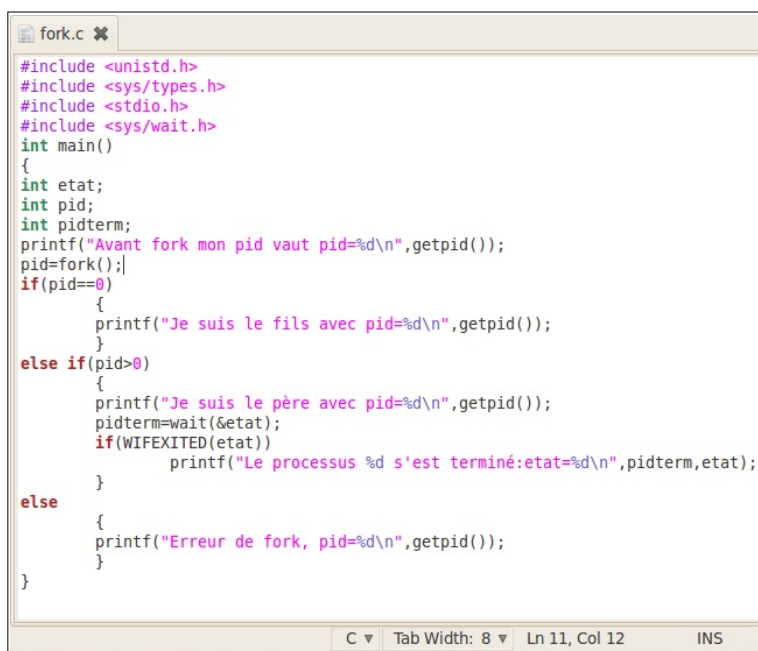
```
#include <sys/wait.h>
int wait (int *status);
```

Cette fonction attend la terminaison d'un quelconque de ses fils.

Paramètre : **status** est un pointeur d'un entier où **wait** va inscrire un compte rendu.

Valeur de retour : l'identifiant du processus fils qui s'est terminé. Elle retourne -1 si aucun fils n'est en train de tourner.

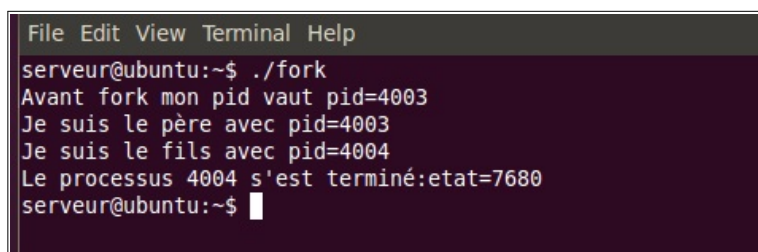
Voici la bonne version du programme précédent :



```
fork.c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/wait.h>
int main()
{
    int etat;
    int pid;
    int pidterm;
    printf("Avant fork mon pid vaut pid=%d\n",getpid());
    pid=fork();
    if(pid==0)
    {
        printf("Je suis le fils avec pid=%d\n",getpid());
    }
    else if(pid>0)
    {
        printf("Je suis le père avec pid=%d\n",getpid());
        pidterm=wait(&etat);
        if(WIFEXITED(etat))
            printf("Le processus %d s'est terminé:etat=%d\n",pidterm,etat);
    }
    else
    {
        printf("Erreur de fork, pid=%d\n",getpid());
    }
}
```

FIGURE 12 – Utilisation de la fonction wait.

On voit que tout se passe bien cette fois :



```
File Edit View Terminal Help
serveur@ubuntu:~$ ./fork
Avant fork mon pid vaut pid=4003
Je suis le père avec pid=4003
Je suis le fils avec pid=4004
Le processus 4004 s'est terminé:etat=7680
serveur@ubuntu:~$
```

FIGURE 13 – Utilisation de la fonction wait : résultat.

La macro renvoie vrai si le fils s'est terminé normalement, c'est-à-dire par un appel à **exit()** ou bien par un retour de **main()**.

Un autre appel système pour attendre la terminaison d'un processus bien précis (on le cible par son pid) est **waitpid**. Voici son prototype :

```
#include <sys/wait.h>
int waitpid(int pid, int *status, int options);
```

Description : Permet à un processus père d'attendre jusqu'à ce que le processus fils numéro **pid** termine. Il retourne l'identifiant du processus fils et son état de terminaison dans la valeur pointée par **status**.

Paramètres :

- **pid** : numéro indentifiant du processus dont il faut attendre la fin.
- **status** : adresse de l'entier où sera enregistré le compte rendu de terminaison
- **options** : voir le manuel

Valeur de retour : En cas de réussite, le PID du fils dont l'état a changé est renvoyé. Sinon -1 en cas d'échec.

2.4 L'appel getppid()

L'appel système **getppid()** permet d'obtenir le **PID** du **processus parent**. Voici son prototype :

```
#include <sys/types.h>
#include <unistd.h>
int getppid();
```

valeur de retour : le PID du processus père.