TP2 - Gestion et Ordonnancement des Processus

A. 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 mesg[] = "ABCDEFGHIJ" ;
      char *ptr;
      pid_t n;

      ptr = mesg ;
      n = fork() ;

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

[1] Pour visualiser si c'est le père ou le fils à écrire les caractères:

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

...
/*dans le while:*/

char s; / chaine à imprimer */
s = malloc(4); /* éviter le segfault */
if (n==0) sprintf(s,"f:");
else sprintf(s, "p:");
strcat(s, ptr); /* on rajoute ptr */

write(STDOUT_FILENO,s,3);
```

```
	}

	return 0;
}
```

1.1 Combien de processus sont actifs pendant l'exécution? Vérifiez à l'aide de `ps` (Vous pouvez bloquer l'exécution avec `Ctrl-Z` et la reprendre avec `fg`).
1.2 Modifiez le programme pour afficher le pid du processus avant d'afficher chaque lettre du message **`mesg[]`** . Vérifiez si les pid correspondent à ceux que vous trouvez avec ps.
1.3 Modifiez le programme pour avoir deux fils au lieux d'un seul.

2. On donne le programme suivant:

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#define N ...mettre un chiffre ici...

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

	pid = getpid() ;
	printf("Debut du processus %d\n", pid ) ;
	for ( i = 1 ; i < N ; i++ ) {
		ret_fork = fork () ;
		pid = getpid() ;
		ppid = getppid() ;
		printf ("Le processus %d dont le père est %d , s'exécute … \n",
				pid, ppid) ;
	}
	printf("\t\t Fin du processus %d \n", pid ) ;
	return(0);
}
```

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

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.

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 ? 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);
      }
      return (0);
}
```

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 */
    start = clock();
    trouve=cherche(a, val, 0, n);
```

```
    end= clock();

    if (trouve!=0) fprintf(stdout, "%d est dans le tableau\n", val);

        fprintf(stdout, "\nrecherche finalisée en %lf millisecondes\n",
(end-start)*1000/(double)(CLOCKS_PER_SEC));

    return 0;
}
```

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).

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 échoué !...\n", argv[0]);
}
```

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 ?

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 ?

B. Ordonnancement

1. Téléchargez le fichier http://lipn.fr/~buscaldi/ordonnanceur.tar.gz . Décompressez le contenu puis 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 priorité du processus (optionnel). 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 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 données de "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 ?

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

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 évoluent les résultats en modifiant les priorités si on utilise 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.

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.

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 moyens.