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

Contrôle continu du mardi 17 avril 2012 de 13h50 à 14h35

Remarques :
- Ce contrôle comprend **4** questions.
- Aucune documentation permise.
- Répondez **directement** sur l'énoncé.
- N'utilisez pas de couleur rouge.

Nom : .....

Question	Résultat
1	0.8
2	1
3	1
4	0.3
Total	3.1
Note	4.3

Question 1 (0,8 point) 0.8/0.8

Répondez aux questions posées.

(1) Parmi les états possibles d'une tâche (thread), il y a l'état dit *bloqué*. Pourquoi faut-il cet état? (0,4 point)

l'état dit bloqué se présente quand un thread est en exécution et qu'il se bloque sur un verrou ou un mutex par exemple. Cet état permet d'arrêter momentanément l'exécution d'un thread (mise en attente). On pourra le réveiller par la suite pour qu'il puisse aller à l'état "prêt"

OK

(2) Pour un problème multi-threadé et ayant une solution par sémaphores, est-il aussi possible de réaliser ce problème uniquement avec des verrous POSIX? Justifiez votre réponse. (0,4 point)

oui, c'est possible! il suffira juste de faire les blocages avec les verrous et donc de créer les fonctions d'initialisation, d'incrémentation et décrémentation d'une variable partagée* et également une fonction pour détruire ce que l'on a créé à la fin du programme.

OK

* fonction P() et V() correspondant aux sémaphores.

Question 2 (1,8 points)

Nous avons les 2 tâches POSIX données ci-dessous. La tâche `Interface` se charge de lire 2 grandeurs physiques (températures, débits, etc.) depuis 2 capteurs différents. La tâche `Control` détermine si les 2 grandeurs lues diffèrent suffisamment pour déclencher une alarme.

1/1.8

```
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <pthread.h>

volatile int Mesure[2];

void *Interface(void *arg) {
   while (true) {                                    // I.1
      Mesure[0] = lire la valeur du 1er capteur      // I.2
      Mesure[1] = lire la valeur du 2e capteur       // I.3
   }                                                 // I.4
   return NULL;                                      // I.5
} /* fin de Interface */

void *Control(void *arg) {
   while (true) {                                    // C.1
      sleep(1);                                      // C.2
      if (Different(Mesure[0],Mesure[1]))            // C.3
         DonnerAlarme();                             // C.4
   }                                                 // C.5
   return NULL;                                      // C.6
} /* fin de Control */

int main(void) {
   pthread_t control, interface;
   if (pthread_create(&interface,NULL,Interface,NULL) == 0) {
      if (pthread_create(&control,NULL,Control,NULL) == 0) {
         pthread_join(interface,NULL);
         pthread_join(control,NULL);
         return EXIT_SUCCESS;
      }
   }
   return EXIT_FAILURE;
} /* fin de main */
```

Les fonctions `Different` et `DonnerAlarme` ne sont pas nécessaires pour la compréhension de cette question, et leurs temps d'exécution ne sont pas connus.

(1) Ce programme n'est pas correct. Pourquoi? (0,3 point) (1 phrase suffit)

non pas du tout correct car Mesure[2] est partagé par les 2 tâches Control et Interface et chacune lit et écrit dessus. il n'y a pas d'exclusion mutuelle. risque de lire des valeurs erronée à un moment donné. OK 0.3

mais soyez précis.

Question 2 (suite)

(2) Comment faut-il le corriger? (0,7 point) (Les points sont attribués au code modifié et pas aux explications.)

juste en dessus de la déclaration de mesure[2] on écrit :

```
pthread_mutext sync = PTHREAD_MUTEX_INITIALIZER;
```

entre ligne I1 et I2 :

```
if ( pthread_mutex_trylock (& sync) ==0) {
```

entre ligne I3 et I4 :

```
pthread_mutex_unlock (& sync); }
```

entre ligne C3 :

```
if ( pthread_mutex_trylock (& sync) ==0) {
```

entre ligne C4 C5 :

```
pthread_mutex_unlock (& sync) ; }
```

OK, mais trylock introduit une attente active.

Il faut aussi initialiser Mesure car il n'est pas impossible que Control lise les valeurs de Mesure avant que Interface les affecte pour la 1ère fois. (cas d'un système surchargé) (pénaliser à la 2.3)

0.7/0.7

(3) Nous souhaitons modifier le code corrigé (c.-à-d. votre réponse au point (2) de cette question) pour que la tâche `Control` ne traite qu'une seule fois le dernier couple de mesures. Autrement dit, si la tâche `Interface` est plus rapide que `Control` et qu'elle a effectué 3 itérations alors que `Control` se trouvait sur la ligne C.2, `Control` traitera uniquement la 3e lecture des valeurs des capteurs (lignes I.2 et I.3) et ceci qu'une seule fois. Comment faîtes-vous cette modification? (Une explication peut être fournie mais les points sont attribués au code modifié.) (0,8 point)

bin en fait y a rien à modifier c'est le fait de mettre un trylock et ça prendra toujours la dernière valeur des capteurs

pas vrai.

Question 2 (suite)

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`).

Rappel sur les verrous Posix :
- `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER` : déclare et initialise le verrou `m` à l'état déverrouillé;
- `pthread_mutex_init(&m)` : initialise le verrou `m` à l'état déverrouillé;
- `pthread_mutex_lock(&m)` : permet d'obtenir l'accès au verrou `m`;
- `pthread_mutex_unlock(&m)` : permet de relâcher le verrou `m`.

réponse coincée en bas sur page précédente désolé ✓

autre version mais avec semaphore. sert à qui? (ne répondra pas la question si interface incrément alors que Control décrémente)

à faire si le temps ... plus le temps.

0/0.8

Question 2 (suite)

vide

Question 3 (1 point)

En s'inspirant de l'algorithme d'exclusion mutuelle de Dekker, nous pouvons concevoir les 2 procédures ci-dessous pour réaliser une exclusion mutuelle entre **3** tâches (*threads*). La procédure `Prelude` précède une section critique, alors que `Postlude` libère cette section critique.

```
#include <stdbool.h>
volatile bool etat[3] = {false,false,false};   // 1
volatile int tour = 0;                         // 2
                                               // 3
void Prelude(int id)                           // 4
{ // id = 0, 1, ou 2                           // 5
  etat[id] = true;                             // 6
  while (etat[(id+1)%3] || etat[(id+2)%3])     // 7
     if (tour != id) {                         // 8
        etat[id] = false;                      // 9
        while (tour != id)                     // 10
           ;                                   // 11
        etat[id] = true;                       // 12
     }
} /* fin de Prelude */
                                               // 13
void Postlude(int id)                          // 14
{ // id = 0, 1, ou 2                           // 15
  tour = (id + 1) % 3;                         // 16
  etat[id] = false;
} /* fin de Postlude */
```

boucle d'attente.

L'exclusion mutuelle est-elle préservée pour une section critique commune entre les 3 tâches? Justifiez votre réponse. (Il n'est pas demandé de vérifier les autres propriétés, telles que l'équité, etc.)
(Les tâches ont des identificateurs uniques compris entre 0 et 2, et chaque section critique est immédiatement précédée par `Prelude` et immédiatement suivie par `Postlude`.)

tour permet de décider qui quittera la boucle d'attente.

tache id	tour prend la valeur au postlude
0	1
1	2
2	0

il ya exclusion mutuelle si toutes les tâches sont dehors de la section critique.
maintenant s'il y en a une dans la section critique.
l'exclusion mutuelle est ok aussi car tour change dans le postlude. même si il ya changement de contexte après la ligne 15 ça n'empêchera pas un des 2 autres tâches entrer en section critique.
Mais est-ce que l'ex. mut est préservée ? ok.

1/1

Question 4 (1,4 points)

Un verrou réentrant est un verrou qui permet à une tâche (*thread*) possédant déjà le verrou de le redemander sans que cette tâche s'interbloque. Par exemple, si nous avons le fragment de code ci-dessous,

```
VERROU v;                // 1
      ...
Verrouille(&v);          // 2
      ...
   Verrouille(&v);       // 3
      ...
   Deverrouille(&v);     // 4
      ...
Deverrouille(&v);        // 5
```

le verrou `v` s'obtient à la ligne 2 et se libère à la ligne 5. Le thread appelant `Verrouille(&v)` à la ligne 3 n'est pas bloqué car il possède déjà le verrou, et pour respecter l'ordre des appels, le verrou n'est pas relâché à la ligne 4.

L'implémentation proposée ci-dessous réalise-t-elle la fonctionnalité souhaitée? Corrigez cette implémentation si ce n'est pas le cas.

```
typedef struct VERROU {
  pthread_mutex_t mutex;
  pthread_mutex_t verrou;
  int pris;
  pthread_t appelant;
} VERROU;

void Verrou_Init(VERROU *v) { // Initialise un enregistrement VERROU
  pthread_mutex_init(v->mutex);                          // 1
  pthread_mutex_init(v->verrou);                         // 2
  v->pris = 0;                                           // 3
} /* fin de Verrou_Init */

void Verrouille(VERROU *v) { // Demande le verrou v      // 4
  pthread_mutex_lock(v->mutex);                          // 5
  if (v->pris == 0)                                      // 6
     pthread_mutex_lock(v->verrou);                      // 7
  else if (v->appelant != pthread_self()) {              // 8
     pthread_mutex_unlock(v->mutex);                     // 9
     pthread_mutex_lock(v->verrou);                      // 10
     pthread_mutex_lock(v->mutex);                       // 11
  }                                                      // 12
  if (v->pris == 0)                                      // 13
     v->appelant = pthread_self();                       // 14
  v->pris += 1;                                          // 15
  pthread_mutex_unlock(v->mutex);                        // 16
} /* fin de Verrouille */

void Deverrouille(VERROU *v) { // Libère le verrou v     // 17
  pthread_mutex_lock(v->mutex);                          // 18
  if (v->pris > 0) {  // verrouillé?                     // 19
     v->pris -= 1;                                       // 20
     if (v->pris == 0)                                   // 21
        pthread_mutex_unlock(v->verrou);                 // 22
  }                                                      // 23
  pthread_mutex_unlock(v->mutex);                        // 24
} /* fin de Deverrouille */
```

Un rappel des verrous Posix se trouve à la fin de la question 2. La fonction `pthread_self()` retourne l'identifiant du thread appelant la fonction.

Question 4 (suite)

il y a bel et bien un problème.

parce que...

dans verrouille

- si c'est pas déjà pris on le prend
- si c'est déjà pris et que c'était moi alors ok je le prend encore une fois
- mais si c'était pas moi alors je dois me bloquer et attendre

dans diverrouille.

- si c'est verrouillé alors on "diverrouille" décrémente de 1 et si y a plus personne alors on fait le unlock

exclusion mutuelle ok

mais souci au niveau de appellant qui n'est pas géré dans le deverrouille. du coup un autre thread pourrait faire un unlock ?

pas clair bénéfice du doute 0.3 / 1.4

vide