Université Grenoble Alpes
U.F.R. Sciences de l'Homme et de la Société
Master MIASHS - DCISS

TP3 : Gestion de processus

Un processus Unix peut en créer d'autres grâce à l'appel de la fonction système **fork**. 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). L'exécution des deux processus continue après l'appel de fork. La **valeur renvoyée par fork** permet de distinguer le père du fils :

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

Si vous travaillez sur une machine personnelle ou virtuelle Linux, décompressez le répertoire fork (fichier fork.zip) dans votre répertoire de travail : http://miashs-www.u-ga.fr/~adamj/documents/fork.zip

Si vous êtes connecté au serveur **miashs-dc.u-ga.fr** via Putty et Xming, copiez le répertoire `/media/commun/tplinux/fork` dans votre répertoire de travail par défaut par la commande suivante :
`cp -r /media/commun/tplinux/fork fork`

Positionnez-vous dans le répertoire `fork`. Ce répertoire contient :

- un fichier `fork.c` correspondant à un modèle de création d'un processus. Vous éditerez ce fichier avec **`gedit`** ou **`featherpad`** (sur les machine virtuelles Lubuntu 20) pour réaliser les programmes demandés ci-dessous. Vous pouvez aussi utiliser **nano**, en ayant ouvert une seconde fenêtre xterm.
- un fichier `Makefile` qui vous permettra de compiler `fork.c` en tapant simplement **`make`**.

Compilez (`make`) puis exécutez le programme fork (`./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 que vos modifications soient bien prises en compte**.

1. Identification et synchronisations simples.

1.1. 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).

Pour afficher des messages contenant des valeurs entières, on utilise l'instruction **`printf`** de la manière suivante : le premier argument de printf est le message à afficher, il contient des **`%i`** aux emplacements où une valeur entière doit apparaître ; les autres arguments sont les expressions dont le résultat est un entier, il doit y avoir autant d'expressions entières que de %i dans le message.

Exemple :

```
int a = 1234 ;
int b = 567 ;
printf("%i fois %i est égal à %i\n", a, b, a*b);
```

L'instruction printf ci-dessus affichera le message suivant :

```
1234 fois 567 est égal à 699678
```

Pour que le processus fils affiche son PID et le PID de son père, on pourra utiliser l'instruction suivante :

```
printf("Je suis le fils, mon PID est %i, mon père est %i\n",getpid(),getppid());
```

Après avoir modifié le programme de sorte que le processus père et le processus fils aient chacun affiché leur PID et celui de leur père, trouvez qui est le créateur (père) du processus père. Pour le savoir, affichez la liste de vos processus avec la commande : **`ps -l`**

1.2. Avec l'instruction `sleep` (voir sa description précise ci-dessous en partie 3 « fonctions utiles »), retardez la fin de l'exécution du processus père ou celle du fils (suivant l'enchaînement obtenu en 1.1 : il s'agit d'inverser l'ordre de fin des deux processus).

Qui est le père du processus fils lorsque son père « naturel » se termine avant lui ? Pour le savoir, faites afficher la liste des processus avec la commande : **`ps -el | more`**

1.3. Avec l'instruction `sleep` retardez à la fois le père et le fils, en faisant en sorte que le fils se termine avant le père : par exemple, faites `sleep(5)` pour le fils et `sleep(10)` pour le père. Lancez l'exécution de fork en arrière-plan par la commande `./fork&` et pendant l'exécution de fork, observez l'état des processus père et fils en faisant plusieurs fois de suite la commande `ps -l`
Ensuite modifier votre programme et faites en sorte que le père finisse avant le fils, et lancez à nouveau l'exécution de fork en arrière-plan par la commande `./fork&` et pendant l'exécution de fork, observez l'état des processus père et fils en faisant plusieurs fois de suite la commande `ps –l`
Notez les différences de comportement du père et du fils dans les 2 situations observées. Quand avez-vous vu un processus zombie ?

1.4. La primitive `wait` (voir sa description précise ci-dessous en partie 3), exécutée par un processus, bloque ce dernier jusqu'à la fin d'exécution de l'un de ses fils. Si le processus n'a pas de fils actif, `wait` n'est pas bloquant et renvoie -1, dans le cas contraire, `wait` renvoie le numéro (PID) du fils mort.

Modifiez votre programme 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 de faire une pause de longueur différente, puis d'afficher leur PID et celui de leur père.

Lancez votre programme ainsi modifié en arrière-plan et pendant son exécution observez l'état des processus père et fils en faisant plusieurs fois de suite la commande `ps –l`

2. Exécution d'une programme externe.

L'instruction **`execlp`** (voir sa description précise en partie 3 « exécution d'un programme ») permet à un processus de lancer n'importe quelle application. Au moment de l'appel de `execlp`, **le code du programme passé en argument remplace le code du processus en cours !** Par exemple, l'instruction suivante provoque le lancement de la commande xterm qui ouvre une nouvelle fenêtre de terminal avec un curseur rouge :

```
execlp("xterm", "xterm", "-cr", "red", NULL);
```

Tous les arguments de `execlp` sont des chaines de caractères, le programme à exécuter apparaît 2 fois, suivi s'il y a lieu des paramètres (options) d'exécution du programme, séparés par des virgules, NULL marque la fin de la liste des paramètres.

2.1. Lancez dans le premier fils, au moyen de l'instruction **`execlp`**, l'exécution de la commande **`ps -l`** et complétez par des traces adéquates de manière à vérifier que ce fils que vous avez créé et la commande **`ps`** ont bien le même `PID` (et donc qu'il s'agit bien du même processus).

2.2. Modifiez votre programme pour que chaque fils lance un processus qui ouvre une fenêtre (**xterm** ou **`xeyes`** ou **`xclock`** ou **`xcalc`**). 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).
Relancez votre programme et observez les états des processus à l'aide de la commande **`ps –l`** après chaque fermeture de fenêtre. Il n'y a plus de zombie, pourquoi ?

3. 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`

Fonctions concernant les 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. Attention, tout appel à la fonction `fork()` provoque cette création ! `fork()` renvoie 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 rend `-1`.
Si un fils est déjà mort, la fonction rend le numéro d'identification de ce fils. Sinon le processus est bloqué jusqu'au décès d'un fils (le premier qui meurt) et rend son numéro (cette attente n'est donc pas sélective).
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 de nombreuses versions de fonctions `exec (execl, execv, execle, execve, execlp, execvp)` mais on ne donne ici que le détail pour la plus simple à utiliser, pour les autres faire : `man 2 exec`.

```
#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("ps", "ps", "-l", NULL);`