ED9 : Synchronisation par Sémaphores

Exercice 1.

Nous nous intéressons à la résolution du problème d'exclusion mutuelle en utilisant les sémaphores de Dijkstra.

Question 1 :
Rappelez le concept des sémaphores en mettant en évidence le fonctionnement des primitives P et V.

Question 2 :
Montrez comment réaliser l'exclusion mutuelle avec deux tâches en utilisant les sémaphores. Vérifiez que votre solution garantit les propriétés de l'exclusion mutuelle.

Exercice 2.

Pour faciliter l'utilisation des sémaphores Posix et des threads Unix, nous proposons une API fournissant des types et des fonctions relatives à la manipulation de sémaphores et de tâches. Pour cela, nous proposons deux fichiers :

- Un fichier *Sem_Task.h* qui contient des déclarations de types facilitant la déclaration de sémaphores et de tâches (threads Unix).
  Ainsi la déclaration d'un sémaphore *s* se résume en la ligne suivante :
  SEM s ;

- Et la déclaration d'une tâche (symbolisant une thread Unix) se fait comme suit :
  TASK t ;

Pour cela, il suffira d'inclure dans le programme principal le fichier *Sem_Task.h* suivant :

Sem_Task.h ----------

```
typedef sem_t * SEM;
typedef  pthread_t * TASK;
typedef void (*TASK_CODE)(void);

// fonctions relatives à la manipulation des sémaphores
SEM newSEM();
int E0(SEM S, int val);
int P(SEM S);
int V(SEM S);
int deleteSEM(SEM S);

//fonctions relatives à la manipulation des tâches
TASK newTASK();
void launchTASK(TASK T, TASK_CODE P);
void waitTASK(TASK T);
```

Comme nous pouvons le constater, le fichier *Sem_Task.h* définit les prototypes de fonctions relatives à la manipulation de sémaphores et de tâches :
newSEM() pour créer un sémaphore,
E0() pour initialiser un sémaphore,

P() pour faire un P sur un sémaphore,
V() pour faire un V sur un sémaphore,
deleteSEM() pour détruire un sémaphore (ressources allouées).

newTASK pour créer une tâche,
launchTASK() pour lancer une tâche en précisant son nom et son code,
waitTASK() pour attendre la fin d'une tâche.

Ces fonctions sont mises à la disposition du programmeur afin de manipuler des sémaphores et des tâches, l'objectif étant de faire abstraction des sémaphores Posix et des threads qui sont souvent difficiles à utiliser (notamment à cause des pointeurs).

Le corps des différentes fonctions est donné dans le fichier *Sem_Task.c*. Le programmeur doit inclure dans son répertoire local ce fichier qui doit être compilé séparément comme indiqué dans le fichier Makefile décrit ici.

Sem_Task.c ----------

```
#include<semaphore.h>
#include<stdlib.h>
#include <pthread.h>
#include "Sem_Task.h"

int E0(SEM S, int val) // initialiser la valeur d'un sémaphore
{ return (sem_init(S, 0, val));
}

int P(SEM S)  { // faire un P sur un sémaphore
      return(sem_wait(S));
      }

int V(SEM S) { //faire un V sur un sémaphore
return(sem_post(S));
}

int deleteSEM(SEM S){ // libérer les resources du sémaphore
return(sem_destroy(S));
}

SEM newSEM () {
      return((SEM) malloc (sizeof(sem_t)));
}

void launchTASK (TASK T, TASK_CODE Proc){
pthread_create (T, 0, (void * (*)()) Proc, NULL);
}

void waitTASK(TASK T) {
      pthread_join (*T, NULL);
      }

TASK newTASK() {
return((TASK)malloc(sizeof(pthread_t)));
}
```

Question 1 :

Traduire la solution de l'exercice 1 (sur l'exclusion mutuelle) en utilisant l'API proposée dans cet exercice.

Question 2 :

On s'intéresse au modèle producteur/consommateur.

Écrire l'algorithme du producteur/consommateur vu en cours, en utilisant des sémaphores. On montrera que cet algorithme respecte bien les propriétés du modèle producteur /consommateur.

Question 3 :

En vous inspirant de l'algorithme du producteur/consommateur, compléter le programme suivant à l'aide des fonctions décrites dans l'API.

```
/* prodcons.c avec des threads*/
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include "Sem_Task.h"

#define Ncases 10   /* nbr de cases du tampon */


int   Tampon[Ncases];         /* Tampon a N cases*/
SEM Snvide, Snplein;    /* les semaphores */

TASK_CODE Producteur() {
      int i, queue=0, MessProd;

      srand(getpid());

      for(i=0; i<20; i++){
            sleep(rand()%3);   /* fabrique le message */
            MessProd = rand() % 10000;
            printf("Product %d\n",MessProd);

            ????? //bloquer le producteur si Tampon plein
            Tampon[queue]=MessProd;
            ????? // signaler la production
            queue=(queue+1)%Ncases;
      }

}

TASK_CODE Consommateur() {
      int tete=0, MessCons, i;

      srand(getpid());
      for(i=0; i<20; i++){
            ??????      // bloquer consommateur si Tampon vide
            MessCons = Tampon[tete];
            ?????  // signaler la consommation
            tete=(tete+1)%Ncases;
            printf("\t\tConsomm  %d \n",MessCons);
            sleep(rand()%3);   /* traite le message */
      }
```

```
}

int   main(void) {
      TASK t1, t2;

/* creation et initialisation des semaphores */
            ?????

/* creation et lancement des taches  */
            ?????

/* attente de la fin des taches */
            ?????

/* suppression des semaphores */
            ??????
return (0);
}
```

A titre d'informations, voici le contenu du fichier Makefile permettant l'exécution de ce programme :

Exécution :

Le fichier Makefile : ----------

```
all: pgm

pgm: prodcons.o Sem_Task.o
      gcc prodcons.o Sem_Task.o -lpthread -o pgm

prodcons.o: prodcons.c Sem_Task.h
      gcc -c prodcons.c

Sem_Task.o: Sem_Task.c Sem_Task.h
      gcc -c Sem_Task.c

clean:
      \rm -f *.o pgm
```

```
$ls
Makefile   prodcons.c   Sem_Task.h   Sem_Task.c
$make
$./pgm
Prod  345
Prod  567
      Consomm 345
Prod  789
      Consomm 567
Prod  298
Prod  675
Prod  890
      Consomm 789
      Consomm 298
...
$make clean  //pour détruire les fichiers objets créés précédemment
```