TP Linux : Les processus

Objectifs : - Création de processus
- Compréhension du fonctionnement multitâche du système Linux

OS : linux

Rappel : Commande *ps*

Taper la commande : *ps*
et la commande *ps -alx | more*

Cette commande permet de visualiser la liste des processus[1] en cours d'exécution (au moment où on tape la commande!). Sans les options, seuls les processus de l'utilisateur sont affichés.
Le « **| more »** permet d'avoir l'affichage page par page.

Un certain nombre d'informations sont ainsi affichées. On peut voir, pour un processus : son PID (Process Identifier), son PPID (Parent PID), son UID (User Identifier), sa fenêtre de sortie standard (TTY) s'il en a une, et bien sur le nom de l'exécutable (COMMAND).

Notez que la commande ps comporte un nombre impressionnant de combinaisons d'options qui permettent de sélectionner précisément ce que l'on veut afficher. Il existe également deux formats de commande : les options sont précédées d'un tiret, et les options indiquées sans le tiret. Suivant le cas, le résultat n'est pas le même. Avec l'habitude, on retient généralement une ou deux commandes ps selon son goût !

Testez : **`ps afu`** et **`ps al`**
Incroyable ! des options (a, f, u, l) sans le signe – devant ! C'est très rare sous Linux…

Observez : Notez les deux façons de représenter la filiation des processus (PPID → PID)

Rappel : Arrière plan et Avant plan

Linux étant un système multi tâches il est possible à un utilisateur d'exécuter plusieurs processus « simultanément » (en réalité, un ordonnanceur répartit l'activité du microprocesseur entre les différents processus, les uns après les autres, ce qui donne cette impression que tout s'exécute simultanément).

Pour mettre en évidence cette fonctionnalité multi tâches, nous utiliserons la commande *sleep* qui se contente d'attendre un nombre défini de secondes.

Nous utiliserons également :

- l'argument & qui envoie l'exécution d'une commande en arrière plan,
- la commande *fg* qui place un processus en exécution avant-plan
- la commande *bg* qui place un processus en exécution arrière plan

[1] Un processus est une instance d'un programme qui s'exécute.

Rappel : Commande kill

La commande ***kill***[2] envoie un signal à un processus en précisant le PID du processus concerné.

kill -n°du_signal PID_du_processus

La commande « kill –l » affiche la liste des signaux disponibles.

A- Programmation C/C++ : Le FORK de base : Création d'un processus fils

L'appel système *fork()* (on dit aussi primitive *fork*) permet la création d'un nouveau processus (processus fils) à partir d'un processus que l'on appelle « processus père ». Ce « processus fils » est l'exacte copie du processus père. Autrement dit, le fils va, à partir du *fork()*, exécuter le même code source que le père.
La primitive *fork()* permet la création dynamique d'un nouveau processus qui s'exécute de manière concurrente avec le processus qui l'a créé. Tout processus Linux, hormis le processus 1, est créé à l'aide de la primitive *fork()*.

Le programme de test :

TRAVAIL : Utilisez un éditeur de texte pour créer le programme fourni page suivante, puis compilez-le pour créer un exécutable.

Par exemple, si le programme source s'appelle prog.cpp, la compilation se fera avec :

```
g++   prog.cpp   -o prog.exe
```

Et le lancement du programme se fera avec :

```
./prog.exe
```

NB : Sous Linux, in n'est pas nécessaire d'ajouter .exe au nom du programme.

[2]kill contrairement à sa traduction, ne signifie pas uniquement « tuer » sous linux

Programme de test du FORK :

```
#include <iostream>
#include <unistd.h> // Fork

using namespace std;

int main()
{
    int a;
    int fk;

    cout << endl << endl << "Debut du processus pid " << getpid() << endl;
    cout << "Tapez un chiffre:" << flush;
    cin >> a;
    fk = fork();
    cout  << "Suite du processus pid " << getpid() << " fk:" << fk << endl;
    if ( fk == 0 ) {
        cout << "(if) --> Processus:" << getpid() << " Saisie :" << a << endl;
        a = a * 2;
        cout << "(if) --> Variable 'a' modifiee : " << a << endl;
        sleep(10);
    }
    else {
        cout << "(else) --> Processus pid " << getpid() " Saisie :" << a << endl;
        a = a * 3;
        cout << "(else) --> Variable 'a' modifiee : " << a << endl;
        sleep(10);
    }
    cout << "Arret processus pid " << getpid() << " Contenu de 'a' :" << a << endl;
}
```

Observez le résultat de votre programme :

1. Lorsque vous avez saisi le nombre, sur une autre console, utilisez la commande **ps** de votre choix pour mettre en évidence que votre processus a bien créé un processus enfant avec un autre numéro de pid. NB : vous avez 10 secondes pour le faire …
   Notez ce numéro : Cliquez ici pour taper du texte.

2. Après avoir saisi le chiffre, observez les messages. Dans le processus principal, à quoi correspond la valeur de fk ?
   - ☐ fk vaut 0
   - ☐ fk prend une valeur aléatoire
   - ☐ fk donne le pid du fils

3. Même question pour le processus fils ?
   - ☐ fk vaut 0
   - ☐ fk prend une valeur aléatoire
   - ☐ fk donne le pid du père

4. A partir de quelle ligne on peut dire qu'il y a 2 processus ?
   Cliquez ici pour taper du texte.

5. On s'intéresse à la variable « a » présente dans les 2 processus. Quelle est sa valeur dans le processus fils au moment de la création de celui-ci ?
   - ☐ « a » vaut 0
   - ☐ « a » a une valeur non déterminée
   - ☐ « a » a la même valeur que dans le processus père

6. Que peut-on dire de la variable « a » présente dans les 2 processus ?
   - ☐ Elle est identique dans les 2 processus
   - ☐ Elle est différente ; chaque processus a sa propre variable « a »
   - ☐ Dans le processus fils, elle suit les modifications du processus père.

7. Quelles sont les numéros de lignes exécutées par le processus parent ?
   Cliquez ici pour taper du texte.

8. Quelles sont les numéros de lignes exécutées par le processus enfant ?
   Cliquez ici pour taper du texte.

9. Copiez ici le « screen shot » du l'ensemble des messages du programme :
   Cliquez ici pour taper du texte.

B - Programmation C++ : Fork fou

Combien de fois de mot « Fils » apparaitra-t-il si on exécute le code suivant ? :

```
1  #include <iostream>
2  #include <unistd.h> // Fork
3
4  using namespace std;
5
6  int main()
7  {
8      int i;
9      int fk;
10
11     for ( i=0; i<5; i++) {
12         fk = fork();
13         if ( fk == 0 ) {
14             cout << "Fils " << getpid() << endl;
15         }
16     }
17
18     int a;
19     cin >> a;
20 }
```

10. Testez ce code. Comme la ligne 21 empêche le processus de se terminer, vous pouvez observer à partir d'un autre terminal la filiation des processus. Copiez ici le « screen shot » :

Cliquez ici pour taper du texte.

11. Comment modifier le programme pour qu'il ne lance que 5 processus fils, comme on l'avait imaginé au départ ?

Cliquez ici pour taper du texte.

C- Programmation C/C++ : Si vous avez compris …

Etudiez et testez le code source c++ fourni (Fork-Partie6.cpp).
Notez que le travail principal consiste à écrire des messages à intervalle régulier dans un fichier toto.log.

Les inconvénients vous apparaitront rapidement : l'écran reste bloqué pendant l'exécution du travail. On souhaite utiliser pleinement le menu : lancer plusieurs processus et pourvoir les supprimer, quitter le logiciel…

12. Apportez la modification nécessaire à cet objectif.
13. Etudiez le comportement d'un processus fils lorsqu'il termine son activité alors que le processus père est toujours actif. Corriger le programme pour que le processu fils ne reste pas référencé (Zombie) dans la mémoire système à la fin de son exécution (signal SIGCHILD à modifier selon le cours).

Cliquez ici pour taper du texte.