TP 1

Gestion des processus

1 Synchronisation de processus père et fils

Dans cette phase, on va utiliser les fonctions suivantes :

- **`exit( i )`** :

Termine un processus, *i* est un octet (donc valeurs possibles : 0 à 255) renvoyé dans une variable du type **`int`** au processus père.

- **`wait( &Etat )`**

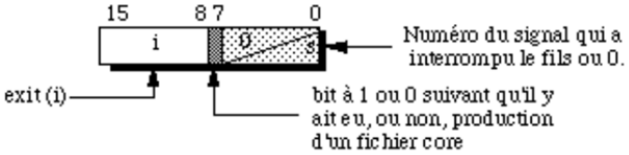
Met le processus en attente de la fin de l'un de ses processus fils. Quand un processus se termine, le signal SIGCHLD est envoyé à son père. La réception de ce signal fait passer le processus père de l'état bloqué à l'état prêt. Le processus père sort donc de la fonction **`wait`**. La valeur de retour de **`wait`** est le numéro du processus fils venant de se terminer.

Lorsqu'il n'y a plus (ou pas) de processus fils dont il faut attendre la fin, la fonction **`wait`** renvoie -1. Chaque fois qu'un fils se termine, le processus père sort de **`wait`**, et il peut consulter **`Etat`** pour obtenir des informations sur le fils qui vient de se terminer.
Etat est un pointeur sur un mot de deux octets :

- L'octet de poids fort contient la valeur renvoyée par le fils (i de la fonction exit(i)),
- l'octet de poids faible :
  - contient 0 dans le cas général,
  - En cas de terminaison anormale du processus fils, cet octet de poids faible contient la valeur du signal reçu par le fils.

Cette valeur est augmentée de 80 en hexadécimal (128 décimal), si ce signal a entraîné la sauvegarde de l'image mémoire du processus dans un fichier core.

Schéma résumant le contenu du mot pointé par **`Etat`** à la sortie de **`wait`** :



EXERCICE 1 *(mécanisme wait/exit)*

1. Compiler et exécuter `sync.c` (en annexe). Analyser et commenter le fonctionnement de ce programme.

2 Père et fils exécutent des programmes différents

Voici la liste des fonctions utilisées pour cette partie :

La fonction **`exec`** charge un fichier dans la zone de code du processus qui l'appelle, remplaçant ainsi le code courant par ce fichier. Une des formes de cette fonction est `execl` :

```
int execl (char *fic, char * arg0, [arg1, ... argn])
```

Commentaires :

- **`fic`** est le nom du fichier exécutable qui sera chargé dans la zone de code du processus qui appelle **`execl`**. Si ce fichier n'est pas dans le répertoire courant, il faut donner son nom complet (chemin absolu).
- Les paramètres suivants sont des pointeurs sur des chaînes de caractères contenant les arguments passés à ce programme (cf. **`argv`** en C).

La convention UNIX impose que le premier soit le nom du fichier lui-même et que le dernier soit un pointeur nul.
Par exemple, si on veut charger le fichier appelé **`prog`** qui se trouve dans le répertoire courant et qui n'utilise aucun argument passé sur la ligne de commande :

```
execl ("prog", "prog", (char *)0)
```

EXERCICE 2

2. Compiler puis exécuter le fichier `fexec.c` (en annexe). Analyser et commenter le fonctionnement de ce programme. On pourra y faire exécuter par `execl` un programme très simple, par exemple celui dont le code source est ci-dessous, comme vous pouvez en proposer d'autres.

```
int main (void) {
   printf("Coucou, ici %d !\n", getpid() );
   sleep (4);
   return 6;
}
```

3 Caractéristiques d'un processus

Dans un système Unix (Linux), plusieurs utilisateurs peuvent travailler sur le même système. Toute activité entreprise par un utilisateur est soumise à des contrôles stricts quant aux permissions qui lui sont attribuées. Chaque processus s'exécute sous une identité précise. Dans la plupart des cas, il s'agit de l'identité de l'utilisateur qui a invoqué le processus (ou UID *réel*, obtenu par la fonction `getuid()`). Toutefois, il existe aussi un UID *effectif* (obtenu par la fonction `geteuid()`) qui correspond aux privilèges accordés à ce processus, et un UID *sauvé* qui est une copie de l'ancien UID effectif lorsque celui-ci est modifié par le processus. Chaque utilisateur appartient à un ou plusieurs groupes. Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé (voir les fonctions `getgid()` et `getegid()`).

Dans l'exercice suivant, vous pouvez utiliser les méthodes suivantes pour récupérer le nom associé à un UID ou à un GID respectivement.

```
struct passwd *getpwuid(uid_t uid);
struct group *getgrgid(gid_t gid);
```

EXERCICE 3

3. Quelle est la différence entre un utilisateur réel et un utilisateur effectif ?
4. Ecrire un programme `users.c` qui permet d'afficher des informations sur les utilisateurs associés à ce programme.

Exemple de résultat :

```
./users
Reel     : aktouf(UID=1000), aktouf(GID=1000)
Effectif : aktouf(UID=1000), aktouf(GID=1000)
```

5. Compiler et exécuter le programme `users` et vérifier si les informations données sont correctes.
6. Utiliser la commande `ls` avec les options `-ln` et `-l` afin de voir l'identité de l'utilisateur associé au programme `users`.
7. Si vous pouvez accéder en tant qu'utilisateur « root », faire les manipulations suivantes :

```
$ su
Password:
# chown root.root users
# chmod +s users
# ls -ln users*
-rwsr-sr-x 1    0    0 5255 Oct  8 14:31 users
-rw-r--r-- 1 1000 1000  405 Oct  8  2012 users.c
# ./users
Reel     : root(UID=0), root(GID=0)
Effectif : root(UID=0), root(GID=0)
# exit
$ ./users
Reel     : debbabi(UID=1000), debbabi(GID=1000)
Effectif : root(UID=0), root(GID=0)
```

4 Exercice

EXERCICE 4

8. En se basant sur les exemples précédents, écrire un programme dont le fonctionnement est le suivant :
   1. il lit sur la ligne de commande (utiliser ***argc*** et ***argv***) le nombre N de processus à créer.
   2. il crée ces N processus en faisant N appels à ***fork*** (cf. plus loin la tâche assignée à ces processus).

   Dans l'annexe, se trouvent un canevas du programme (***nprocess.c***) à écrire ainsi que des indications sur la création des N fils.
   3. il se met en attente (appel à ***ret_fils = wait(&Etat)***) de ces N processus fils et visualise leur identité (***ret_fils*** et valeur de ***Etat***) au fur et à mesure de leurs terminaisons.

   Pour attendre la fin de tous les fils, utiliser le fait que ***wait*** renvoie la valeur `-1` quand il n'y a plus de processus fils à attendre.

Ce que fait chacun des processus fils P_i:

1. il visualise son pid (***getpid***) et celui de son père (***getppid***),
2. il se met en attente pendant `2*i` secondes (***sleep (2*i)***), visualise la fin de l'attente,
3. il se termine par ***exit (i)***.

Compiler en utilisant l'option ***-Wall*** de ***gcc***, il ne doit pas y avoir de ***warnings*** après la compilation.

Annexe

sync.c

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

int
main (void)
{
   int valeur, ret_fils, etat ;
   printf ("Je suis le processus pere de pid %d \n", (int)getpid());
   valeur=fork();
   switch (valeur)
      {
      case 0 :
         printf("\t\t\t\t********\n\t\t\t\t* FILS *    \n\t\t\t\t********\n");
         printf ("\t\t\t\tProcessus fils de pid %d \n\t\t\t\tPere de pid %d \n", (int) getpid(), (int)getppid() );
         printf("\t\t\t\tJe vais dormir 30 secondes ...\n");
         sleep (30);
         printf("\t\t\t\tJe me reveille ,\n\t\t\t\tJe 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 ("Processus pere de pid %d \nFils de pid %d \n", (int) getpid(), valeur );
         printf ("J'attends la fin de mon fils...\n");
         ret_fils = wait (&etat);
         printf ("Mon fils de pid %d est termine,\n Son etat etait : %04x\n",ret_fils, etat);
      }
 return 0;
}
```

fexec.c

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

int
main (int argc, char *argv[])
```

```
{
   int Pid;
   int Fils, Etat;
   /*
      ---------------------------------------------------------
      On peut executer ce programme en lui passant diverses
      commandes en argument, par exemple, si l'executable est fexec :
      fexec /usr/bin/ps
      ---------------------------------------------------------
   */

   if (argc != 2)
   {
      printf(" Utilisation : %s fic. a executer ! \n", argv[0]);
      exit(1);
   }

   printf (" Je suis le processus %d je vais faire fork\n", (int)getpid());
   Pid=fork();
   switch (Pid)
   {
      case  0 :
         printf (" Coucou ! je suis le fils %d\n", (int)getpid());
         printf (" %d : Code  remplace par %s\n", (int)getpid(), argv[1]);
         execl(argv[1], argv[1], NULL);
         printf (" %d : Erreur lors du exec \n", (int)getpid());
         exit (2);
      case  -1 :
         printf ("Le fork n'a pas reussi ");
         exit (3);
      default :
         /* le pere attend la fin du fils */
         printf (" Pere numero %d attend\n ",(int) getpid());
         Fils=wait(&Etat);
         printf ( " Le fils etait : %d ", Fils);
         printf (" ... son etat etait :%04x (hexa) \n",Etat);
         exit(0);
   }
   return 0;
}
```

nprocess.c

```
/* ---------------------------------------
  * Dans main, on indique comment  utiliser
  * les parametres passes sur
  * la ligne de commande
  * ----------------------------------------
  */
int
main (int argc, char *argv[])
{
   int Nbre_de_Proc, i_fils, pid_fils;

   if (argc != 2)    /* On utilise un seul parametre */
   {
      printf(" Utilisation : %s nbre-de-processus ! \n", argv[0]);
      exit(2);
   }
```

```
   Nbre_de_Proc = atoi (argv [1]);  /* conversion ascii -> entier */

   /* ---------------------------------------
    *  creation des  processus fils
    * ---------------------------------------
    */
   for ( i_fils =1; i_fils <= Nbre_de_Proc  ; i_fils++ )
   {
      pid_fils = fork();
      switch (pid_fils)
      {
      case 0 :
         fils(i_fils);       /* il faut ecrire la fonction fils ... */
         break;
      case -1 :
         perror("Le fork n'a pas reussi ");
         exit(33);  /* si erreur -> fin du pere ! */
      }
   }

   /* -----------------------------------------
    *  Dans la fonction pere, on utilisera le
    *  fait que wait renvoie la valeur -1 quand
    *  il n'y a plus de processus fils a attendre.
    * -----------------------------------------
    */
   pere();                 /* il faut aussi ecrire la fonction pere ... */

   return 0;

}
```