Système d'Exploitation Unix

TP

Les appels système : gestion de processus

Tous les programmes devront être développés avec passage de leurs éventuels paramètres à la fonction `main (int argc, char *argv [])`. Les valeurs de retour des appels aux primitives devront être testées et les messages d'erreurs affiches avec `perror`. Les messages d'erreurs à destination de l'utilisateur se feront sur le fichier standard des erreurs `stderr`.

Question 1 Création de processus : la primitive `fork ()`

Soit 2 processus P1 et P2. P1 crée le processus P2 puis exécute une boucle suffisamment longue pour permettre l'observation et dans laquelle il écrit à chaque itération:

`PERE, mon pid est` *`valeur du pid`*
`Le pid de mon fils est` *`valeur du pid`*

P2 dès sa création, exécute une boucle suffisamment longue pour permettre l'observation et dans laquelle il écrit à chaque itération:

`FILS, mon pid est` *`valeur du pid`*
`Le pid de mon pere est` *`valeur du pid`*

Fonctions utiles:

- *fork(): cf. cours.*
- *getpid() et getppid() retournent respectivement le pid du processus le pid de son père.*

NB : on n'utilisera ni le exit ni le wait.

a) Écrire le programme C correspondant (respecter l'indentation des messages de P1 et P2 pour davantage de lisibilité sur l'écran au moment de l'exécution)

b) Dans une autre fenêtre shell, vérifiez la validité des valeurs des pid affichés par les 2 processus à l'aide de la commande ps aux | grep *votre_nom_de_login*.


```
#include <stdio.h>
#include <stdlib.h>

main (int argc, char *argv[]){
 int pid, i;
 int iter_pere,
     iter_fils;

 if (argc != 3){
     fprintf(stderr, "Usage: %s iteration_pere iteration_fils\n", argv[0]);
     exit(1);
 }

 iter_pere=atoi(argv[1]);
 iter_fils=atoi(argv[2]);
 pid=fork ();
 if (pid==-1){
     perror("fork ");
     exit(1);
 }
 if (pid == 0){                  /* CODE DU FILS */
   for (i=1;i<=iter_fils;i++)
     {
      printf ("\t\t\tFILS, mon pid est %d\n", getpid());
      printf ("\t\t\tFILS, le pid de mon pere est %d\n", getppid());
      fflush (stdout);
     }
  }
 else if (pid>0)                  /* CODE DU PERE */
         for (i=1;i<=iter_pere;i++){
             printf ("PERE, mon pid est %d\n", getpid());
             printf ("PERE, le pid de mon fils est %d\n", pid);
         }
}
```

Question 2 Vie des processus

a) Lancez plusieurs fois le programme précédent. Quelles séquences d'exécution observez-vous, que faut-il en déduire ?

b) Ajustez le nombre d'itérations du processus père P1 de telle sorte qu'il se termine *AVANT* son fils P2. Que constatez-vous ?

c) Ajustez le nombre d'itérations du processus fils P2 de telle sorte qu'il se termine *AVANT* son père P1. Dans quel état se trouve P2 du point de vue du système ?

Question 3 Effet du « clonage » sur les données

Le processus fils hérite d'une copie des données du père. Cette copie n'est réalisée que lors d'accès en écriture (copy on write) et est telle que :

- l'ensemble des adresses valides des deux processus est le même,
- les valeurs des données dans les deux processus sont les mêmes au retour de l'appel du fork(),
- les données physiques sont différentes

a) Ecrire le programme suivant de façon à vérifier les affirmations précédentes :
Soit 2 processus P1 et P2. P1 crée le processus P2 PUIS exécute une boucle (suffisamment longue pour permettre l'observation) dans laquelle il décrémente à chaque itération une variable entière X initialisée avec une valeur quelconque:
PERE *valeur de X*

P2 dès sa création, exécute une boucle (nombre d'itérations égal à celui du père) dans laquelle il écrit à chaque itération:
FILS, *valeur de X*

P1 et P2 affiche l'adresse de X

```
#include <stdio.h>
#include <stdlib.h>

int pid, x;

main (int argc, char *argv[]) {
    x=1;
    pid=fork();
    if (pid==-1){
          perror ("pb fork ");
          exit (1);
    }

    if (pid==0){
          printf("\t\tfils - adresse de x: %p\n", &x);
          printf("\t\tfils - valeur de x apres fork: %d\n",x);
          x=x+1;
          printf("\t\tfils - valeur de x=x+1: %d\n", x);
          printf("\t\tfils - adresse de x: %p\n", &x);
    } else{
          printf("pere - adresse de x: %p\n", &x);
          printf("pere - valeur de x apres fork: %d\n", x);
          x=x+2;
          printf("pere - valeur de x=x+2: %d\n", x);
          printf("pere - adresse de x: %p\n", &x);
    }
}
```

Question 4 La famille processus

Un processus P crée un fils F1 et un fils F2. F2 crée 1 fils F3.
F2 affiche :
"Je suis *w* : F2, le frère de *x:* F1"
F3 affiche :
"Je suis *y* : F3, le petit-fils de *p* : P et le neveu de *x:* F1"

a) Écrire en C le programme correspondant en sachant que *w* est la valeur du pid de F2, *x* est la valeur du pid de F1, *y* est la valeur du pid de F3 et *p* la valeur du pid de P.


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

main(){
 int w,// PID de F2
     y,// PID de F3
     x,// PID de F1
     p;// PID de P;
 int pid;

 p=getpid();
 x=fork(); // CODE de P
 if (x==0){/* CODE DE F1 */ }

 else { // P
       pid=fork();
       if (pid==0){/* CODE DE F2 */
          w=getpid();
          printf("Je suis %d: F2, le frere de %d: F1\n", w, x);

          pid=fork();
          if (pid==0){/* CODE DE F3 */
             y=getpid();
             printf("Je suis %d: F3, le petit-fils de %d: P et le neveu
de %d\n", y, p, x);
          } /* FIN CODE DE F3 */
       } /* FIN CODE DE F2 */
     } /* FIN CODE DE P */
}/* main */

#include <stdio.h>b

main(){
 int w, y, x, p;
 int pid;

 p=getpid();

 if (x=fork()==0){/* CODE DE F1 */ }
 else if (pid=fork()==0){/* CODE DE F2 */
          w=getpid();
          printf("Je suis %d: F2, le frere de %d: F1\n", w, x);

          if (pid=fork()==0){/* CODE DE F3 */
             y=getpid();
             printf("Je suis %d: F3, le petit-fils de %d: P et le neveu de %d\n", y,
p, x);
          } /* if (pid=fork()==0) CODE DE F3 */
      } /* if (pid=fork()==0) CODE DE F2 */
}/* main */
```

b) On voudrait aussi que F1 affiche : "Je suis *x:* F1, le frère de *w* : F2". Quel problème cela pose-t-il ?

Question 5 La synchronisation père/fils

Soit un processus P1 qui crée **d'abord** n (n<5) processus et **ensuite** attend qu'ils terminent.

a) Chaque P_n entre dans une boucle (suffisamment longue pour permettre l'observation) puis se termine par exit (n). Chaque fois qu'un de ces fils se termine, P1 affiche :

- le pid du fils terminé,
- la valeur transmise par le exit du fils qui se termine.
- le nombre de fils restant à attendre

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
main (int argc, char *argv[]){
int pid, i;
int code_retour, nbre_fils, dors_fils;

 if (argc != 3){
     fprintf(stderr,"Usage: %s nombre de fils duree_sommeil_fils\n",
                                                 argv[0]);

     exit(1);
 }
 nbre_fils=atoi(argv[1]);
 dors_fils=atoi(argv[2]);

 for (i=1; i<= nbre_fils; i++){
    pid=fork ();
    if (pid==-1){
           perror("fork ");
           exit(1);
    }
    if (pid == 0){                  /* CODE DU FILS */
           printf("\nFils: mon PID est %d, je m endors pour %d s\n",
getpid(),dors_fils);
           sleep(dors_fils);
           exit (i);
    }
 } // for

 /* ATTENTE DES FILS */
 for (i=nbre_fils; i >= 1; i--){
         pid=wait(&code_retour);
         if (pid==-1){
            perror ("wait: ");
            exit (1);
         }
         printf("\nPere: mon fils %d s'est termine. \n", pid);
         printf("     : sa valeur de retour est %d\n", code_retour);
            printf("Encore %d fils\n", i-1);
  } // for
} // main
```

b) Modifier le programme précédent de façon que le processus père indique également si la terminaison de son fils s'est effectuée :

- normalement et affiche la valeur transmise par le `exit` du fils qui se termine.
- suite à la réception d'un signal non intercepté et affiche le n° du signal en cause


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


main (int argc, char *argv[]){
 int pid, i;
 int code_retour,
    nbre_fils,
    dors_fils;

 if (argc != 3){
     fprintf(stderr, "Usage: %s nombre de fils  duree_sommeil_fils\n", argv[0]);
     exit(1);
 }

 nbre_fils=atoi(argv[1]);
 dors_fils=atoi(argv[2]);

 for (i=1; i<= nbre_fils; i++){
    pid=fork ();
    if (pid==-1){
           perror("fork ");
           exit(1);
    }
    if (pid == 0){                  /* CODE DU FILS */
           printf("\nFils: mon PID est %d, je m endors pour %d s\n",
getpid(),dors_fils);
           sleep(dors_fils);
           exit (i);
    }
 } // for

 /* ATTENTE DES FILS */
 for (i=nbre_fils; i >= 1; i--){

         pid=wait(&code_retour);
         if (pid==-1){
            perror ("wait: ");
            exit (1);
         }

         printf("\nPere: mon fils %d s'est termine. \n", pid);

         if (WIFEXITED(code_retour)){
            printf("      normalement et m'a retourner %d \n",
WEXITSTATUS(code_retour));
            //exit (0);
         }

         if (WIFSIGNALED(code_retour)){
            printf("      de facon anormale\n");
            printf("      il a recu le signal %d\n",WTERMSIG(code_retour));
            //exit (0);
         }
           printf("Encore %d fils\n", i-1);

  } // for
} // main
```

Fonctions utiles:

fork, wait, exit: cf. cours et le man

Question 6 Terminaison des processus

Les primitives `exit()` et `_exit()` entraînent la terminaison d'un processus, à la demande de ce dernier.

`exit()` diffère de `_exit()`, entre autres, par le fait qu'elle vide les tampons associés aux fichiers ouverts.

a) Vérifier, par l'écriture d'un programme, l'affirmation précédente.

`exit()` diffère également de `_exit()`par le fait qu'elle appelle toutes les fonctions dont la demande d'exécution en cas de terminaison a été formulée au moyen de la fonction standard `int atexit(void (*fonction)(void));`.

b) Ecrire un programme qui formule la demande d'exécution de trois fonctions (func1(), func2() et func3()) en cas de terminaison. Le processus doit se terminer sur un appel à `exit()`.

c) Vérifier, en remplaçant `exit()` par `_exit()`dans le programme précédent, que les fonctions (func1(), func2() et func3()) ne sont pas exécutées à la terminaison du processus

d)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void fin_A (){
    printf(" FONCTION A \n");
}

void fin_B (){
    printf(" FONCTION B \n");
}


int main(int argc, char *argv[]){
    atexit(fin_A);
    atexit(fin_B);

    printf(" PROGRAMME TERMINE "); // pas de \n donc pas de vidage de stdout

    // avec exit les buffers des fichiers ouvert sont vidés et appelle
    // les fonctions dont les demandes d'exécution cas de terminaisons
    // ont été formulées avec atexit
    // _exit ne fait rien de tout ça, elle met simplement fin au processus

       //exit(EXIT_SUCCESS);
        _exit(EXIT_SUCCESS);

}// main
```

Question 7 Les temps CPU

La primitive `times()` renvoie les temps CPU consommés dans les deux modes (utilisateur et noyau) par le processus ainsi que par ses fils terminés ayant été attendus.

Les temps retournés dans les différents champs de la structure tms sont exprimés en nombre de clics d'horloge. Pour les convertir en des durées exprimées en secondes, il suffit de les diviser par le nombre de clics d'horloge par seconde obtenu avec `sysconf (_SC_CLK_TCK)`.

Ecrire un programme qui :

1) affiche le nombre de clics écoulés sur la machine et le nombre clics par seconde
2) effectue un certain calcul assez long (> 1 seconde)
3) affiche les temps CPU utilisateur et système consommés

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

clock_t nb_tick;

struct tms info;
int i, p;
float u, s, tick_sec;


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

    if (argc !=2){
          fprintf (stderr, "Usage %s: nb iterations (>30 000 000)\n", argv[0]);
          exit (1);
    }

    for (i=1,p=1; i<=atoi(argv[1]); i++, p++){
          i=i*1;
          if (p==100){
               //printf ("%d\n", i);
               p=0;
          }
    } // for

    printf ("\n");
    nb_tick=times (&info);
    if (nb_tick==-1){
          perror ("pb times ");
          exit (1);
    }

    printf("nb ticks écoulé : %d - nb tick/sec: %d\n", nb_tick, sysconf(_SC_CLK_TCK));

    u=info.tms_utime;
    s=info.tms_stime;
    tick_sec=sysconf(_SC_CLK_TCK);

    printf("tms_utime: %f s \n", u/tick_sec);
    printf("tms_stime: %f s \n", s/tick_sec);


} //main
```

Question 8 Les groupes de processus

La primitive `pid_t getpgrp(void)` renvoie le PID du leader du groupe de l'appelant.

a) Ecrire un programme qui affiche le PID du processus leader du groupe auquel il appartient.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pid_t mon_groupe, mon_leader;

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

    mon_leader = getpgrp();

    if (mon_leader==-1){
          perror ("pb times ");
          exit (1);
    }

    printf("Mon PID est: %d \n", getpid());
    printf("Le processus leader de mon groupe est: %d \n", mon_leader);

} //main
```

Un processus appartient au même groupe que celui auquel appartient son père et ceci jusqu'à ce qu'il demande à en changer.

b) Modifier le programme précédent de façon qu'il crée un processus fils qui affiche aussi le PID du processus leader du groupe auquel il appartient.

c) Vérifier si les deux processus précédents appartiennent au même groupe ?

d) Appartiennent-ils au même groupe auquel appartient le shell à partir duquel vous avez demandé l'exécution (`ps j`) ?


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pid_t mon_groupe,
          mon_leader,
          pid
          ;

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

    pid=fork();
    if(pid==-1) {
          perror("Erreur du fork");
          exit (1);
    }

    if(pid==0) { // Fils ===================

          mon_leader = getpgrp();

          if (mon_leader==-1){
                perror ("Erreur getpgrp fils");
                exit (1);
          }

          printf("FILS, mon PID est: %d \n", getpid());
          printf("FILS, Le processus leader de mon groupe est: %d \n\n", mon_leader);
    }
    else { // Père ======================
          mon_groupe = getpgrp();

          if (mon_groupe==-1){
                perror ("Erreur getpgrp père ");
                exit (2);
          }
          printf("PERE, mon PID est: %d \n", getpid());
          printf("PERE, Le processus leader de mon groupe est: %d \n\n", mon_groupe);
    }

} //main
```

Lorsqu'un processus leader d'un groupe de processus se termine, un signal `SIGHUP` est émis à chaque processus stoppé membre du groupe.

e) Modifier le programme précédent de façon que le processus fils ne se termine pas immédiatement (utiliser `pause()` à la fin du code du fils) et mettez le à l'état "stoppé".

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

pid_t mon_groupe, mon_leader, pid;

main (int argc, char *argv[]) {
    pid=fork();
    if(pid==-1) {
           perror("Erreur du fork");
           exit (1);
    }

    if(pid==0) { // Fils ==========================
       titre="FILS";

           mon_leader = getpgrp();

           if (mon_leader==-1){
                 perror ("Erreur getpgrp fils");
                 exit (1);
           }

           printf("FILS, mon PID est: %d \n", getpid());
           printf("FILS, Le processus leader de mon groupe est: %d \n\n", mon_leader);

           // Au terminal, Envoyer SIGSTOP au fils pendant qu'il est en pause puis
           // observer ce qu'il devient quand leader de son groupe se termine
           pause();
           printf("FILS terminé \n");
    }
    else { // Père ================================
           titre="PERE";

           mon_groupe = getpgrp();

           if (mon_groupe==-1){
                 perror ("Erreur getpgrp père ");
                 exit (2);
           }

           printf("PERE, mon PID est: %d \n", getpid());
           printf("PERE, Le processus leader de mon groupe est: %d \n\n", mon_groupe);
           sleep(30);
           printf("PERE, Terminé \n");
    }

} //main
```

f) Que devient le processus fils une fois que le processus père (leader de son groupe) s'est terminé ?

La primitive `pid_t setpgid(pid_t pid, pid_t pgid)` rattache le processus `pid` au groupe `pgid`.

g) Modifier le programme précédent de façon que le processus fils change de groupe par rapport à son père.


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>


pid_t mon_groupe, mon_leader, pid;

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

    pid=fork();
    if(pid==-1) {
           perror("Erreur du fork");
           exit (1);
    }

    if(pid==0) { // Fils ==========================
       titre="FILS";

       // Le fils quitte le groupe de son père en
       // créant son propre groupe dont il devient le leader

        printf("FILS: Je quitte le groupe %d et je cree mon propre groupe\n",
                                                                    getpgrp());
        if (setpgid(0,0)== -1){
                 perror ("Erreur setpgid fils");
                 exit (1);       }

           mon_leader = getpgrp();

           if (mon_leader==-1){
                 perror ("Erreur getpgrp fils");
                 exit (2);
           }

           printf("FILS, mon PID est: %d \n", getpid());
           printf("FILS, Le processus leader de mon groupe est: %d \n\n", mon_leader);

           // Au terminal, envoyer SIGSTOP au fils pendant qu'il est en pause puis
           // observer ce qu'il devient quand leader de son groupe se termine
           pause();
           printf("FILS terminé \n");
    }
    else { // Père ================================
           titre="PERE";

           mon_groupe = getpgrp();

           if (mon_groupe==-1){
                 perror ("Erreur getpgrp père ");
                 exit (3);
           }

           printf("PERE, mon PID est: %d \n", getpid());
           printf("PERE, Le processus leader de mon groupe est: %d \n\n", mon_groupe);
           sleep(30);
           printf("PERE, Terminé \n");
    }

} //main
```

h) Que devient le processus fils une fois que le processus père (leader de son groupe) s'est terminé ?