CPE LYON

ÉCOLE SUPÉRIEURE DE CHIMIE PHYSIQUE ÉLECTRONIQUE DE LYON

Ex 1	Ex 2	Ex 3	Ex 4	Ex 5	Ex 6	Ex 7	Ex 8	Ex 9	Ex 10	Ex 11
1,5	1,5	1,5	1,5	1	0,5	2	2	1,5	1,5	4
1,5 point(s)	1,5 point(s)	1,5 point(s)	1,5 point(s)	2 point(s)	2 point(s)	2 point(s)	2 point(s)	1,5 point(s)	1,5 point(s)	4 point(s)

Note
18,5 **/20**

Consignes relatives au déroulement de l'épreuve

Date : 10 Janvier 2019
Contrôle : PROGRAMMATION CONCURRENTE – SESSION 1 - 4IRC 2018/2019
Durée : **2 heures**

Professeur responsable : T. LIMANE

Documents Cours/TP : ☒ autorisées ☐ non autorisés Calculatrices : ☐ autorisées ☒ non autorisées

8

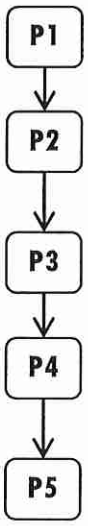
LES TELEPHONES PORTABLES ET AUTRES APPAREILS DE STOCKAGE DE DONNEES NUMERIQUES NE SONT PAS AUTORISES.
Les oreilles des étudiants doivent être dégagées.

Rappels importants sur la discipline des examens

- La présence à tous les examens est strictement obligatoire; tout élève présent à une épreuve doit rendre une copie, même blanche, portant son nom, son prénom et la nature de l'épreuve.
- Toute absence non justifiée est sanctionnée par un zéro.
- Toute fraude ou tentative de fraude avérée est sanctionnée par un zéro à l'épreuve et portée à la connaissance de la direction des études qui pourra réunir le Conseil de Discipline. Les sanctions prises peuvent aller jusqu'à l'exclusion définitive du (des) élève(s) mis en cause.
- **TOUTE SUSPICION SUR LA REGULARITE ET LE CARACTERE EQUITABLE D'UNE EPREUVE EST SIGNALEE A LA DIRECTION DES ETUDES QUI POURRA DECIDER L'ANNULATION DE L'EPREUVE; TOUS LES ELEVES CONCERNES PAR L'EPREUVE SONT ALORS CONVOQUES A UNE EPREUVE DE REMPLACEMENT A UNE DATE FIXEE PAR LE RESPONSABLE D'ANNEE.**

Exercice 1 [/1,5 point]

Soit l'arborescence de 5 processus présentée par la figure ci-dessous. Proposez un programme qui va générer cette arborescence de processus. Chacun des processus affichera son **pid** et le **pid** de son père.




```
int main () {
    printf("P1 : %d", getpid());
    if(!fork()) {
        printf("P2 : %d et père : %d", getpid(), getppid());
        if(!fork()){
            printf("P3 : %d et père : %d", getpid(), getppid());
            if(!fork()) {
                printf("P4 : %d et père : %d", getpid(), getppid());
                if(!fork()){
                    printf("P5 : %d et père : %d", getpid(), getppid());
                }
            }
        }
    }
}
```

Exercice 2 [/1.5 point]

Soit le programme **prog.c** (**prog** est le nom de l'exécutable) suivant :

```
int main() {
    int pid = fork() ;
    int X = 10 ;
    if (pid == 0) X += 5 ;
    else {
        pid = fork() ;
        X += 10 ;
        if (pid > 0)
            X += 2 ;
    }
    return 0;
}
```

1) Combien y-a-t-il de copies de **prog**? 3

2) Quelles sont leurs valeurs de la variable **X** à la fin du programme ?
   1er prog : X = 22
   2ème prog : X = 15
   3ème prog : X = 20

3) Que se passe-t-il si on remplace **if (pid>0) X+=5 ;** par : **if (pid>0) execlp(prog , prog , NULL) ;**

Le 1er prog sera écrasé par prog, c'est à dire que le programme va se lancer en boucle sans s'arrêter (sauf intervention extérieure)

Exercice 3 [/1,5 point]

Commenter ce programme et préciser la fonctionnalité réalisée par ce programme.

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

  int status, i , pid;

  for (i=1 ; i<argc; i++) {

    if (fork() > 0) {
                                  PERE
      pid = wait(&status) ;
        Attente de fin du fils
      if( WEXITSTATUS(status) != -1 ) printf( "%d - %s\n", pid, argv[i] ) ;
        Si le statut de sortie n'est pas une erreur, on print les infos du fils
    }

    else {
                                  FILS
      execlp(argv[i] , argv[i] , NULL) ;
          le programme fils est remplacé par une commande passée en argument
      exit(-1) ;
            -1 en cas d'erreur
    }

  }

  return 0;

}
```

Fonctionnalité :

Ce programme exécute les commandes passées en paramètre et affiche celles qui se sont exécutées avec succès.

Exercice 4 [/1.5 point]

On se propose d'implanter la ligne de commande shell **ls -l | wc –l** (qui compte le nombre d'entrées dans le répertoire courant) par le programme **lswc.c** suivant :

```
#include <unistd.h>
#include <stdlib.h>
int main() {
        int tube[2] ;
        pipe(tube) ;
        if(fork() != 0) {
                dup2(tube[0] , 0) ;
                close(tube[0]) ;
                execlp ("wc", "wc", "-l",  NULL) ;
        }
        if(fork() != 0) {
                dup2(tube[1] , 1) ;
                close(tube[1]) ;
                execlp ("ls", "ls, "-l", NULL) ;
        }
        while (wait(NULL) != -1) ;
        return 0 ;
}
```

Corrections

```
if ( fork() == 0 ) {
    close(tube[1]);
} else {
    close(tube[0]);
}
return 0 ;
}
```

Que se passe-t-il à l'exécution de ce programme ?

~~Un fils est créé à deux reprises, mais ses fils n'ont pas de traitement~~
Le père créé un fils, copie la sortie du tube vers l'entrée standard et se fait recouvrir par wc -l.

Expliquez la raison du comportement inattendu de ce programme.

Il faut que le premier execlp soit fait dans un fils.
Il est également plus sûr de fermer les entrées/sorties inutilisées du tube.

Corrigez ce programme par conséquent.

Exercice 5 [/2 points]

Trouvez toutes les erreurs logiques dans le segment de code suivant, en indiquant le numéro de la ligne erronée, ce qui est erroné puis proposez une correction.

```
1.  int S = sem_create(123 , 0) ;
2.  //ajouterItemQueue() retourne 0 si l'élément item a été ajouté à la file queue ; -1 sinon.
3.  int ajouterItemQueue(queue_t queue , item_y item) {
4.      P(S) ;
5.      if ( isFull(queue) )    //teste si la file est pleine
6.                  return -1;
7.      else {
8.          appendQueue(queue , item) ;    //ajouter item dans la file
9.      return 0 ;
10.     V(S) ;
11.     }
12. }
```

Ligne 4 et 1 : Prise de jeton dans un sémaphore vide. Le programme est bloqué.
Il faut initialiser le sémaphore à 1 : sem_create(123, 1)

Exercice 6 [/2 points]

Considérons les trois processus concurrents **p1**, **p2** et **p3** suivants.
Ils partagent trois sémaphores **S1**, **S2** et **S3** initialisés à **0**.

```
int main() { // p1
  F1() ;
  V(S2) ;
  V(S3) ;
  P(S1) ;
  P(S1) ;
  G1();
  return 0 ;
}
```

```
int main() { // p2
  F2() ;
  V(S1) ;
  V(S3) ;
  P(S2) ;
  P(S2) ;
  G2() ;
  return 0 ;
}
```

```
int main() { // p3
  F3() ;
  V(S1) ;
  V(S2) ;
  P(S3) ;
  P(S3) ;
  G3() ;
  return 0 ;
}
```

Quelle synchronisation a-t-on imposée sur les exécutions des fonctions **F1( )**, **F2( )**, **F3( )**, **G1( )**, **G2( )** et **G3( )** ?

Aucune synchronisation sur F1(), F2(), F3().
Mais un rendez vous à 3 est imposé sur G1(), G2(), G3().

Exercice 7 [/2 points]

Ecrire un programme C qui crée deux processus à l'aide de l'appel système **fork()**. Le père affichera les entiers pairs compris entre 1 et 100, le fils affichera les entiers impairs compris dans le même intervalle. Synchroniser les deux processus à l'aide des **signaux** pour que l'affichage soit **1 2 3 ... 100**.

```
int main() {
    int k, pid;
    if ((pid=fork())!= 0) {
        k=0;
        signal(SIGUSR1, fonc P);
        while (k<=100);
    } else {
        k=1;
        signal(SIGUSR2, fonc F);
        printf("%d", k);
        kill(getppid(), SIGUSR1);
        while (k<=100);
    }
    return 0;
}

void fonc P() {
    k = k+2;
    printf("%d", k);
    kill(pid, SIGUSR2);
}

void fonc F() {
    k=k+2;
    printf("%d", k);
    kill(getppid(), SIGUSR1);
}
```

Exercice 8 [/2 points]

```
#include <stdio.h>
#include <pthread.h>
int k;
void addition( void* arg){
  k = 10;
  printf("Hello Thread Enfant - %d\n" , k);
  k = k +20;
  printf("Hello Thread Enfant - %d\n" , k);
}
int main( ){
  pthread_t th;
  k = 0;
  pthread_create(&th, NULL, addition, NULL);
  k = k +100;
  printf("Hello Thread Principal - %d\n" , k);
  k = k + 200;
  printf("Hello Thread Principal - %d\n" , k);
  pthread_join(th , NULL);
  return 0;
}
```

1) Expliquez la différence de comportement entre la fonction **fork()** et la fonction **pthread_create()**.

fork() crée un nouveau processus alors que pthread_create() crée des thread au sein d'un processus.

2) Quelles remarques peut-on faire si l'on compare les traces d'exécution obtenues avec celles d'un programme réalisant la même fonctionnalité en utilisant l'appel **fork()** au lieu de l'appel **pthread_create()**.

La variable k étant globale, on ne peut déterminer sa valeur avant l'exécution avec l'utilisation de threads.
Avec fork(), k sera à 30 dans le fils et 300 dans le père.

Exercice 9 [1,1 / 1,5 point]

```
void maFonction(int sig) {
        printf("Ok");
}
int main(int argc, char **argv) {
        int t , i ;
        // conversion de la chaîne de caractères pointée par argv[1] en un entier et stockage du résultat à l'adresse &t
        sscanf(argv[1] , "%d" , &t) ;
        signal(SIGALRM , maFonction) ;
                Lance maFonction lorsque le signal d'alarme est détecté
        for( ;;) {
                alarm(t) ;
                        Lance SIGALARM après t secondes
                pause() ;
                for(i=2 ; i< argc ; i++)  kill(argv[i] , SIGUSR1) ;
                        Envoi du signal SIGUSR1 aux process passés en arguments
        }
}
```

Commenter ce programme et décrire la fonctionnalité réalisée.

Fonctionnalité :

Ce programme envoie SIGUSR1 aux process passés en argument (à partir du 2ème arg), et affiche "Ok" après un temps en seconde passé en 1er argument.

Exercice 10 [/1.5 point]

Soit le programme C suivant :

```
sem_t sem;
void* F(void* arg) {
    sem.post(sem) ;
    printf("Hello") ;
    sem.wait(sem) ;
    return NULL ;
}
int main( ) {
        pthread_t th[10] ;
        sem_init(&sem, 0, 0) ;
        for (int ind=0 ; ind<10 ; ind++) {
                if ( (rep = pthread_create(&th[ind], NULL, F, NULL)) == 0 ) {
                    printf("Pthread %d crée\n", ind) ;
                }
                else {
                    fprintf(stderr , "%d : ", ind) ;
                    perror("pthread_create") ;
                }
        }
        for (int ind=0; ind<10; ind++) {
            pthread_join(th[ind], NULL) ;
        }
        return 0 ;
}
```

Un inter-blocage entre les threads est-il possible ?

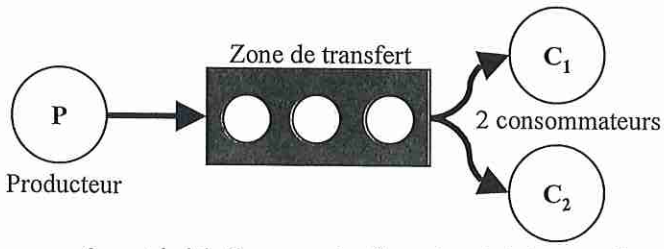
❍ **Oui** ⊗ **Non**

Justifiez votre réponse

Les threads font toujours un post avant un wait, ils pourront donc agir.

Exercice 11 [4 / 4 points]

On vous demande de synchroniser à l'aide de sémaphores **3 processus concurrents**. Vous disposez d'un type



sémaphore que vous pouvez créer et initialiser par la fonction **init(sem,valeur)** et deux fonctions **P(sem)** et **V(sem)**. Vous pouvez utiliser également des variables globales. Vous ne devez pas utiliser d'attente active et vous ne devez pas modifier la structure du problème. Dans un atelier d'assemblage automatisé, nous avons une machine **P** qui produit des pièces, et deux machines **C1** et **C2** qui les consomment. Le transfert entre le producteur et les consommateurs utilise un espace partagé qui peut contenir **3** pièces au maximum. Le producteur comporte une partie **ProduitUnePièce()** et une partie **DéposeLaPièce()**. Les consommateurs comportent une partie **RetireUnePièce()** et **ConsommeLaPièce()**. Il faut synchroniser l'exécution de ces différentes parties.

Il y a plusieurs contraintes de synchronisation à respecter. Le producteur peut commencer la partie de dépôt seulement lorsqu'il y a au moins un espace libre dans la zone de transfert. Les processus consommateurs doivent attendre qu'une pièce soit disponible dans la zone de transfert avant de commencer le retrait. Un seul consommateur peut retirer une pièce à la fois. Nous ne demandons pas d'alternance stricte entre les consommateurs. Un consommateur doit pouvoir retirer une pièce s'il est disponible, peu importe s'il a retiré la dernière pièce ou non. Un consommateur ne peut essayer de retirer une pièce que le producteur est en train de déposer. Le consommateur doit attendre la fin de dépôt de la pièce. La même contrainte s'applique pour les espaces libres de la zone de transfert.

Compléter la solution ci-dessous pour assurer la bonne synchronisation des processus. Il faut éviter que les processus attendent inutilement, ainsi que les inter-blocages.

init(mutex2, 1); init(semLibre, 3); init(mutexC, 1); init(semPièce, 0);

// _____Processus P_____	// _____Processus C1_____	// _____Processus C2_____
while(1) {	while(1) {	while(1) {
	P(mutex2); P(mutexC); P(semPièce);	P(mutex2); P(mutexC); P(semPièce);
ProduitUnePièce();	RetireUnePièce(); V(semLibre); V(mutexC); V(mutex2);	RetireUnePièce(); V(semLibre); V(mutexC); V(mutex2);
P(mutex2); P(semLibre);	ConsommeLaPièce();	ConsommeLaPièce();
DéposeLaPièce(); V(semPièce); V(mutex2);		
}	}	}