TP n° 2

Programmation des processus sous GNU/Linux

1. Quelques primitives de gestion des processus

Il s'agit de primitives et de fonctions de la bibliothèque C standard **unistd.h**, qui donnent des informations générales sur les processus qui les utilisent.

1.1. Identification de processus

getpid() et **getppid()** fournissent respectivement le numéro du processus appelant et celui du processus parent.

1.2. Mise en sommeil d'un processus

sleep(int n) suspend l'exécution du processus appelant pour une durée de **n** secondes.

1.3. Création de processus : fork()

Cette primitive qui permet la création dynamique d'un nouveau processus, qui s'exécute de façon concurrente avec le processus qui l'a créé. **fork()** entraîne la création d'un nouveau processus, qui est une copie exacte du processus appelant. Cela signifie que le processus ainsi créé, appelé processus fils, hérite du processus qui l'a créé, appelé processus père, d'un certain nombre de ses attributs :

- Le même code ;
- Une copie de la zone de données ;
- L'environnement, la priorité, les différents propriétaires.

Le seul moyen de distinguer le processus fils du processus père est la valeur de retour de la fonction **fork()**. Elle vaut zéro dans le processus fils créé, et elle est égale au numéro du processus fils (différent de zéro) dans le processus père.

Si la primitive **fork()** échoue (et qu'il n'y a donc pas création d'un nouveau processus), la valeur de retour de la fonction est -1. C'est le cas si l'utilisateur a lancé trop de processus, ou si le nombre total de processus sur le système est trop élevé.

1.4. Synchronisation de processus : wait() et waitpid()

En se terminant, tout processus passe à l'état **zombie** où il reste aussi longtemps que son père n'a pas pris connaissance de sa terminaison. En effet, dans le noyau Linux, au moment de la terminaison d'un processus, le système désalloue les ressources qu'il possède mais ne détruit pas son bloc de contrôle dans la table de processus. Le système passe le processus dans l'état zombie (représenté généralement par un Z dans la colonne « **statut** » lors du listage des processus par la commande **ps**). Le signal **SIGCHLD** est alors envoyé au processus père afin de l'informer de ce changement. Dès que le processus père a obtenu le code de fin du processus achevé au moyen des primitives **wait()** ou **waitpid()**, le processus terminé est définitivement supprimé de la table des processus. Donc, il est toujours recommandé qu'un processus se termine après la fin de ses processus fils.

La primitive **wait(int *statut)**, de la bibliothèque (**sys/wait.h**), provoque la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.

La primitive **waitpid(int PID, int *statut, 0)**, de la même bibliothèque, permet de suspendre l'exécution du processus appelant jusqu'à la fin de son processus fils dont l'identifiant **PID** est donné en argument.

1.4. Substitution de code : execl()

La fonction **execl()** permet à un processus d'écraser les segments de texte, de données et de pile et de les remplacer par d'autres segments relatif à un autre programme exécutable.
Syntaxe : **int execl(char* ref, char* arg0, char* arg1, ..., NULL);**
execl() lance l'exécution du programme dont la référence est **ref** avec les arguments **arg0**, **arg1**, etc. La liste des arguments se termine par le pointeur **NULL**. La partie du programme qui suit l'appel de la primitive **execl()** ne s'exécute pas, car le segment de texte où elle se trouve est remplacé par un autre.

2. Exercices

Exercice 1

Faite une exécution manuelle du programme suivant :

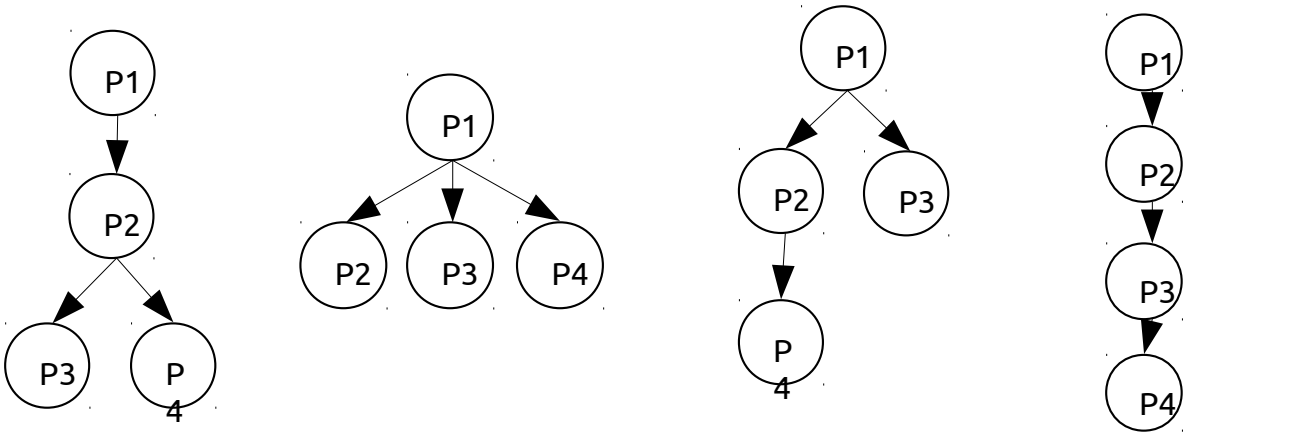
```
int main() {
  int f, i=1;
  f=fork();
  switch(f) {
    case -1 : printf("Erreur dans fork() !!");
      return 1;
    case 0 : /* C'est le processus fils */
      i ++;
      printf ("fils i= %d\n",i);
      break;
    default : /* C'est le processus père */
      printf ("père i = %d\n",i);
      break;
    }
    printf ("fin %d\n",i);
    return 0;
}
```

Exercice 2

Écrire un programme C qui crée deux fils, l'un affiche les entiers de 1 à 5, l'autre affiche les entiers de 6 à 10. Fixer un intervalle d'une seconde entre l'affichage de chaque entier.

Exercice 3

Écrire un programme qui permet de créer chacune des quatre arborescences de processus ci-dessous. Chaque processus doit afficher son nom (indiqué dans la figure), son identifiant et l'identifiant de son père.



Exercice 4

Si tous les appels fork() réussissent, combien de processus le programme suivant crée-t-il ? Dessiner l'arbre des processus engendrés.

```
int main() {
       fork();
       fork();
       fork();
       return 0;
}
```

Exercice 5

Soit le programme suivant :

```
int main() {
  int pid1,pid2;
  printf("Je suis le processus de pid %d et le pid de mon père est %d",getpid(),
getppid());
  pid1=fork();
  if(!pid1)
    printf("Je suis le processus de pid %d et le pid de mon père est %d", getpid(),
getppid());
  pid2=fork();
  if(!pid2)
    printf("Je suis le processus de pid %d et le pid de mon père est %d", getpid(),
getppid());
  return 0;
}
```

1. Combien de processus fils sont engendrés par ce programme ?
2. Représenter l'arborescence des processus.
3. Modifier ce programme pour avoir une arborescence en profondeur.

Exercice 6

1. Écrire un programme C `**processus1.c**' qui permet de générer un affichage semblable au suivant (les valeurs numériques et l'ordre d'affichage peuvent être distincts) :

dans le processus père, valeur de fork = 781
dans le processus fils, valeur de fork = 0
identification du processus fils : 781
identification du père du processus fils : 780
fin du processus fils.
identification du processus père : 780
identification du père du processus père : 677
fin du processus père.

2. Modifier `**processus1.c**' de telle sorte que le processus père se termine avant le processus fils. Qui devient le père du processus fils ?
3. Modifier `**processus1.c**' pour garantir que le processus père se termine après le processus fils.

Exercice 7

Compléter le programme ci-dessous pour que le processus administrateur se termine obligatoirement après :

1. La fin de l'une des deux tâches, principale ou secondaire.
2. La fin de la tâche principale.
3. La fin de la tâche secondaire.
4. La fin des deux tâches, principale et secondaire.

```
int main(){
  int pp, ps;
  printf("Je suis le processus administrateur de PID %d\n", getpid());
  pp=fork();
  switch(pp) {
    case -1: printf("echec du fork\n"); exit(-1);
    case 0: printf("Je suis le processus fils de PID %d qui exécute la tâche
principale\n", getpid());
      sleep(2);//exécution de la tâche principale
      printf("Fin de la tâche principale\n");
      exit(1);
    default: ps=fork();
      switch(ps) {
        case -1: printf("echec de fork"); exit(-2);
        case 0: printf("Je suis le processus fils de PID %d qui exécute la tâche
secondaire\n", getpid());
          sleep(10);// exécution de la tâche secondaire
          printf("Fin de la tâche secondaire\n");
          exit(2);
        default: printf("Je suis le processus administrateur, j'attends la fin
de .................................\n");
          ..............................
          ..............................
          ..............................
          printf("Fin du processus administrateur\n");
      }
   }
   return(0);
}
```

Exercice 8

1. Écrire un programme simple qui permet de créer un processus zombie.
2. Modifier ce programme pour éliminer le processus zombie.

Exercice 9

Écrire un programme dans le quel le processus fils écrase le code hérité de son père par celui de la commande **`ls`**, pour exécuter la commande « **`ls -l /`** » qui liste l'ensemble des fichiers du répertoire racine.