TP N°03 : Gestion des processus (Partie I)

Remarque :
- **Créez un répertoire réservé à vos exercices de programmation.**
- **L'exercice 3.6 est implémenté et exécuté dans un environnement Windows.**

Exercice 3.1 (Création de processus avec system())

La fonction **system** lance l'exécution d'un processus SHELL interprétant la commande passée en argument dans la chaîne de caractères ch : **int system (char *ch)**
Cette fonction crée pour cela un nouveau processus, qui se termine avec la fin de la commande. Le processus à l'origine de l'appel de system est suspendu jusqu'à la fin de la commande.

Q1) Ecrire un programme qui lance la commande shell « ls l / » à l'aide de la primitive system.

Exercice 3.2 (La primitive fork())

Q1) Quel est le résultat affiché sur écran en exécutant chacun des deux programmes suivants :

```
int main (int argc, char *argv[])
{
    int i;
    for (i =0 ; i <4 ; i++)
       fork();
    printf ("A\n");
}
```

```
int main (int argc, char *argv[])
{
   int i;
   for (i = 0; i < 4; i++)
      if (fork () == 0)
      {
           printf ("A\n");
           exit (0);
      }
   printf ("B\n");
   while (wait(NULL) >= 0);
}
```

Q2) Quel est le rôle de la boucle : while (wait(NULL) >= 0); ?

Exercice 3.3 (getpid, getppid, execl)

Considérons le programme suivant :

```
#include<unistd.h>
main( ) {
      fork();
      fork();
      fork();
}
```

Q1) Sans faire l'exécution, donner le nombre de processus générés

Q2) Utilisez l'appel système **getpid()** et **getppid()** pour construire l'arbre des processus engendrés.

Q3) Remplacez le premier appel à « **fork()** » par un appel à « **execl()** » qui exécute le programme lui-même. Que se passe-t-il ?

Q4) Remplacez le deuxième appel à « **fork()** » par un appel à « **execl()** » qui exécute le programme lui-même. Que se passe-t-il maintenant ?

Exercice 3.4 (L'arbre généalogique)

Considérons les 3 programmes définis ci-dessous.

```
#include<unistd.h>
main()
{
      fork() && fork() ;
}
```

```
#include<unistd.h>
main()
{
      ( fork() || fork() ) ;
}
```

```
#include<unistd.h>
main()
{
      fork() && (fork() || fork()) ;
      sleep(2)
}
```

Q1) Sans les exécuter, dessiner l'arbre généalogique engendré par leurs exécutions.

Q2) Implémenter et exécuter ces programmes pour comparer les résultats.

Exercice 3.5 (Fonction wait)

Soit le programme suivant :

```
#include<stdio.h>
#include<unistd.h>
#include <sys/types.h>
int main (int argc, char *argv[])
{
    pid_t id;
    id = getpid();
    int i;
    for (i = 0; i < 3; i++)
        fork ();
    /*1*/
    if (getpid () == id)
        printf ("Le grandpère: %d fils du shell: %d se termine\n", getpid(), getppid());
    else
        printf ("Le processus: %d fils de: %d se termine\n", getpid(), getppid());
    return (0);
}
```

Q1) Exécuter ce programme plusieurs fois. Que remarquer vous ?

Q2) Juste devant le commentaire **/*1*/**, ajouter le code : **while (wait(NULL) >= 0);** et réexécuter plusieurs fois le programme. Que remarquer vous ? Déduire le rôle du code ajouté.

Exercice 3.6 (Créer un processus sous Windows) [environnement Windows]

Sous Windows, il y avait principalement trois options pour écrire des programmes multi-tâches :

- Utiliser la **Win32** Application Programming Interface (API) en conjonction avec la C runtime library (CRT). Comme son nom l'indique, l'API est implémentée en C et elle intègre les fonctions d'appels systèmes de plus bas niveau accessibles au développeur d'applications Windows actuel.

- Utiliser la **Microsoft Foundation Class** (MFC) en conjonction avec le C++. La MFC a été développée comme un ensemble de classes C++ qui agissent en tant que «wrappers » de l'API Win32, afin d'en faciliter l'utilisation et l'intégration dans le cadre d'un développement orienté-objet.
- Utiliser la plateforme **.NET** qui offre plusieurs options aussi pour décomposer une application en processus légers. Le « namespace » System.Threading en est une.

Évidemment, d'autres framework sont disponibles. On peut citer : Qt, CLX/VCL de Borland, …

L'appel **CreateProcess** de la Win32 permet la création et l'exécution d'un processus enfant.

Remarque : Si on veut comparer avec Unix, l'appel **CreateProcess()** est l'équivalent des appels **fork()** et **exec()** regroupés.

Dans la plateforme Moodle, Télécharger le dossier **« processus-src-windows »** qui contient les fichiers father, child, CreateProcess.
(https://elearning.univ-msila.dz/moodle/mod/folder/download_folder.php?id=130686)

Q1) Compiler (avec Dev-Cpp ou MinGW) et tester. Vérifier les valeurs affichées en utilisant le gestionnaire de tâches de Windows (**taskmgr.exe**)

TP N°03 : Gestion des processus (Partie II)

Exercice 3.7 (Manipulation de Signaux)

Écrivez un programme en C qui crée un processus fils.

- Le processus fils doit envoyer un signal personnalisé au processus parent.
- Le processus parent doit afficher "Signal reçu" lorsqu'il reçoit le signal.

Exercice 3.8 (Gestion de Signaux en Boucle)

Écrivez un programme en C qui crée un processus.

- Le processus doit exécuter une boucle infinie dans laquelle il attend un signal (`SIGINT`).
- Lorsqu'il reçoit le signal, il doit afficher "Signal d'interruption reçu" et se terminer.

Exercice 3.9 (Signaux)

Q1) Ecrire un programme qui affiche bonjour lorsqu'il reçoit le signal USR1, bonsoir lorsqu'il reçoit le signal USR2 et quitte après avoir affiche fin du programme lorsque l'on tape Ctrl-C sur le clavier.

Q2) Tester le programme en envoyant les signaux à partir du shell

Exercice 3.10 (Signaux et processus zombie)

Ecrire un programme qui crée un fils. Le père déroute le signal SIGCHLD pour supprimer le fils zombie, puis il fork. Le fils affiche son pid. Le programme ne se terminera pas, il faut le tuer à partir de la ligne de commande

Exercice 3.11 (Synchronisation)

Ecrire un programme dans lequel un processus crée un fils et initialise un handler (afficher BONJOUR) sur SIGUSR1. Le fils affiche des informations à l'écran puis envoi le signal SIGUSR1 a son père.

Attention le programme fils doit se terminer avant le processus père.

La sortie (les messages sur l'écran) du programme devra ressembler à ceci :

père *: Je suis le processus 27406*

fils *: Je suis le processus 27407*

père *: j'attends un signal BONJOUR*

fils *: fin du processus*

père *: fin du processus*

Exercice 3.12 (Un Tube)

Écrivez un programme en C qui crée un tube (pipe), puis crée un processus fils. Le processus père envoie le message "Bonjour !" au processus fils via le tube (pipe), et le processus fils lit le message et l'affiche à l'écran.

Exercice 3.13

Écrivez un programme en C qui crée un tube (pipe), puis crée deux processus fils, P1 et P2. Le processus P1 envoie les nombres pairs de 1 à 10 au processus P2 via le tube (pipe), et le processus P2 lit les nombres et les affiche à l'écran.

Exercice 3.14 (Deux Tubes)

Écrivez un programme en C qui crée deux pipes, puis crée trois processus fils (P1, P2 et P3). Le processus P1 envoie les nombres de 1 à 10 au processus P2 via le premier tube (pipe). Le processus P2 multiplie ces nombres par 2 et les envoie au processus P3 via le deuxième tube (pipe). Le processus P3 affiche les nombres finaux.

Exercice 3.15 (Héritage d'un tube ordinaire)

Ecrire un programme qui crée un tube (pipe) ordinaire puis un fils.
Successivement, le père et le fils s'échangent des messages (Salut PAPA et Salut Fils).

Exercice 3.16 (Tube sans écrivains)

- **Q1)** Ecrire un programme qui crée un tube (pipe) ordinaire p et lit dans p[0].
- **Q2)** Que se passe-t-il si le programme ne ferme pas p[1] avant de lire dans p[0] ?
- **Q3)** Même question si le programme ferme p[1] avant de lire dans p[0]?

Exercice 3.17 (Tube sans lecteurs)

- **Q1)** Ecrire un programme qui crée un tube (pipe) ordinaire p et écrit dans p[1].
- **Q2)** Que se passe-t-il si le programme ne ferme pas p[0] avant d'écrire dans p[1]?
- **Q3)** Même question si le programme ferme p[0] avant d'écrire dans p[1] ?