Introduction à Linux (IM713)

TP 3 – Manipulation de processus en langage C

On appelle *processus* un objet dynamique correspondant à l'exécution d'un programme ou d'une commande Unix. Cet objet regroupe plusieurs informations, en particulier l'état d'avancement de chaque programme, l'ensemble des données qui lui sont propres, ainsi que d'autres informations sur son contexte d'exécution.

Un *processus* représente l'ensemble programme en cours d'exécution + données de ce programme. Un utilisateur ne peut exécuter qu'un nombre limité de processus.

Le système d'exploitation attribue un numéro unique (pid) à chaque processus et conserve en mémoire une arborescence des processus, consultable grâce aux commandes *pstree*. Les processus sont groupés (un processus est un groupe à lui tout seul). Quand un processus est tué, son père en est normalement averti. Toutefois, il peut arriver que le père ne soit pas averti de la disparition de l'un de ses fils (notamment dans le cas où ce dernier a été victime d'un brutal 'kill -9'). Le fils est donc toujours référencé alors qu'il n'existe plus en réalité. Le processus defunct (ancien processus fils) ne disparaîtra donc qu'avec la mort du père.

Prenons par exemple la fameuse commande kill -9 -1. Elle envoie le signal 9 (SIGKILL) à tous les processus, les tuant sans condition. Par défaut, si aucun numéro de signal n'est précisé, le signal 15 (SIGTERM) est envoyé.

Quelle différence entre les signaux 9 et 15 ? Le signal 9 tuera à coup sûr et immédiatement le processus (sans lui demander son avis en quelque sorte), alors que le signal 15 donne l'ordre au processus de se terminer.

Le signal 9 est radical : il tue tout et ne laisse même pas le temps au système de prendre bonne note du déloguage. Le signal 15 fait tout dans les règles, c'est-à-dire qu'il donne l'ordre au processus de s'auto-terminer, ainsi qu'à ses fils (les processus qu'il a lui-même lancé). Ainsi, il se peut que certains fils indignes fassent la forte tête : des processus récalcitrants peuvent survivre.

$ ps aux | grep user | sort -n +4

sort est le fils de grep, qui est le fils de ps, lui même le fils du shell : ils constituent un groupe

La commande ps affiche la liste des processus.

-e : tous les processus

-d : tous sauf leaders de sessions

-f : full listing (pid, ppid)

-l : long listing (pid, ppid, uid, priorité)

-j : (pid, pgid et sid)

Manipulation

On ne crée jamais un processus, on le duplique avec l'appel système fork(). Après un fork, 2 processus identiques existent : tout est dupliqué, y compris les variables globales. Deux cas se présentent : le processus créateur fait son travail, ou le processus créé fait le sien.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    pid_t pid;
```

```
    switch(pid=fork()) {
           case -1: printf("Erreur %d \n",pid); break; // erreur
           case 0: printf("Fils %d \n",pid); break; // le fils execute cette partie
           default: printf("Pere %d \n",pid); break; // le pere execute cette partie
    }
    return 0;
 }
```

Remarque : les pid sont strictement positifs.

Le processus père reçoit l'id de chaque fils. Un processus n'a qu'un seul père. La fonction getpid() renvoie à un processus le pid de son père. Le fils et le père s'exécutent indépendamment après fork. Si le fils se termine, il restera zombie jusqu'à ce qu'un autre processus le remarque.

exit(int) // ne retourne jamais

abort() // crée un fichier core avant d'appeller exit

L'entier passé à exit() transmet de l'information au processus qui attend la fin du processus courant. Par convention, exit(0) correspond à une exécution normale sans erreur. Si un processus n'appelle pas exit, la valeur renvoyée est aléatoire, sauf si un return est effectué dans la fonction main().

Attendre un processus

Le shell effectue un fork pour exécuter une commande, puis attend la fin de ce processus. En C, la fonction wait() attend et retourne la valeur passée à exit() par le fils.

pid_t wait(int *) ;

// attend la fin de n'importe quel fils

// int * : valeur de retour passée à exit par le fils

pid_t waitpid(pid_t, int *, int) ;

// attend la fin d'un processus donné

// le 3e paramètre permet de passer des flags

wait est bloquante mais retourne immédiatement si le fils a terminé son exécution ou est devenu zombie.

waitpid ne bloque pas avec l'option WNOHANG

Remarque : Le processus père tourne sans arrêt. Quand il lance un fils, il récupère un int status :

Lien utile : :`http://sdz.tdct.org/sdz/la-programmation-systeme-en-c-sous-unix.html#Prsentationdesprocessus`

Quelques rappels en C

```
#include <stdio.h>
int main(int argc,char **argv)
{
/*argc: nombre d'argument dans la ligne de commande (y compris l'executable)
argv[i]:le ieme argument
*/
int i;
printf("le nombre d'argument est %d \n",argc);
/*affiche sur la sortie standard*/
for(i=0;i<argc;i++)
    printf("l'argument %d est %s \n",i,argv[i]);
return0;
```

```
}
```

En fait, main peut accepter un 3eme argument, qui sera l'ensemble des variables d'environnement

```
#include <stdio.h>
int main ( int argc , char **argv , char **env ){
  int  i ;
  for ( i =0; env[i] ! =NULL ; i++)
        printf ( "%d %s \n" , i , env[i] ) ;
return0;
}
```

Exercice 1 – *Informations d'un processus :* `=getpid() getppid() .... .`

1. Ecrire un programme qui affiche les information suivantes associées à un processus :
   - Le numero du processus (pid) : pid=getpid() ;
   - le numéro du père du processus (ppid) : ppid=getppid() ;
   - l'UID réel du processus (uid) : uid=getuid() ;
   - l'UID effectif du processus (euid) : euid=geteuid() ;
   - le GID réel du processus (gid) : gid=getgid() ;
   - le GID effectif du processus (egid) : egid=getegid() ;
2. Rependre l'exercice 1 et afficher les informations relatives aux processus père et fils

```
   ./a.out
Valeur fork = 0
Je suis le processus de pid : 22851
Mon pere est le processus de pid : 22850
Mon uid : 322
Mon euid : 322
Mon gid : 100
Mon egid : 100
Mon repertoire de travail : "/Users/chahir/exemple"


Valeur fork = 22851
Je suis le processus de pid : 22850
Mon pere est le processus de pid : 5411
Mon uid : 322
Mon euid : 322
Mon gid : 100
Mon egid : 100
Mon repertoire de travail : "/Users/chahir/exemple"
```

Exercice 2 – *Création et synchronisation de processus fils :* `fork`.

1. Ecrire un programme C qui crée deux fils, l'un affichant les entiers de 1 à 50, l'autre de 51 à 100.
2. Modifier le programme précédent pour que l'affichage soit 1 2 3 ...100.

Exercice 3 – *Exécution de commande shell :* `La commande execvp`.

1. Ecrire un programme C permettant de lancer la commande passée en argument. Exemples d'exécution :

```
monexec ls -l /etc/httpd/conf.d/
total 36
-rw-r--r-- 1 root root 3424 Sep 2 2022 auth_pgsql.conf
-rw-r--r-- 1 root root 814 Sep 3 2022 perl.conf
-rw-r--r-- 1 root root 459 Sep 3 2022 php.conf
-rw-r--r-- 1 root root 988 Sep 2 2022 python.conf
-rw-r--r-- 1 root root 180 Sep 4 2022 README
-rw-r--r-- 1 root root 251 Aug 6 2022 squirrelmail.conf
-rw-r--r-- 1 root root 11140 Sep 4 2022 ssl.conf
```

Exercice 4 – *Simultanéité vs. séquentialité : .*

1. Ecrire un programme C équivalent à la commande shell suivante :

```
who & ps & ls -l
```

1) Les commandes séparées par & s'exécutent simultanément.
2) Les commandes séparées par ; s'exécutent successivement.

Exercice 5 – *Synchronisation des processus père et fils par la commande :* `wait` .

1. Ecrire un programme qui prenant une matrice de taille 2*2 en paramètre et crée quatre fils. Chaque fils calcule un élément du carré de la matrice initiale et le renvoi au processus père comme code de retour.