TP 5

Début avec les processus

Durée prévue : 1 séance (1h30)

On fera attention à la différence entre le chiffre `1` et la lettre `l`. Les fichiers mentionnés dans le sujet sont dans l'archive `lifasr5_tp05.tgz` téléchargeable depuis la page web de l'UE.

EXERCICE 1 ▶ Observation des processus depuis le shell

Rappels : Un processus est un programme en cours d'exécution sur le système. L'un des rôles du système d'exploitation est de permettre à un ensemble de processus de progresser dans leur exécution en même temps sur votre système : il doit partager les ressources disponibles (processeurs, mémoire, périphériques) entre les différents processus. Ainsi, à chaque instant de nombreux processus sont soit en cours d'exécution, soit bloqués en attente d'exécution, mais vous avez toujours l'impression qu'ils s'exécutent en « même temps ».

1) Par défaut, la commande `ps` liste les processus rattachés au terminal (`TTY`) dans lequel elle est lancée. Ouvrez un terminal, et entrez la commande `ps` : parmi les informations listées, interprétez les colonnes `CMD`, `PID` et `TTY`.

.....

.....

Dans la suite, on va considérer un exemple particulier de processus dont l'exécution "se voit à l'œil nu", c'est `xclock`. Évidemment ce n'est qu'un exemple...

— Si ce logiciel n'est pas disponible sur votre machine personnelle vous pouvez l'installer (le paquet Ubuntu est `x11-apps`).

— S'il n'est pas disponible sur les machines de TP, utilisez à la place le programme `clock.py` qui vous est fourni à la place de la commande `xclock -update 1` indiquée dans le sujet.

2) Lancez deux fois la commande `xclock -update 1 &`, puis entrez à nouveau `ps` : qu'observez-vous ?

.....

.....

3) Maintenant, entrez la commande `bash`, puis lancez à nouveau une horloge à l'arrière plan avec `xclock -update 1 &`. Utilisez maintenant la commande `ps -l` pour afficher plus d'informations. Comment interprétez-vous la colonne `PPID` ?

.....

.....

4) Dans le résultat de la commande `ps -l` précédente, comment interprétez-vous la colonne `UID` ? Pour vous aider, vous pouvez aussi vous intéresser au résultat de la commande `id`.

.....

.....

5) Vous avez dû comprendre que les processus sont organisés d'une façon arborescente. Pour la visualiser, vous pouvez utiliser `pstree` : à l'aide de `ps`, repérer le `PID` du premier processus `bash` lancé dans votre terminal ; on note n ce `PID` ; entrez la commande `pstree -p` n. Que constatez vous ?

.....

.....

6) Maintenant, ouvrez un *nouveau* terminal, puis entrez la commande `ps` : vous ne retrouvez pas dans la liste les processus que vous aviez lancés dans le premier terminal... Comment afficher la liste de tous les processus que vous avez lancés sur le système ?

.....

7) Comment afficher tous les processus lancés sur le système ?

.....

8) Fermez tous ces terminaux utilisés pour l'exercice (clic sur la croix). Cela supprime les processus `xclock` lancés et les éventuelles connections ssh. On fera "plus propre" dans l'exercice suivant.

EXERCICE 2 ► Les débuts avec fork()

Rappels : La fonction `fork()` permet à un processus, dit « père », de se dupliquer : à la suite d'un appel réussi à `fork()`, le système comporte un nouveau processus, dit « fils », qui est une copie du père. **Il est important de comprendre qu'après l'appel à fork(), le code est *commun* au père et au fils.**

1) Consultez le manuel de la fonction `fork()`. Quelles valeurs retourne-t-elle ?

.....

2) Consultez le manuel des fonctions `getpid()` et `getppid()`. Quelles valeurs retournent-t-elles ?

.....

3) On considère le source `unfils.cpp` (que vous devez télécharger sur la page de l'UE) suivant :

```
10 int main(void) {
11   int pid;
12   int a = 0;
13
14   cout << "Je suis le père, de PID " << getpid() << ". "
15        << "Je vais créer un fils..." << endl << flush;
16
17   pid = fork();
18
19   if(pid == -1) {
20     cerr << "Erreur, aucun fils n'a été créé : " << strerror(errno) << "."
21     << endl << flush;
22     return 1;
23   }
24
25   if(pid > 0) {
26     cout << "Je suis le père, de PID " << getpid() << ". "
27          << "J'ai un fils dont le PID est " << pid << "."
28          << endl << flush;
29   }
30   else {
31     cout << "Je suis le fils, de PID " << getpid() << ". "
32          << " Mon père a pour PID " << getppid() << "."
33          << endl << flush;
34   }
35
36   cout << "Je suis le processus de PID " << getpid() << ". "
37        << "Mon père a le PID " << getppid() << "."
38        << endl << flush;
39
40   if(pid > 0) {
41     cout << "Je suis le père, de PID " << getpid() << ". "
42          << "Je vais me terminer..." << endl << flush;
43   }
44   else {
45     cout << "Je suis le fils, de PID " << getpid() << ". "
46          << "Mon père a pour PID " << getppid() << ". "
47          << "Je vais me terminer..." << endl << flush;
48   }
49
50   cout << "Variable a : " << a << endl << flush;
51
52   return 0;
53 }
```

Compilez avec `g++ -Wall -o unfils unfils.cpp` ce programme, et exécutez-le plusieurs fois **dans un premier terminal**. Ouvrez un **second terminal**, qui vous servira pour « observer » les processus que vous lancez (débrouillez vous pour pouvoir afficher les deux terminaux en même temps sur votre écran).

4) Dans le programme, comment fait-on pour différencier le code qui va être exécuté uniquement par le père, et celui qui va être exécuté spécifiquement par le fils ?

.....

.....

5) L'affichage des lignes 36 à 38 est effectué par le père et par le fils : pourquoi ?

.....

.....

6) Quels processus exécutent le `return 0` de la ligne 52 ?

.....

7) Quel est le PID du bash dans lequel vous exécutez votre programme (`ps`)?

.....

8) Quel est le processus dont le PID est retourné par `getppid()` pour le père?

.....

9) En utilisant `sleep(60)`, modifiez le programme de façon à ce que le père se termine avant que le fils n'affiche son dernier message : quel est le père du processus fils lorsqu'il affiche ce dernier message? Utilisez pour répondre, depuis votre terminal « d'observation », la commande `ps -l PID_DU_FILS` pour obtenir le PID du père, puis cherchez le nom de ce processus en utilisant une commande similaire.

.....

.....

.....

10) Modifiez le programme de façon à ce que le père se termine longtemps après que le fils n'affiche son dernier message (en utilisant `sleep(60)` par exemple) : à l'aide de `ps -l`, observez quel est l'état du processus fils avant que son père se termine. Pourquoi reste-t-il dans cet état?

.....

.....

11) Modifiez le programme pour que le père attende, avec la primitive système `waitpid()` (`man waitpid` si besoin) que son fils se termine avant de se terminer lui-même. Faites en sorte que le fils reste en sommeil 60 secondes avant de se terminer. Vérifiez que le père attend bien que son fils termine : pour cela, utilisez dans votre terminal « d'observation » la commande `pstree`, pour afficher l'arborescence des processus en partant du bash dans lequel vous avez lancé votre programme.

12) Si le fils modifie le contenu de la variable `a`, est-ce que cette modification va être visible pour le père? Inversement, si le père modifie le contenu de la variable `a`, est-ce que cette modification va être visible pour le fils? Expérimentez pour répondre à ces questions.

.....

.....