# Travaux Pratiques N° 2

**Manipulation des processus : utilisation des primitifs systèmes C sous Unix**

## 1) Rappel théorique

**a°) La primitive fork () :**

Permet de dupliquer un processus père

Retourne la valeur –1 lorsque elle échoue (par manque d'espace mémoire ou l'utilisateur a déjà créé trop de processus)

Retourne PID (numéro du fils) dans le processus père

Retourne la valeur 0 dans le processus fils

**b°) les primitives getpid() et getppid() :**

- **getpid () :** retourne l'identification du processus courant

- **getppid () :** retourne l'identification du processus père

**c°) la primitive exit ()**

- **exit ( ) :** ne retourne pas de valeur. Elle termine le processus qui l'appel : v*oid exit ( int s )*

L'argument s est un entier qui indique au shell qu'une erreur s'est produite. On la laisse à zéro pour indiquer une fin normale.

**d°) la primitive wait ()**

- **wait ()** permet à un processus d'attendre la fin de l'un de ces fils. Si un processus n'a pas de fils ou si une erreur s'est produite wait () retourne –1. Sinon wait () bloque le processus père jusqu'à la fin de l'un de ces fils et elle retourne son PID. int wait( int srt) avec str doit être nul.

## 2) Travail demandé

### Exercice N°1

1. Après compilation de exo1.c, on exécutera le programme exo1 plusieurs fois. Que se passe-t-il ? Pourquoi ?

```
#include <sys/types.h>
#include <unistd.h>
int main ()
{
      int valeur;
     valeur = fork();
     printf (" Valeur retournee par la fonction fork:
%d\n", valeur);
     printf ("Je suis le processus numero %d\n",
getpid());
```

```
return 0 ;
}
```

2. Ajouter maintenant la ligne suivante derrière l'appel à `fork` :
   `if (valeur == 0) sleep (4);`
   Que se passe-t-il ? Pourquoi ?

## Exercice N° 2

1- Compiler le programme `fork-mul.c`, puis l'exécuter.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main ()
{

  int valeur, valeur1 ;
  printf (" print 1 - Je suis le processus pere num=%d
\n", getpid() );
  valeur = fork();
  printf (" print 2 - retour fork: %d - processus num= %d
-num pere=%d \n", valeur, getpid(), getppid() );
  valeur1 = fork();
  printf (" print 3 - retour fork: %d - processus num= %d
-num pere=%d \n", valeur1, getpid(), getppid() );
}
```

2- Après exécution et à l'aide du schéma suivant, relever les numéros des processus et numéroter l'ordre d'exécution des instructions printf de façon à retrouver l'ordre d'exécution des processus.

## Exercice N°3

Compiler et exécuter `fork-sync.c`

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main ()
{
  int valeur, ret_fils,etat ;
  printf ("Je suis le processus pere num=%d \n",
getpid());
  valeur=fork();
```

```
  switch (valeur)
    {
    case 0 :
      printf
     ("********* FILS *********\n");
      printf ("Proc fils num= %d - Pere num= %d
\n",getpid(),getppid() );
      printf("Je vais dormir 30 secondes ...\n");
      sleep (30);
      printf("Je me reveille,Je termine mon execution par
un EXIT(7)\n");
      exit (7);
    case -1:
      printf ("Le fork a echoue");
      exit(2);
    default:
      printf("********\n* PERE *\n********\n");
      printf ("Proc pere num= %d -\n Fils num= %d \n",
getpid(),valeur );
      printf ("J'attends la fin de mon fils: \n");
      ret_fils = wait (&etat);
     printf("Mon fils de num=%d est termine,\nSon etat
     etait :%d\n",ret_fils,etat);
    }
Return 0 ;
}
```

## Exercice N°4

En utilisant la primitive fork ( ) écrire un programme qui permet de créer la hiérarchie des processus présentée par la figure ci-dessous, en donnant à chaque fois le PID et le PPID de chaque processus.