Haute école d'ingénierie et de gestion du Canton de Vaud
Département TIC
Programmation concurrente 1 (PCO 1)

Contrôle continu du mardi 14 juin 2011 de 11h15 à 12h00

Remarques :
- Ce contrôle comprend **4** questions.
- Aucune documentation permise.
- Vous pouvez répondre sur l'énoncé, sinon vous pouvez le conserver.
- N'utilisez pas de couleur rouge.

Question 1 (1,8 points)

Dans cette question, on souhaite réaliser des fonctions réalisant une section critique prioritaire. Il y a 2 niveaux de priorité : *haute* et *basse*.

- La fonction `DemandeHautePriorite` est appelée par les tâches qui sont prioritaires lorsque celles-ci doivent accéder à la section critique; si la section critique est déjà allouée au moment de cet appel, la tâche appelante doit se bloquer.
- La fonction `DemandeBassePriorite` est semblable, mais elle est appelée par les tâches qui sont moins prioritaires.
- Enfin la fonction `Libere` relâche la section critique et la passe à une autre tâche en favorisant celles qui sont bloquées par `DemandeHautePriorite` avant celles qui se trouvent bloquées par `DemandeBassePriorite`.

Proposez une solution en C en utilisant uniquement le concept de moniteur fourni par POSIX. Complétez le squelette fourni ci-dessous.

```
void Initialisation(void) {

}

void DemandeHautePriorite(void) {

}

void DemandeBassePriorite(void) {

}

void Libere(void) {

}
```

Rappel sur les moniteurs Posix:

- `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER` : déclare et initialise le verrou `m`;
- `pthread_mutex_init(&m)` : permet d'initialiser un verrou et le mettre à l'état déverrouillé.
- `pthread_mutex_lock(&m)` : permet d'obtenir l'exclusion mutuelle du moniteur protégé par le verrou `m`;
- `pthread_mutex_unlock(&m)` : permet de relâcher l'exclusion mutuelle du moniteur protégé par le verrou `m`;
- `pthread_cond_t c = PTHREAD_COND_INITIALIZER` : déclare et initialise une variable condition `c`;
- `pthread_cond_init(&c)` : permet d'initialiser dynamiquement la variable condition `c`;
- `pthread_cond_wait(&c,&m)` : correspond à attendre sur la variable condition `c`, en relâchant le verrou `m`;
- `pthread_cond_signal(&c)` : correspond à signaler la variable condition `c`.

Question 2 (1 point)

Qu'est-ce que le paradigme des lecteurs et des rédacteurs? Votre réponse devra indiquer le rôle des tâches et les contraintes du problème. (N'écrivez aucun code pour cette question.)

Question 3 (1,2 points)

Les 3 fonctions données ci-dessous implémentent le problème d'un producteur et de 2 consommateurs où chaque consommateur doit consommer les items déposés par le producteur. Ces 2 consommateurs sont identifiés par un indice pouvant prendre les valeurs 0 ou 1.

```
#define TAILLE 8                                          /* 1 */
int en, hors[2];                                          /* 2 */
sem_t m[2], p[2];                                         /* 3 */
ITEM tab[TAILLE];                                         /* 4 */

void Initialise(void) {                                   /* 5 */
  int i;                                                  /* 6 */
  for (i = 0; i < 2; i += 1) {                            /* 7 */
     sem_init(&m[i],0,0);                                 /* 8 */
     sem_init(&p[i],0,TAILLE);                            /* 9 */
     hors[i] = 0;                                         /* 10 */
  }                                                       /* 11 */
  en = 0;                                                 /* 12 */
}                                                         /* 13 */

void Deposer(ITEM item) {                                 /* 14 */
  sem_wait(&p[0]);                                        /* 15 */
  sem_wait(&p[1]);                                        /* 16 */
  tab[en] = item;                                         /* 17 */
  en = (en + 1) % TAILLE;                                 /* 18 */
  sem_post(&m[0]);                                        /* 19 */
  sem_post(&m[1]);                                        /* 20 */
}                                                         /* 21 */

ITEM Prelever(int groupe) {                               /* 22 */
  ITEM item;                                              /* 23 */
  sem_wait(&m[groupe]);                                   /* 24 */
  item = tab[hors[groupe]];                               /* 25 */
  hors[groupe] = (hors[groupe] + 1) % TAILLE;             /* 26 */
  sem_post(&p[groupe]);                                   /* 27 */
  return item;                                            /* 28 */
}                                                         /* 29 */
```

a) Que faut-il modifier si nous admettons qu'il peut y avoir plusieurs producteurs? (0,6 point)

b) Que faut-il modifier si les 2 consommateurs sont remplacés par un ensemble de consommateurs qui sont répartis entre 2 groupes? Autrement dit, plusieurs consommateurs peuvent appartenir à un groupe. (0,6 point)

Rappel sur les sémaphores Posix :

- `sem_init(&s,0,n)` : correspond à initialiser le sémaphore `s` de type `sem_t` à `n` (`n` doit être ≥ 0) ;
- `sem_wait(&s)` : correspond à P(`s`) ;
- `sem_post(&s)` : correspond à V(`s`).

Question 4 (1 point)

En classe nous avons montré comment transformer un moniteur écrit dans un langage haut niveau tel que Pascal Concurrent en une suite de règles pour que le code résultant n'utilise que des sémaphores pour la synchronisation.

Rappel des règles

1. Définir une structure pour représenter un moniteur :
```
typedef struct {                    // 1.1
   sem_t mutex;                     // 1.2
   sem_t signale;                   // 1.3
   unsigned nbSignale;              // 1.4
} T_Moniteur;                       // 1.5
```
2. Puis déclarer une instance :
```
T_Moniteur mon;                     // 2.1
```
3. Chaque procédure ou point d'entrée du moniteur est encadré par :
```
sem_wait(&mon.mutex);               // 3.1
< code de la procédure >            // 3.2
if (mon.nbSignale > 0)              // 3.3
   sem_post(&mon.signale);          // 3.4
else                                // 3.5
   sem_post(&mon.mutex);            // 3.6
```
4. Définir une structure pour modéliser une file d'attente pour les variables condition
```
typedef struct {                    // 4.1
   sem_t attente;                   // 4.2
   unsigned nbAttente;              // 4.3
} T_Condition;                      // 4.4
```
5. Pour chaque variable condition *cond* du moniteur, déclarer
```
T_Condition cond;                   // 5.1
```
6. Dans toutes les procédures du moniteur, substituer *cond.attente* par :
```
cond.nbAttente += 1;                // 6.1
if (mon.nbSignale > 0)              // 6.2
   sem_post(&mon.signale);          // 6.3
else                                // 6.4
   sem_post(&mon.mutex);            // 6.5
sem_wait(&cond.attente);            // 6.6
cond.nbAttente -= 1;                // 6.7
```
7. Dans toutes les procédures du moniteur, substituer *cond.signale* par :
```
if (cond.nbAttente > 0) {           // 7.1
   mon.nbSignale += 1;              // 7.2
   sem_post(&cond.attente);         // 7.3
   sem_wait(mon.signale);           // 7.4
   mon.nbSignale -= 1;              // 7.5
}                                   // 7.6
```

La transformation donnée ci-dessus s'applique au cas

priorité(*entrée*) < *priorité*(*signale*) < *priorité*(*attente*)

où *entrée*, *signale* et *attente* désignent respectivement les tâches qui accèdent au moniteur depuis l'extérieur, effectue l'appel à *cond.signale* et se font réveiller après un appel à *cond.attente*.

Que faut-il changer à ces règles si nous voulons que le moniteur suive l'ordre de priorité

priorité(*entrée*) < *priorité*(*attente*) < *priorité*(*signale*)

Autrement dit, une tâche qui réveille une autre ne lui cède pas le moniteur mais continue son exécution.

14.06.2011

PCO : Test no 2

4.0

1	1.8
2	1
3	0
4	0.1
	2.9

Question 1

1.8 / 1.8

```
void Initialisation(void) {
    pthread_mutex_init(&mutex);
    pthread_cond_init(&SCestLibreHaute);
    pthread_coud_init(&SCestLibreBasse);
    attenteHaute = attenteBasse = 0;
    etatSC = false;
}

void DemandeHautePriorite(void) {
    pthread_mutex_lock(&mutex);
    attenteHaute++;
    while (etatSC)
        pthread_coud_wait(&SCestLibreHaute, &mutex);
    etatSC = true;
    attenteHaute--;

    pthread_mutex_unlock(&mutex);
}

void DemandeBassePriorite(void) {
    pthread_mutex_lock(&mutex);
    if (etatSC || attenteHaute + attenteBasse > 0)
        attenteBasse++;
        pthread_coud_wait(&SCestLibreBasse, &mutex);
        attenteBasse--;
    etatSC = true;
    pthread_mutex_unlock(&mutex);
}
```

```
void Libere(void) {
    etatSC = false;
    if (attenteHaute > 0)
        pthread_coud_signal(&SCestLibreHaute);
    else
        pthread_coud_signal(&SCestLibreBasse);
}
```

Question 2

Ce paradigme définit si des données peuvent être lues, ou modifiées.

Les lecteurs peuvent lire les données à plusieurs en même temps.

Les rédacteurs peuvent aussi écrire les données, mais un seul à la fois.

Il ne peut pas y avoir de lecteur lisant les données si un rédacteur est en cours d'écriture. Sinon les données risqueraient d'être incohérentes pour le lecteur.

Il ne peut pas y avoir plusieurs rédacteurs en cours d'écriture. Sinon les données risqueraient de nouveau d'être incohérentes.

Question 4

Echanger 6.6-6.7 avec 6.3 | interblocage.

0/1 " 7.3 avec 7.4

1 Une tâche qui signale ne doit pas se mettre en attente, sinon Prio(signale) > que tous les autres n'est pas respectée.

hors proj

Question 3 0/1.2

a) ajout d'une variable comptant le nb de cases libres afin que le producteur suivant n'attende pas que les items du producteur précédent soient consommées mais qu'il puisse les mettre dans la case suivante tout de suite. c'est déjà fait par le biais des sémaphores p[0] et p[1].

0.6/0.6 mais il faut une exclusion mutuelle.

b) rien

Imaginons qu'un producteur dépose 2 items dans des cases, et qu'il y a 2 consommateurs appartenant au même groupe qui appelent Prelever.
Ces 2 consommateurs peuvent franchir la barrière réalisée par la ligne 24.
Supposons que le premier consommateur réalise les lignes 24 et 25 puis se fait préempter.
Le second consommateur fait ensuite les ligne 24, 25 (lit la même chose que le premier consommateur) et continue avec les ligne 26 à 29. Quand le premier consommateur reprend son exécution, il positionnera hors[groupe].
Mais alors le 2e item déposé par le producteur n'est jamais lu par ce groupe alors que le groupe a lu 2x le 1er item.
=> il faut une exclusion mutuelle.

0/0.6

PCO1 : Corrigé du CC du 14 juin 2011

Question 1

1ère solution

```
int nbHaute, nbBasse, occupe;
cond_t haute, basse;
mutex_t mutex;
void Initialisation(void) {
  nbHaute = nbBasse = occupe = 0;
  pthread_cond_init(&haute);
  pthread_cond_init(&basse);
  pthread_mutex_init(&mutex);
}
void DemandeHautePriorite(void) {
  pthread_mutex_lock(&mutex);
  if (occupe || nbHaute + nbBasse > 0) {
     nbHaute += 1;
     pthread_cond_wait(&haute,&mutex);
     nbHaute -= 1;
  }
  occupe = 1;
  pthread_mutex_unlock(&mutex);
}
void DemandeBassePriorite(void) {
  pthread_mutex_lock(&mutex);
  if (occupe || nbHaute + nbBasse > 0) {
     nbBasse += 1;
     pthread_cond_wait(&basse,&mutex);
     nbBasse -= 1;
  }
  occupe = 1;
  pthread_mutex_unlock(&mutex);
}
void Libere(void) {
  pthread_mutex_lock(&mutex);
  occupe = 0;
  if (nbHaute > 0)
     pthread_cond_signal(&haute);
  else
     pthread_cond_signal(&basse);
  pthread_mutex_unlock(&mutex);
}
```

Autre solution

```
int nbHaute, nbBasse, occupe;
cond_t haute, basse;
mutex_t mutex;
void Initialisation(void) {
  nbHaute = nbBasse = occupe = 0;
  pthread_cond_init(&haute);
  pthread_cond_init(&basse);
  pthread_mutex_init(&mutex);
}
void DemandeHautePriorite(void) {
  pthread_mutex_lock(&mutex);
  if (occupe) {
     nbHaute += 1;
     pthread_cond_wait(&haute,&mutex);
     nbHaute -= 1;
  }
  occupe = 1;
  pthread_mutex_unlock(&mutex);
}
void DemandeBassePriorite(void) {
  pthread_mutex_lock(&mutex);
  if (occupe) {
     nbBasse += 1;
     pthread_cond_wait(&basse,&mutex);
     nbBasse -= 1;
  }
  occupe = 1;
  pthread_mutex_unlock(&mutex);
}
void Libere(void) {
  pthread_mutex_lock(&mutex);
  if (nbHaute > 0)
     pthread_cond_signal(&haute);
  else if (nbBasse > 0)
     pthread_cond_signal(&basse);
  else occupe = 0;
  pthread_mutex_unlock(&mutex);
}
```

Question 3

a) S'il y a plusieurs producteurs, plusieurs producteurs peuvent simultanément faire les lignes 17 et 18 en même temps => désastre.
Solution : introduire une exclusion mutuelle entre les producteurs
- entre les lignes 4 et 5, ajouter
  `sem_t mutexProd;`
- entre les lignes 11 et 12, ajouter
  `sem_init(&mutexProd,0,1);`
- entre les lignes 16 et 17, ajouter
  `sem_wait(&mutexProd);`
- entre les lignes 18 et 19, ajouter
  `sem_post(&mutexProd);`

b) S'il y a plusieurs consommateurs appartenant au même groupe, ceux-ci peuvent simultanément faire la ligne 26 => désastre.
Par exemple : Un producteur dépose 2 items et incrémente 2 x le sémaphore `p[i]`, ce qui permet de faire 2 appels à `Prelever(i)` simultanément. Si le premier consommateur réalise les lignes 22 à 25 et se fait préempter, le second peut alors prendre le même item, incrémenter `en[i]`. Quand le premier consommateur reprend son exécution, il fait aussi l'incrément de `en[i]` => 2e item déposé par le producteur n'est jamais consommé.
Solution : introduire une exclusion mutuelle entre les consommateurs d'un groupe :

- entre les lignes 4 et 5, mettre
```
sem_t mutexCons[2];
```
- entre les lignes 9 et 10, mettre
```
sem_init(&mutexCons[i],0,1);
```
- entre les lignes 24 et 25, mettre
```
sem_wait(&mutexCons[groupe]);
```
- entre les lignes 26 et 27, mettre
```
sem_post(&mutexCons[groupe]);
```

Question 4

Quand un thread A signale un autre, B, qui est en attente sur une condition (autrement dit quand A fait `pthread_cond_signal(&c)` et réveille B qui a fait `pthread_cond_wait(&c,&m)`, A doit poursuivre son exécution en maintenant l'usage exclusif du moniteur (pas de relâchement du verrou `m`), le thread B doit reprendre le moniteur que lorsque A le libère. Il nous faut donc une *file globale* emmagasinant tous les threads réveillés.
Pour faire le moins de changement possible au code, nous pouvons changer le sens de la paire (*signale,nbSignale*) : au lieu de compter le nombre de threads qui ont signalé et qui sont en attente (ce nombre est toujours 0), on peut y stocker les threads qui se sont fait réveiller.

Changement à faire
Remplacer la ligne 6.7 par
```
sem_wait(&mon.signale); // Attend que le signaleur libère le moniteur
mon.nbSignale -= 1;     // Un de moins à réveiller
```

Entre les lignes 7.1 et 7.2, mettre
```
cond.attente -= 1;      // Réveiller qu'une seule fois le même thread
```
Supprimer les lignes 7.4 et 7.5

Avec ce changement:
1. Un thread qui réveille un autre continue son exécution.
2. Quand un thread quitte le moniteur, le moniteur est passé à un thread qui s'est fait réveillé; s'il n'y en a pas, le moniteur est libéré afin de permettre à ceux venant de l'extérieur.
3. Quand un thread se met en attente, il passe d'abord le moniteur à un thread qui s'est fait réveillé comme au point 2, et évidement s'il y en a pas, il relâche le moniteur.