TP4 : Gestion de processus sous Linux

Un processus est un programme en cours d'exécution. L'objectif de ce TP est de comprendre comment Linux crée et gère les processus.

1. La table des processus

Chaque processus Linux est identifié par un numéro unique, son PID (Process IDentifiaction). Le système gère une table des processus qu'on peut afficher avec la commande `ps`.

Affichez la liste de vos processus avec : `ps -fU <votreLogin>`. Quels sont les processus en cours ? A quoi correspondent les différentes colonnes de la table des processus ?

2. Création de processus

Un processus Unix peut en créer d'autres grâce à l'appel système `fork` (fourche). Un processus qui appelle `fork` est dupliqué par le système en un processus père et un processus fils (qui est l'exacte copie du père). Les processus père et fils poursuivent alors l'exécution du *même* programme. La valeur retournée par *fork* permet de distinguer entre le père et le fils pour réaliser des choses différentes dans la suite du programme. Cette valeur vaut :

- 0 dans le processus fils ;
- le numéro d'identification (`PID`) du processus fils créé, dans le processus père.

Copier le **répertoire** `fork` dans votre répertoire de travail depuis le serveur :

```
ssh imss-dc
cp -r /mnt/windata/commun/Lemaire/tpfork tpfork
exit
```

Ce répertoire contient :

- un fichier en langage C : `fork.c` ; vous éditerez ce fichier avec `gedit` ou `SCiTE` pour réaliser les programmes demandés ci-dessous.
- un fichier `Makefile` qui vous permettra de compiler `fork.c` en tapant : `make fork`.

Compilez et exécutez le programme `fork` et étudiez le code source pour comprendre le schéma de création d'un processus. Pour la suite du TP, vous modifierez le fichier `fork.c` ; pensez à bien recréer l'exécutable avec la commande `make` afin de rendre vos modifications effectives.

3. Identification et synchronisations simples.

3.1. Programmez pour chacun des processus père et fils l'affichage de son PID et du PID de son père (fonctions getpid() et getppid() qui renvoient un entier). Qui est le créateur (père) du processus père ? Pour le savoir, faites afficher la liste de vos processus avec : `ps -fU <votreNom>`

3.2. Avec `sleep`, retarder la mort (la fin de l'exécution) du processus père ou celle du fils (il s'agit d'inverser l'ordre de décès des deux processus obtenu au 3.1). Qui est le père du processus fils lorsque son père « naturel » se termine avant lui ? Pour le savoir, faites afficher la liste de vos processus avec la commande : `ps -ef | more`

3.3. La primitive `wait`, exécutée par un processus, bloque ce dernier jusqu'à la fin d'un de ses fils. Si le processus n'a pas de fils actif, `wait` renvoie -1, dans le cas contraire, elle renvoie le numéro (PID) du fils mort. Modifiez le schéma de création de façon à créer deux fils et à faire en sorte que le père les attende grâce à `wait(NULL)` et affiche pour chacun son identification (valeur retournée par `wait`). Les fils se contenteront d'afficher leur PID.

4. Exécution de programmes depuis un programme C

4.1. Lancer dans le fils, au moyen de la primitive `execlp` (voir ci-dessous), l'exécution de la commande `ps -fU votreLogin` et compléter de traces adéquates de manière à vérifier que le fils que vous avez créé et la commande `ps` ont bien le même `PID` (et qu'il s'agit donc bien du même processus).

4.2. Modifiez le programme du 3.3 pour que chaque fils lance un processus xterm (qui ouvre une fenêtre shell). Puis observez le comportement du père quand vous fermez les fenêtres (chaque fermeture de fenêtre doit provoquer l'affichage d'un message par le père).

5. Fonctions utiles

Pour utiliser les fonctions ci-dessous dans vos programmes, vous devez inclure le fichier de déclarations indiqué (par `#include`). Le numéro indiqué à droite est le numéro du manuel où la fonction est décrite. Pour avoir le manuel complet correspondant, taper : `man` *`n fonction.`* Par exemple : `man 2 fork`.

Afficher du texte sur la sortie standard

```
#include <stdio.h>
#int printf( const char *format, …);
```

La fonction `printf` permet d'afficher du texte sur la sortie standard. On doit préciser le type des variables qu'on veut afficher dans une chaîne formatée. Par exemple, pour afficher un entier:

```
pid_t pid_proc = getpid() ;
printf("Je suis le processus %d\n", pid_proc);
```

Le %d indique que la variable `pid_proc` est un entier.

Processus

```
#include <unistd.h>                                                    (3)
unsigned int sleep(unsigned int s)
```

Le processus appelant s'endort pour environ s secondes. La valeur renvoyée par sleep est la différence entre le nombre demandé et le nombre de secondes effectives de sommeil. Cette valeur peut être > 0 car sleep est suspendue par n'importe quel signal arrivant au processus. La plupart du temps, on se contente d'utiliser sleep comme une procédure : `sleep(2)`;

```
#include <unistd.h>                                                    (2)
pid_t fork() ;
```

Le processus appelant crée un processus identique à lui-même ; le processus appelant est le père et le processus créé est le fils. `fork()` rend la valeur 0 dans le processus fils, et le numéro d'identification (PID) du fils créé dans le processus père ; on récupère donc cette valeur par une instruction du type : `ident = fork()`. En cas d'erreur, `fork` renvoie -1.

```
#include <unistd.h>                                                    (2)
pid_t getpid() ;
pid_t getppid() ;
```

Renvoient au processus appelant son numéro d'identification (`getpid`) ou le numéro d'identification de son père (`getppid`). Renvoient -1 en cas d'erreur.

```
#include <stdlib.h>                                                    (2)
void exit(int status)
```

Le processus appelant met fin à son exécution. Un appel `exit(status)` est équivalent à un return `status` dans la fonction main d'un programme C. L'entier `status` est un code de terminaison et est rendu au processus père si celui-ci attend la fin de son fils (voir `wait`).

```
#include <sys/types.h>                                                 (2)
#include <sys/wait.h>
pid_t wait(int *stat loc)
```

Cette fonction permet à un processus d'attendre la mort d'un de ses fils. Si le processus n'a pas eu de fils, ou si tous les fils sont morts au moment de l'appel de `wait`, la fonction renvoie -1. Si un fils est déjà mort, la fonction renvoie le numéro d'identification de ce fils. Sinon le processus est bloqué jusqu'au décès d'un fils (le premier qui meurt) et renvoie son numéro. Cette fonction s'utilise de 2 façons :

`wait(NULL)` donne le fonctionnement décrit ci-dessus ;

`wait(&status)` la valeur renvoyé par le fils par exit est stockée dans l'entier `status`.

Exécution d'un programme

Un processus peut lancer un programme exécutable qui se trouve dans un fichier sur disque en utilisant un des appels système `exec()`. Le code du programme sur disque remplace le code du processus en cours et donc un appel à `exec()` ne retourne jamais, sauf en cas d'erreur où il retourne -1 et positionne la variable système `errno`. Il existe différentes versions de `exec()`, nous utiliserons ici **`execlp`**`()`.

```
#include <unistd.h>                                                    (2)
int execlp(char *path,
           char *arg0, char *arg1, ..., char *argn, NULL)
```

Dans cette version, `path` est une chaîne de caractères donnant le nom du fichier qui contient le programme à exécuter (cherché dans les chemins du `PATH`, `arg0` contient par convention le nom du fichier (sans répertoire) et les autres `arg` sont les paramètres du programme à exécuter. La liste des paramètres doit se terminer par un pointeur nul (NULL).

Exemple : `execlp("ls", "ls", "*.c", NULL)` ;