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	1,5	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 19 **/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

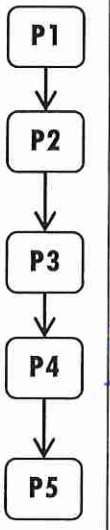
LES TELEPHONES PORTABLES ET AUTRES APPAREILS DE STOCKAGE DE DONNEES NUMERIQUES NE SONT PAS AUTORISES.
Les oreilles des étudiants doivent être dégagées.

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 /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 ( ) {
    for(int i=0 ; i <4 ; i++){
        if (fork !=0)
            break;
    }
    printf(" pid : %d - ppid : %d \n ", getpid(), getppid());
    return 0;
}
```

Exercice 2 [1,5 /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**?
   2 copies (fils donc duplication du processus) + père
2) Quelles sont leurs valeurs de la variable **X** à la fin du programme ?
   fils 1 : 15
   fils 2 : 20
   père : 22
3) Que se passe-t-il si on remplace **if (pid>0) X+=5 ;**
   par : **if (pid>0) execlp(prog , prog , NULL) ;**
   Le programme s'exécuterait à l'infini car le premier fils exécuterait ce même programme.

Exercice 3 [1,5 / 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++) { // on boucle autant qu'il y a d'arguments

    if (fork() > 0) { // On crée un processus fils et si on est dans le père on rentre dans la condition
      pid = wait(&status) ; // On attend que le fils se termine et on met son code de terminaison dans status et
                            // son pid dans pid.
      if( WEXITSTATUS(status) != -1 ) printf( "%d - %s\n", pid, argv[i] ) ;
      // Si le status de terminaison du fils est différent de -1, alors on affiche son pid
      // et la commande exécutée
    }

    else {
      // On est dans le fils
      execlp(argv[i] , argv[i] , NULL) ;
      // on execute la iième commande passée en argument
      exit(-1) ;
      // S'il y a eu un problème on retourne -1
    }

  }

  return 0;
  // Fin du programme
}
```

Fonctionnalité :

Ce programme exécute les commandes passées en paramètre (en créant des processus fils) et affiche le pid du fils et la commande si elle a été correctement exécutée.

Exercice 4 [1 /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) {
                close(tube[1]);
                dup2(tube[0] , 0) ;
                close(tube[0]) ;
                execlp ("wc", "wc", "-l",  NULL) ;
        }
        if(fork() != 0) {
                close(tube[0]);
                dup2(tube[1] , 1) ;
                close(tube[1]) ;
                execlp ("ls", "ls, "-l", NULL) ;
        }
        while (wait(NULL) != -1) ;
        return 0 ;
}
```

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

Ce programme n'affiche rien et reste bloqué → il ne se termine pas

Expliquez la raison du comportement inattendu de ce programme.

Le tube n'a pas été fermé en écriture dans le 1er fils et en lecture dans le second fils. De plus les 2 fils ne servent à rien, étant donné que tout est fait dans le père.

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. }
```

l.3 : il faut passer des pointeurs en paramètre pour que l'ajout soit fait sur ces objets
l.4 : Va rester bloqué car aucun jeton n'est créé → modifier l.1 et ajouter 1 jeton
l.10 : Il faut placer le V(S) avant le return de la ligne 9 sinon il ne sera jamais exécuté et aucun jeton ne sera libéré.
l.8 : Comme pr ligne 3 : mettre des pointeurs en adresses mémoire.

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( )** ?

Pour que G1(), G2() et G3() soient exécutées, il faut que F1(), F2() et F3() aient fini de s'exécuter dans les trois programmes.

Exercice 7 [2 /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 p;
void afficheChiffre (int sig) {
    printf("%d\n", p);
}
int main() {
   signal (SIGALRM, afficheChiffre);
   int pid;
   pid = fork();
   for(p=1 ; p<=100 ; p++){
       if (pid==0 && p%2==0){
           kill(getppid(), SIGALRM);
       }
       else if(pid>0 && p%2!=0){
           kill(pid, SIGALRM);
       }
       else {
  }    }  pause();
   return 0;
}
```

Exercice 8 [2 /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()**.

Un fork() duplique le processus. C'est à dire que dans ce programme, la variable k ne sera pas la même pour le père ou pour le fils. Si le fils modifie la variable k, ça n'aura pas d'impacte sur la variable k du père. Par contre, dans un thread, les variables globales sont communes et donc une modification de la variable k est visible pour les threads enfant et pour le thread principal.

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()**.

Avec fork : Hello Thread Enfant - 10
Hello Thread Enfant - 30
Hello Thread Principal - 100
Hello Thread Principal - 200
} peut changer selon ordre exécution

Avec les threads, comme la variable k est commune, la valeur peut changer selon l'exécution

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

```
void maFonction(int sig) {

        printf("Ok");
        //affiche 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) ;
        //Si signal SIGALRM est reçu, on exécute la fonction maFonction
        for( ;;) {
        //boucle infinie
                alarm(t) ;
                //envoie d'un signal SIGALAM au bout de t secondes
                pause() ;
                //attente
                for(i=2 ; i< argc ; i++)  kill(argv[i] , SIGUSR1) ;
                //envoie signal SIGUSR1 à tous les paramètre à partir du 2ème
        }

}
```

Commenter ce programme et décrire la fonctionnalité réalisée.

Fonctionnalité :

Ce programme envoie (en boucle) au bout d'un certains nombre de secondes passées en paramètre un signal SIGUSR1 à tous les processus passées aussi en paramètre. Lorsque l'envoie de signal SIGUSR1 commence, le programme affiche "Ok"

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 ?

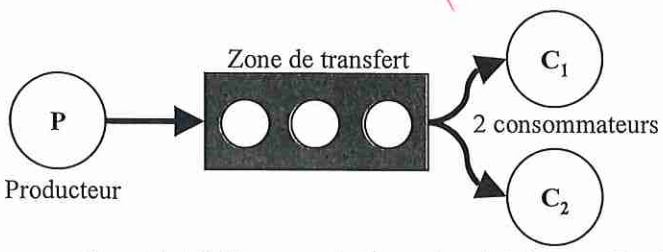
○ **Oui** ⊗ **Non**

Justifiez votre réponse

Il n'y a pas d'inter-blocage possible car chaque thread exécute la fonction F qui produit d'abord un jeton, donc le jeton consommé en fin de fonction aura toujours été produit auparavant.

Exercice 11 [/ 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 **C_1** et **C_2** 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.

// _____Processus P_____	// _____Processus C1_____	// _____Processus C2_____

while(1) {
init(semLibre, 3);
init(semAction, 1);
init(semDispo, 0);
 	
ProduitUnePièce();
P(semLibre);
P(semAction);
 	
DéposeLaPièce();
V(semAction);
V(semDispo);
}
	while(1) {
	init(semLibre, 3);
	init(semAction, 1);
	init(semDispo, 0);
	P(semDispo);
	P(Action);
	RetireUnePièce();
	V(semDispo);
	V(Action);
	V(semLibre);
	ConsommeLaPièce();
	
	}
		while(1) {
		init(semLibre, 3);
		init(semAction, 1);
		init(semDispo, 0);
		P(semDispo);
		P(Action);
		RetireUnePièce();
		V(semDispo);
		V(Action);
		V(semLibre);
		ConsommeLaPièce();
		
		}