Haute école d'ingénierie et de gestion du Canton de Vaud
Département d'électricité et informatique

Programmation concurrente 1 (PCO1-B)
Contrôle continu

Mardi 5 juin 2012 de 13h50 à 14h35

Remarques :

- Le contrôle comprend 3 questions.
- Aucune documentation permise.
- Vos réponses doivent être **brèves**, **concises** et **claires**.
- Répondez uniquement sur les feuilles distribuées.
- Vous pouvez conserver l'énoncé.

Question 1 (2 points)

Nous avons traité en classe le problème où un ensemble de threads se répartissait en 2 classes selon les contraintes suivantes :

- Les threads appartenant à la même classe peuvent accéder concurremment à la ressource partagée.
- Les threads appartenant à différentes classes ne peuvent pas accéder à la ressource simultanément. Autrement dit, les threads de classes différentes s'excluent mutuellement.

Le code ci-dessous présente une implémentation en utilisant les moniteurs POSIX.

```
pthread_mutex_t protege = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t attente[2] = {PTHREAD_COND_INITIALIZER,PTHREAD_COND_INITIALIZER};
unsigned dedans[2] = {0,0};

void EntreeAcces(unsigned classe)                          // E1
{ // classe = 0 ou 1                                       // E2
  pthread_mutex_lock(&protege);                            // E3
  while (dedans[1-classe] > 0)                             // E4
     pthread_cond_wait(&attente[classe],&protege);         // E5
  dedans[classe] += 1;                                     // E6
  pthread_cond_signal(&attente[classe]);                   // E7
  pthread_mutex_unlock(&protege);                          // E8
  // Accès à la ressource                                  // E9
} /* fin de EntreeAcces */                                 // E10

void SortieAcces(unsigned classe)                          // S1
{ // classe = 0 ou 1                                       // S2
  pthread_mutex_lock(&protege);                            // S3
  dedans[classe] -= 1;                                     // S4
  if (dedans[classe] == 0)                                 // S5
     pthread_cond_signal(&attente[1-classe]);              // S6
  pthread_mutex_unlock(&protege);                          // S7
} /* fin de SortieAcces */                                 // S8
```

Cette solution présente une famine potentielle, car si une classe obtient l'usage de la ressource et réussit à maintenir un flux constant d'accès, les threads de l'autre classe en seront pénalisés. Pour contourner cette situation, le développeur décide de mettre en place une priorité dynamique : dès que l'une des classes a plus de 8 threads en attente, les threads de l'autre classe n'ont plus le droit d'accéder à la ressource.

Donnez les modifications à faire au moniteur pour implémenter cette solution. (Vos modifications ne devront pas faire appel à de l'attente active, et les points sont attribués uniquement au code modifié et pas aux explications.)

Rappel sur les moniteurs Posix :
- `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER` : déclare et initialise le verrou `m`;
- `pthread_mutex_init(&m,NULL)` : 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,NULL)` : 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)

Dans cette question, nous avons **un** producteur fournissant des données à **des** consommateurs par l'intermédiaire de 2 tampons de 64 octets chacun. Si les données ont une taille ≤ à 64 octets, nous pouvons directement appliquer la solution classique des producteurs et des consommateurs que nous avons vue en classe. Par contre, si ces données ont une taille supérieure à 64 octets mais inférieure à 256, un protocole possible entre le producteur et ses consommateurs serait que le premier octet du message contienne la taille des données déposées et que le producteur remplisse le reste du tampon partagé ainsi que les suivants avec les données à consommer. Un consommateur peut alors allouer un tampon (*malloc*) ayant la taille de la donnée, puis remplir ce tampon avec les données se trouvant dans les tampons partagés.

Une solution incomplète est donnée ci-dessous. Comment faut-il la changer pour que cette solution réponde à la spécification demandée? (Les points sont attribués uniquement au code modifié et pas aux explications.)

```
typedef unsigned char uint8;
int En, Hors;
uint8 Tampon[2][64];
sem_t Vide, Plein;

void IniTampon(void) {                                                  // I1
  En = Hors = 0;                                                        // I2
  sem_init(&Vide,0,2);                                                  // I3
  sem_init(&Plein,0,0);                                                 // I4
} /* fin de IniTampon */                                               // I5

void Depose(uint8 *donnee, uint8 taille) {                              // D1
  uint8 t;                                                              // D2
  sem_wait(&Vide);                                                      // D3
  Tampon[En][0] = (uint8)taille;                                        // D4
  memcpy(&Tampon[En][1],donnee, taille > 63 ? 63 : taille);             // D5
  sem_post(&Plein);                                                     // D6
  En = (En + 1) % 2;                                                    // D7
  for (t = 63; t < taille; t += 64) {                                   // D8
     sem_wait(&Vide);                                                   // D9
     memcpy(Tampon[En],&donnee[t], taille - t > 64 ? 64 : taille - t);  // D10
     sem_post(&Plein);                                                  // D11
     En = (En + 1) % 2;                                                 // D12
  }                                                                     // D13
} /* fin de Depose */                                                  // D14

uint8 *Retire(uint8 *taille) {                                          // R1
  uint8 t, *donnee;                                                     // R2
  sem_wait(&Plein);                                                     // R3
  *taille = Tampon[Hors][0];                                            // R4
  donnee = (uint8 *)malloc(*taille*sizeof(uint8));                      // R5
  memcpy(&Tampon[Hors][1],donnee, *taille > 63 ? 63 : *taille);         // R6
  sem_post(&Vide);                                                      // R7
  Hors = (Hors + 1) % 2;                                                // R8
  for (t = 63; t < *taille; t +=64) {                                   // R9
     sem_wait(&Plein);                                                  // R10
     memcpy(Tampon[Hors],&donnee[t],*taille - t > 64 ? 64 : *taille - t); // R11
     sem_post(&Vide);                                                   // R12
     Hors = (Hors + 1) % 2;                                             // R13
  }                                                                     // R14
  return donnee;                                                        // R15
} /* fin de Retire */                                                  // R16
```

Remarque : Pour une communication USB, certains constructeurs de microcontrôleurs mettent à disposition 2 tampons de 64 octets et ayant un contrôle de flux pour éviter l'écrasement de tampons encore à consommer. Ici, le pilote d'entrée de l'USB s'apparente à un producteur.

Rappel sur les sémaphores Posix :
- `sem_t a` : déclare un sémaphore
- `sem_init(&a,0,n)` : correspond à initialiser le sémaphore `a` à `n` (`n` doit être ≥ 0);
- `sem_wait(&a)` : correspond à `P(a)`;
- `sem_post(&a)` : correspond à `V(a)`.

Question 3 (2 points)

Les fonctions données sur la page suivante permettent de résoudre le problème d'**un** producteur fournissant des items à *N* consommateurs et où chaque consommateur doit consommer l'item produit. Les consommateurs sont numérotés de 0 à N–1 et appellent la fonction *Preleve* en passant leur numéro. Cette solution est cependant fausse.

(a) En vous servant d'un exemple d'exécution, montrez la faille décelée. (0,8 points)

(b) Montrez comment corriger ce code. (Les points sont attribués uniquement au code modifié et pas aux explications.) (1,2 points)

Indication : La fonction *memcpy*() copie *n* octets depuis la zone mémoire *src* vers la zone mémoire *dest*.

```
void *memcpy(void *dest, const void *src, size_t n);
```

```
#define N  2
#define Vide false
#define Plein true

static ITEM Element;                                          // 1
static sem_t Mutex, AttendreVide, AttendrePlein;              // 2
static int AttenteProd, AttenteCons;                          // 3
static bool Etat[N]; // = {Vide,...,Vide};                    // 4

bool InitialiseTampon(void) {                                 // I1
  int i;                                                      // I2
  if (sem_init(&Mutex,0,1) == 0)                              // I3
     if (sem_init(&AttendreVide,0,0) == 0)                    // I4
        if (sem_init(&AttendrePlein,0,0) == 0) {              // I5
           for (i = 0; i < N; i += 1)                         // I6
              Etat[i] = Vide;                                 // I7
           AttenteProd = AttenteCons = 0;                     // I8
           return true;                                       // I9
        }                                                     // I10
  return false;                                               // I11
} /* fin de InitialiseTampon */                              // I12

bool TamponVide(void) {                                       // T1
  int i;                                                      // T2
  for (i = 0; i < N; i += 1)                                  // T3
     if (Etat[i] == Plein) return false;                      // T4
  return true;                                                // T5
} /* fin de TamponVide */                                    // T6

void Depose(ITEM item) {                                      // D1
  int i;                                                      // D2
  sem_wait(&Mutex);                                           // D3
  if (!TamponVide()) {                                        // D4
     AttenteProd += 1;                                        // D5
     sem_post(&Mutex);                                        // D6
     sem_wait(&AttendreVide);                                 // D7
     sem_wait(&Mutex);                                        // D8
  }                                                           // D9
  Element = item;                                             // D10
  for (i = 0; i < N; i += 1)                                  // D11
     Etat[i] = Plein;                                         // D12
  if (AttenteCons > 0) {                                      // D13
     for (i = 0; i < AttenteCons; i += 1)                     // D14
        sem_post(&AttendrePlein);                             // D15
     AttenteCons = 0;                                         // D16
  }                                                           // D17
  sem_post(&Mutex);                                           // D18
} /* fin de Depose */                                        // D19

ITEM Preleve(int id) { // id = 0 .. N-1                       // P1
  ITEM item;                                                  // P2
  sem_wait(&Mutex);                                           // P3
  if (Etat[id] == Vide) {                                     // P4
     AttenteCons += 1;                                        // P5
     sem_post(&Mutex);                                        // P6
     sem_wait(&AttendrePlein);                                // P7
     sem_wait(&Mutex);                                        // P8
  }                                                           // P9
  item = Element;                                             // P10
  Etat[id] = Vide;                                            // P11
  if (TamponVide() && AttenteProd > 0) {                      // P12
     AttenteProd -= 1;                                        // P13
     sem_post(&AttendreVide);                                 // P14
  }                                                           // P15
  sem_post(&Mutex);                                           // P16
  return item;                                                // P17
} /* fin de Preleve */                                       // P18
```

PCO TE2 05.06.2012

3.5

1	1
2	1
3	0.2
	2.2

Problème 1

code avant //E1

int nbEnAttente[2] = {0,0}; // on compte le nb de threads en attente d'accès.

les lignes E4 et E5 deviendront

désolé c'était juste.

```
while (de dans [1-classe] > 0 || nbEnAttente[1-classe] > 0)
{
    nbEnAttente[classe]++;
    pthread_cond_wait(&attente[classe], &protege);
    nbEnAttente[classe]--;
}
```

présente un interblocage 1/2

le reste change pas.

on ne fait pas de l'attente active on bloque juste quand il faut.

Problème 3

1ère partie: illustration du problème 0.2/2
2e partie: solution

il y a un problème effectivement ligne D13 à D15 car on réveille jusqu'à attenteCons

or le tableau état peut être à vide au début

oui, mais on veut réveiller uniquement ceux en attente.

exemple attenteCons = 4

tabEtat

vide	vide	vide	vide	Plein	Plein	...

et on va réveiller les 4 premiers or leur tampon respectif n'a pas encore été rempli ?

Par arriver à D13, il faut avant faire D4 et donc tous les états sont vides et puis deviennent pleins à D11.

pour corriger le problème (bon il y en a peut-être d'autres mais à première vue je n'en vois pas) il faudrait déjà les réveiller tous donc ligne D14 i < N

pourquoi il ne sont pas forcément en attente.

et pour contrôler le flux des consommateur on remplace le if par un while ligne P4 en gardant la condition. cette solution n'est pas optimisé car on les réveille tous, mais elle fonctionne. non le problème de l'interblocage est toujours présent.

1

Question 2

comme il y a plusieurs consommateurs, il faut mettre en place une exclusion mutuelle entre eux.

ligne avant I1
sem_t mutex ;

ligne I4-I5
sem_init(&mutex,0,1);

entre ligne R1 et R2
sem_wait(&mutex);

entre ligne R14 et R15
sem_post(&mutex);

✓ 1/1

vide

vide