TP2 - Gestion et Ordonnancement des Processus

1. Création de processus et exécution concurrente; synchronisation père-fils; recouvrement

Petit rappel: pour compiler les programmes (dans la plupart des cas):

```
gcc -Wall -o nom_executable source.c
```

1. Exécuter le programme suivant avec les valeurs 1, 2, et 3 de SP. Observez le résultat obtenu dans chaque exécution concurrente. Exécuter le programme plusieurs fois et observer l'ordre des sorties.

```
#include <unistd.h>
#include <sys/types.h>
#define SP 2 /*changer pour 1, 2, 3 .... */

int main ( void ) {
      char mes[] = "ABCDEFGHIJ" ;
      char *ptr;
      pid_t n ;

      ptr = mes ;
      n = fork() ;

      while ( *ptr != '\0'){
            /*on parcourt mes[] caractère par caractère*/
            write(STDOUT_FILENO,ptr,1);
             ptr++ ;
            if ( n == 0 ) sleep(1);
            else sleep(SP) ;
      }

      return 0;
}
```

1.1 Combien de processus sont actives pendant l'exécution? Vérifiez avec l'aide de `ps` (Vous pouvez bloquer l'exécution avec Ctrl-Z et la reprendre avec `fg`).

/******* Exemples d'executions *******
Pour simplifier, j'ai mis SP en argument sur la ligne de commande Voici les résultats :

[pekergin]$ aff_melange2 1
AABBCCDDEEFFGGHHIIJJ
[pekergin]$ aff_melange2 2
AABBCDCEFDGHEIJFGHIJ
[pekergin]$ aff_melange2 3
AABCBDEFCGHIDJEFGHIJ

Exemple de ps pendant la suspension:
[pekergin]$ ps

PID TTY	TIME CMD
3689 pts/0	00:00:00 bash
3812 pts/0	00:00:00 melange
3813 pts/0	00:00:00 melange
3815 pts/0	00:00:00 ps

L'intérêt de l'exercice est de voir qu'il y a DEUX PROCS et l'exécution est concurrente!

1.2 Comment afficher le pid du chaque processus? Modifiez le programme pour afficher le pid du processus avant d'afficher chaque lettre du message mes[]. Vérifiez si les pid correspondent à ceux que vous pouvez trouver avec ps.
pour le pid, il faut utiliser getpid. Modification triviale mais il peuvent découvrir (s'ils ne la connaissent déjà) la différence entre stdout et stderr.

1.3 Modifiez le programme pour avoir deux fils au lieux d'un seul.
Erreur classique: introduire un deuxième fork(). Il faut le mettre dans un if pour vérifier qu'on est pas dans le fils: par exemple, if(n>0) n=fork();
Cela nous mène vers l'exo 2:

2. On donne le programme suivant:

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

```
#include <sys/types.h>
#define N ...

void main () {
  int i ;
  pid_t pid, ppid, ret_fork ;

  pid = getpid() ;
  printf(" Debut du processus no : %d \n", pid ) ;

  for ( i = 1 ; i < N ; i++ ) {
    ret_fork = fork () ;
    pid = getpid() ;
    ppid = getppid() ;
      printf(" Processus %d dont le père est %d , s'exécute ... \n", pid,
ppid);
    //if ( ret_fork == 0 ) break ;
    // Question 2.1) Evite les petits-fils …
  }
  printf(" \t \t Fin du processus no : %d \n", pid ) ;
}
```

Combien de processus sont créés pour N=2, N=3 et N=4 ?

N	Nombre de proc (père inclus)
2	2
3	4
4	8

En général

n 2 ^(N-1)

2.1 Modifier le programme pour que seulement 3 processus fils soient créés par le processus initial : aucun processus petit-fils ou arrière petit-fils ne doit être créé.

voir code

3. D'après l'observation des résultats du programme suivant, qu'est ce que l'on peut déduire à propos de l'héritage des variables? Peut-on passer des informations du père au fils et du fils au père?

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

int i = 3 ;
int main ( void ) {
      pid_t idf ;

      printf(" Avant fork : i = %d \n",i);
      idf = fork() ;
      if ( idf == 0 ) {
            printf("\t Dans le FILS : i = %d \n",i);
            i++;
            printf("\t Dans le FILS après la MODIF : i = %d \n",i);
      } else {
            printf(" Dans le PERE : i = %d \n",i);
            i--;
            printf(" Dans le PERE après la MODIF : i = %d \n",i);
      }
}
```

L'image du processus initial qui fait un fork est dupliquée : le fils commence son exécution juste à la suite de fork qui lui retourne la valeur 0 tandis que le processus initial, que nous appelons désormais père, reçoit le pid du fils. Ainsi, le fils hérite les variables du père (du proc initial) et les deux processus continuent leur exécution chacun avec son image : la modif de i dans le père n'affecte pas la variable i du fils et vice-versa. A partir du fork, il ne peut avoir de passage d'information ni du père au fils, ni du fils au père (à moins d'utiliser les mécanismes de communication entre les processus, qui existent dans le système et qu'on verra plus tard).

4. Téléchargez le programme suivant (fichier http://lipn.fr/~buscaldi/exoTP2_3.c):

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#define MAX 10
```

```
void init_tab(int n, int* a) {
    int i;
    fprintf(stderr, "tableau:\n");
    for(i=0; i< n; i++) {
       int r = rand() % MAX;
       a[i]= r;
       fprintf(stderr, "%d ", r);
    }
    fprintf(stderr, "\n");
}

/* fonction qui prend un tableau, une valeur à chercher
et les limites de la zone dans le tableau où chercher la
valeur */
int cherche(int* a, int val, int debut, int fin) {
    if(debut > fin || fin < debut) {
       fprintf(stderr, "erreur des limites de recherche");
       return 0;
    }

    int i;
    for(i=debut; i< fin; i++) {
       int tmp=a[i];
       if (val==tmp) {
             return 1;
       }
    }
    return 0; //on a pas trouvé val
}

int main ( int argc, char *argv[] ) {
    int n=0;
    int val;

    clock_t start, end;

    if ( argc != 3 ) {
       printf( "usage: %s taille_tableau valeur_a_chercher \n", argv[0] );
    } else {
       n=atoi(argv[1]);
       val=atoi(argv[2]);
    }
    srand(time(NULL)); //Initialisation du générateur de nombres aléatoires
```

```
    int a[n];
    init_tab(n, a);

    int trouve;

    /* Ici on cherche dans le tableau entier */
    /* bloc à remplacer */
    start = clock();
    trouve=cherche(a, val, 0, n);
    end= clock();

    if (trouve) fprintf(stdout, "%d est dans le tableau\n", val);

         fprintf(stdout, "\nrecherche finalisée en %f millisecondes\n",
(end-start)*1000/(double)(CLOCKS_PER_SEC));
    /* fin bloc à remplacer */

}
```

Le programme prend en paramètre deux valeurs: la taille d'un tableau qui est rempli aléatoirement par *init_tab* avec des valeurs entre 0 et 10, et une valeur à chercher dans le tableau. Le programme écrit un message si la valeur est contenue dans le tableau.

4.1 La fonction *cherche* prend en paramètres le tableau, la valeur à chercher, et deux limites: la position du tableau où commencer à chercher et celle où arrêter de chercher.
Modifiez le programme pour utiliser deux processus: le père recherche dans la première moitié du tableau, le fils dans la seconde (suggestion: fork, wait).

Une possible solution:

```
pid_t pid=fork();
start = clock();

int h=n/2; /* la moitié du tableau */

if(pid > 0) {
        //PERE
        trouve=cherche(a, val, 0, h);
        if (trouve) fprintf(stdout, "%d est dans le tableau\n", val);
        wait(NULL);
        end= clock();
```

```
            fprintf(stdout, "\nrecherche finalisée en %f millisecondes\n",
(end-start)*1000/(double)(CLOCKS_PER_SEC));
        exit(0);
  } else if (pid==0) {
        //FILS
        trouve=cherche(a, val, h+1, n);
        if (trouve) fprintf(stdout, "%d est dans le tableau\n", val);
        exit(0)  ;  //Le fils se termine.
  }
```

5. Exécuter le programme rec1.c donné ci-dessous et comprendre le résultat. Il y a eu recouvrement ou pas? Pourquoi? En cas négatif, corriger le programme afin que le recouvrement se déroule correctement.

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

int main ( int argc, char * argv[] )
{
      printf (" Programme %s d'identité %d s'exécute...\n", argv[0],
      getpid());

      execl("ps","ps",NULL);

      printf(" j'existe donc exec dans %s a echoué !...\n", argv[0]);
}
```

Les instructions se trouvant après un exec ne peuvent s'exécuter que si cet exec echoue, donc il n'y a pas eu de recouvrement. Pour corriger le programme, il suffit de remplacer execl avec execlp (execl n'utilise pas le PATH), ou remplacer le "ps" en premier paramètre de la execl avec son chemin absolu ("/bin/ps").

6 Ecrire un programme en langage C, qui crée un fils. Le fils affiche 2 fois son identité et celle de son père avec un intervalle de temps de *sf* secondes. Le père se suspend en appelant *sleep(sp)*, puis effectue un recouvrement par *execl("/bin/ps","ps","t","pts/x",0)* où x est le numéro du terminal que vous utilisez pour lancer ce programme. (La commande tty affiche pour chaque fenêtre le nom du pseudo terminal qui lui est associé.)

6.1. Exécutez votre programme, notez l'identité du père affichée et l'état du fils pour
sf = 2 , sp = 4 , ensuite pour
sf = 4 , sp = 2.
Que peut-on déduire sur le comportement du système quand le père se termine avant le fils?

```
#include <unistd.h>

main()
{
pid_t n ;
int   sf, sp ;

printf("Entrer sf et sp : \n");
scanf("%d %d", &sf, &sp);

n = fork() ;

if ( n == 0 )
      {
      printf(" Fils commence: proc %d de pere = %d \n ",
                getpid(), getppid());
      sleep(sf);

      printf(" Fils termine : proc %d de pere = %d \n ",
                getpid(), getppid());
      exit(1);
      }

else { sleep(sp);
      printf("Pere se transforme ...\n ");
      execl("/bin/ps", "ps","tp4",0);
      exit(0);
      }
}

      /****** EXEMPLE d'EXECUTION *******/
[ferhan@localhost tp12]$ exo1.x
Entrer sf et sp :
2 4
Fils commence: proc 2105 de pere = 2104
Fils termine : proc 2105 de pere = 2104
Pere se transforme ...
PID TTY STAT TIME COMMAND
1332 p4 S 0:00 -bin/tcsh
2104 p4 R 0:00 ps tp4
2105 p4 Z 0:00 (exo1.x <zombie>)
[ferhan@localhost tp12]$ !!
exo1.x
```

```
Entrer sf et sp :
4 2
Fils commence: proc 2107 de pere = 2106
Pere se transforme ...
PID TTY STAT TIME COMMAND
1332 p4 S 0:00 -bin/tcsh
2106 p4 R 0:00 ps tp4
2107 p4 S 0:00 exo1.x
[ferhan@localhost tp12]$ Fils termine : proc 2107 de pere = 1
[ferhan@localhost tp12]$
```

6.2 En utilisant la primitive ***wait( )***, transformer votre programme tel que le père effectue le recouvrement après la terminaison de son fils, indépendamment des durées ***sp*** et ***sf*** .
Dans ce cas, a-t-on toujours besoin de ***sleep( )*** dans le code du père ?
Laisser ou supprimer la primitive ***sleep( )*** dans le code du père conduit-il a une différence de temps d'exécution total ?
Avec les primitives que vous connaissez, est-il possible de garantir que le fils se termine après le père ?

Il suffit de remplacer *sleep(sp)* par wait(0) pour faire attendre le père jusqu'à la fin du fils.
Comme la réponse l'indique, il n'est pas nécessaire de garder la primitive sleep() . En outre, ne pas la supprimer provoquera un délai supplémentaire pour la terminaison si la durée de sommeil ou de suspension pour le père n'est pas déjà expirée au moment de la terminaison du fils : par exemple avec sf = 2 et sp = 4, on attendra approximativement 2 secondes de plus pour que le père continue et on aura autant de temps perdu sur le temps de terminaison global.

Il est bien connu que les primitives concernant les procs permettent que le père attend la fin du fils mais pas l'inverse. Si un tel comportement est à mettre en œuvre (signalons en passant que l'intérêt de ceci n'est pas clair et par conséquent il n'est pas très conforme à l'esprit du système) on dispose plein de moyens pour assurer une telle synchronisation en commençant par les signaux, les tubes, les verrous sur fichiers, les sémaphores etc… En principe toute opération bloquante dont l'événement libératoire peut être produit par un autre processus peut être utilisée dans ce but y compris le couple [wait(), kill()] : le fils devra créer un fils à son tour que le père tuera par sa dernière action de sa vie. Evidemment ce schéma n'est pas facile à mettre en œuvre :
Le père devra obtenir la liste des procs et identifier le pauvre petit-fils à éliminer…
Laissant de côté cette discussion quelque peu philosophique, il suffit de préciser que le wait ne fonctionne que dans un sens…

2. Ordonnancement

1. Téléchargez le fichier http://lipn.fr/~buscaldi/ordonnanceur.tar.gz . Décomprimez le contenu et compilez-le avec **make**. Le programme **ordsim** est un simulateur d'ordonnancement. Il est lancé avec la syntaxe:

```
$ ordsim -i data.txt -s sjf,fcfs,srt,rr -v -q n
```

Où data.txt est un fichier qui contient, ligne par ligne, deux (ou trois) valeurs séparés par des virgules:
temps d'arrivée du processus, taille du processus et (optionel) priorité du processus. Par exemple, dans le dossier "data", il y a le fichier "cours.txt" qui contient une codification de l'exemple vu en cours:

```
0,8
1,4
2,2
3,5
```

L'option -s permet de choisir entre des algorithmes d'ordonnancement différents:
- fcfs: First Come, First Served
- sjf: Shortest Job First
- srt: Shortest Remaining Time
- rr: Round Robin (Tourniquet)
- unix: Ordonnanceur POSIX (PS)

Dans le cas de choisir rr, on peut spécifier avec "-q n" un quantum de taille n. L'option -v montre le détail de l'ordonnancement à chaque cycle d'horloge.

Testez les différents algorithmes avec les processus dans "data/cours.txt" et vérifiez les résultats de l'ordonnancement avec des paramètres différentes (note: les résultats sont différents de ceux vu dans le cours parce que le simulateur prend en compte le temps nécessaire pour chaque changement de contexte). Quel algorithme semble permettre d'obtenir les meilleures performances?

Ils doivent trouver que SRT donne les meilleurs en termes de temps de rotation, par ailleurs RR occupe moins la UC

2. Utilisez l'algorithme du tourniquet avec des quantums différents (q=1, q=2, q=3, q=4). Avec quel quantum les performance sont optimales?

Ils doivent voir que pour des petit q il y a trop d'overhead, par contre si on augmente q la performance s'aligne sur FCFS

3. Modifiez les données, introduisant un nouveau processus de taille 9 à t=2 et un nouveau processus de taille 4 à t=6, et des priorités pour chaque processus (priorité 20 pour les vieux processus, priorité 50 pour les nouveaux); il faut introduire une troisième colonne dans le fichier des données pour spécifier les priorités. Vérifiez comment le résultat changent en modifiant les priorités si on tilise l’algorithme du tourniquet.

4. Soient les données d'ordonnancement suivantes :

Processus	Date d’arrivée	Temps de traitement
A	0	5
B	1	2
C	2	5
D	3	3

Quel est le déroulement de l'exécution des processus pour l'algorithme SRT ? Même question pour l'algorithme du tourniquet.

SRT:
t=0 UC:A_5
t=1 UC:B_2 Queue:A_4
t=2 UC:B_1 Queue:A_4 C_5
t=3 UC:D_3 Queue: A_4 C_5
t=4 UC:D_2 Queue: A_4 C_5
t=5 UC:D_1 Queue: A_4 C_5
t=6 UC:A_4 Queue: C_5
t=7 UC:A_3 Queue: C_5
t=8 UC:A_2 Queue: C_5
t=9 UC:A_1 Queue: C_5
t=10 UC:C_5
t=11 UC:C_4
t=12 UC:C_3
t=13 UC:C_2
t=14 UC:C_1

RR:
t=0 UC:A_5
t=1 UC:B_2 Queue:A_4
t=2 UC:A_4 Queue:B_1 C_5
t=3 UC:B_1 Queue: C_5 A_3 D_3

t=4 UC:C_5 Queue: A_3 D_3
t=5 UC:A_3 Queue: D_3 C_4
t=6 UC:D_3 Queue: C_4 A_2
t=7 UC:C_4 Queue: A_2 D_2
t=8 UC:A_2 Queue: D_2 C_3
t=9 UC:D_2 Queue: C_3 A_1
t=10 UC:C_3 Queue: A_1 D_1
t=11 UC:A_1 Queue: D_1 C_2
t=12 UC:D_1 Queue: C_2
t=13 UC:C_2
t=14 UC:C_1

4.1 Le temps de rotation de chaque processus peut être calculé en soustrayant la date à laquelle le processus a été introduit dans le système à la date à laquelle celui-ci a pris fin. Calculez les temps de rotation des 4 processus précédents et pour les deux algorithmes SRT et RR.

SRT:
A -> 10-0 = 10
B -> 4-1 = 3
C -> 15-2 = 13
D -> 6-3 = 3

RR:
A -> 12-0 = 10
B -> 4-1 = 3
C -> 15-2 = 13
D -> 13-3 = 10

4.2 Le temps de rotation moyen est alors calculé en faisant la somme des temps de rotation et en divisant par le nombre de processus concernés. Calculez les deux temps moyen.

SRT:
(10+3+13+3)/4 = 7,25

RR:
(10+3+13+10)/4 = 9

5. Écrire un programme C qui retourne sa propre priorité (getpriority).

```
#include <sys/resource.h>

int main ( int argc, char * argv[] )
{
        printf (" Programme %s d'identité %d...\n", argv[0], getpid());


        int pri = getpriority(PRIO_PROCESS, getpid());

        printf("Priorité statique: %d\n", pri);
}
```