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

Contrôle continu du mardi 19 avril 2011 de 11h15 à 12h00

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

Nom : . . . . . .

Question	Résultat
1	1.2
2	0.1
3	0.1
4	0.5
Total	1.9
Note	3.1

Question 1 (1,6 points)

1.2/1.6

Répondez aux questions posées.

(1) Qu'est-ce qu'une action atomique? (0,4 point)

c'est une action qui n'est pas divisible.
exemple : plusieurs instructions qui doivent s'exécuter à la suite sans être coupé (interrompue).

action indivisible, généralement une instruction au niveau de l'application

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

c'est quand un thread a fini son exécution. et c'est avant que l'on fasse une jointure pour passer à l'état terminé utile pour dire que il existe mais qu'il ne fait rien. ok
état transitoire permettant de retourner des valeur en retour... info dans cours.

(3) Dans le concept de la programmation concurrente, qu'est-ce qu'une préemption? De manière générale, est-il possible d'éviter une préemption? Justifiez votre réponse. (0,4 point)

0.2 c'est le passage de l'état élu à l'état prêt. ✓ c'est quand un thread recomme les actions qu'il a à faire (boucle while(true)). car particulier
on peut l'éviter si le thread ne s'exécute que une seule fois. non
se faire voler le processeur à son insu.
interrompt la tâche dans sont exécution.
non pas possible d'enlever les interruptions donc pas éviter les préemption
c'est l'ordonnanceur qui gère l'ordre d'exécution.

(4) 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)

0.2 oui cela est possible ✓ c'est plus dur à réaliser car les sémaphores peuvent avoir des valeurs entières donc bloquage plusieurs fois alors que verrous ont que true et false donc bloquer que 1 fois.
on fait les fonctions perso init. incrém. décrém verrouille. déverrouille en utilisant les verrou dedans.
ceci ne justifie pas votre réponse.

Question 2 (1,4 points)

En s'inspirant de l'algorithme d'exclusion mutuelle de Peterson, 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.

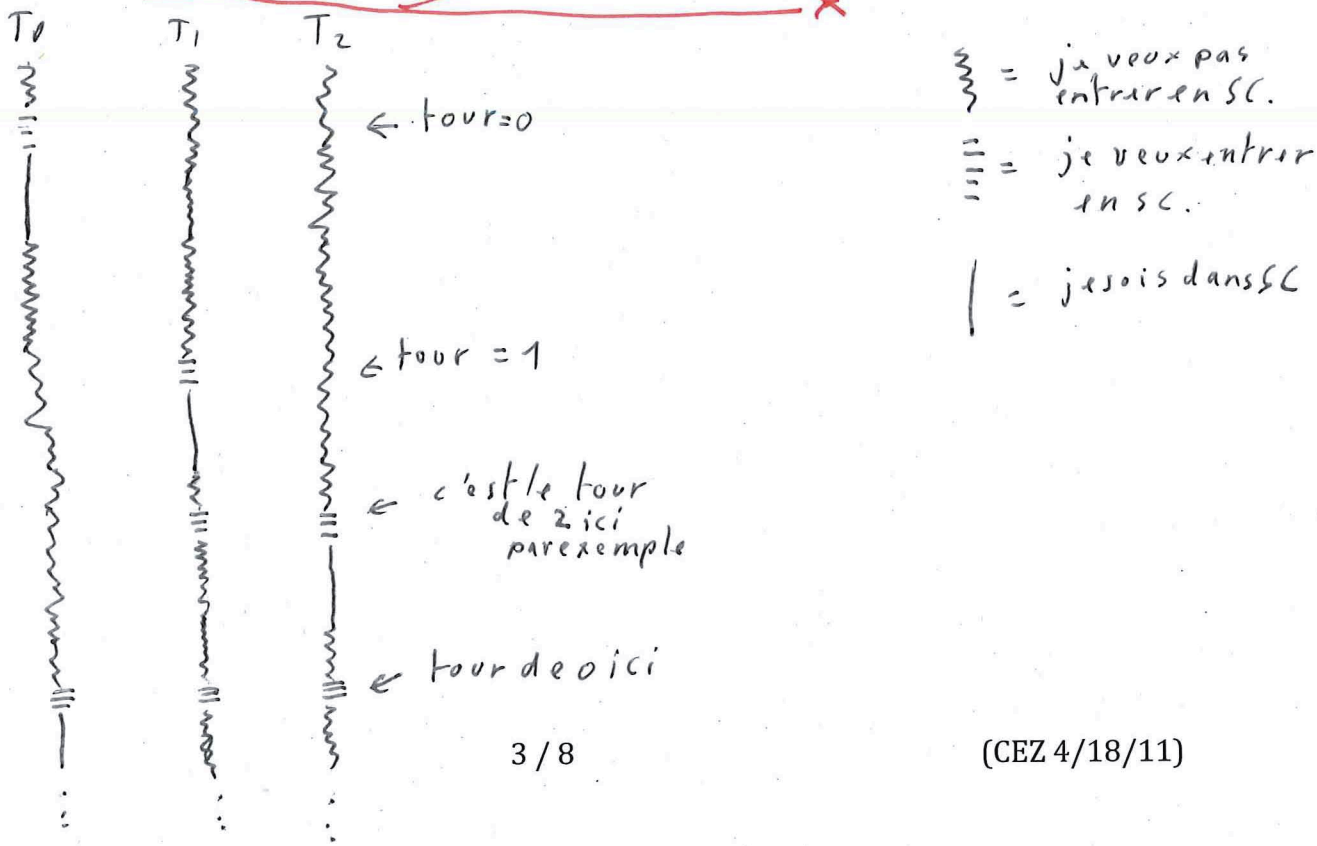
```
volatile int tour = 0;
volatile bool intention[3] = {false,false,false};

void Prelude(int id)
{ // id = 0, 1, ou 2
  intention[id] = true;                                                    //1
  tour = (id + 4) % 3;                                                     //2
  while ((intention[(id+4)%3] || intention[(id+2)%3]) && tour != id)        //3
    ;                                                                      //4
} /* fin de Prelude */

void Postlude(int id)
{ // id = 0, 1 ou 2
  intention[id] = false;                                                   //5
} /* fin de Postlude */
```

L'exclusion mutuelle est-elle préservée pour une section critique commune entre les 3 tâches? Justifiez votre réponse.
(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`.)

pour savoir on doit essayer de voir si y en a 2 ou 3 qui peuvent être en même temps dans la section critique. pour cela, les 3 doivent être sortis de la boucle while. et pour en sortir les 2 autres tâches (soit l'une soit l'autre) ne doivent pas vouloir rentrer et que l'id du tour soit différent du notre.
en plus comme l'id du tour sera 1 plus grand que notre id à nous.
alors oui l'exclusion mutuelle est garantie



Question 2 (suite)

donc 3 par exemple, tour peut prendre l'une des valeurs 0, 1 ou 2. donc 2 tâches voulant entrer

si y en a plusieurs qui veulent rentrer alors la variable tour départagera qui va entrer en section critique. pas toujours

au début du post lude on va indiquer notre intention de rentrer en S.C. et indiquer que c'est le tour de la tâche suivante avant de tester si on peut nous entrer en S.C.

donc si 0 et 1 veulent entrer tour peut prendre la valeur 1.

à la fin quand on fait le postlude on indique que l'on a plus envie d'entrer en S.C. car on vient d'en sortir.

il n'y pas grand chose de juste ici.

0 1.4

il y a d'autres cas de figure que nous n'avez pas considéré.

Principe poursuivi : Quand il y a un conflit pour l'accès (c'est à dire plusieurs tâches veulent entrer en S.C.), "tour" indique qui a le droit de sortir de sa boucle. Ainsi la tâche i positionne "tour" à

i	tour
0	1
1	2
2	0

Mais s'il y a déjà une tâche dedans, il faudrait aussi que les 2 autres tâches puisse mettre tour à l'indice de celui qui est dedans.

Ce qui n'est pas possible:
si la tâche 0 est dedans, les 2 autres tâches peuvent faire

tâche 1 tâche 2

intention[1] = true → intention[2] = true

tour = 0

tour = 2 → sort du

Question 3 (1,4 points)

où est cette procédure ?

Nous souhaitons réaliser une procédure qui met en couple des tâches (threads). Les tâches sont initialement réparties en 2 groupes et chaque groupe est composé d'un nombre quelconque de tâches. L'objectif poursuivi par la procédure

```
void MettreEnCouple(bool groupe)
```

est de coupler une tâche appartenant à un groupe avec une autre tâche de l'autre groupe. Quand une tâche appelle cette procédure, s'il n'y a pas de tâche appartenant au groupe inverse, cette tâche attend; sinon, la tâche réveille l'une des tâches du groupe inverse et les 2 tâches poursuivent leur exécution.

En utilisant uniquement des sémaphores Posix, proposez une implémentation de la procédure demandée. Votre implémentation doit aussi fournir une procédure permettant d'initialiser vos variables partagées.

Pour cette question, il n'est pas nécessaire de traiter les erreurs retournées par les fonctions `sem_init`, `sem_wait`, et `sem_post`.

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`).

Votre implémentation :

```
// tâche x (du groupe 0)
void * tacheX (void *arg)
{
   ...
   sem_wait(&s); //attend une tache
   mettreEnCouple(1);
   sem_post(&s);
   ...
}
```

```
//tâche Y (du groupe 1)
void * tacheX(void *arg)
{
   ...
   sem_wait(&s); //attend une tache
   mettreEnCouple(0);
   sem_post(&s);
   ...
}
```

s'il y a attente dans MettreEnCouple() comme c'est indiqué dans l'énoncé, il y a interblocage.

en gros c'est comme cela.

```
void init(void)
{ int n = 6; // 6 pour l'exemple
  sem_t s;
  sem_init(&s, 0, n);
}
```

s'il y a n tâches
à quoi sert le sémaphore car toutes les tâches peuvent appeller MettreEnCouple.

0,1 / 1,4

Question 3 (suite)

Question 4 (0,6 point)

Les verrous POSIX comprennent une fonction appelée `pthread_mutex_trylock` qui permet de verrouiller un verrou si et seulement si le verrou n'est pas encore verrouillé.
Pour cette question, *nous supposons que cette fonction `pthread_mutex_trylock` n'existe pas*, et l'objectif est de réaliser une telle fonction opérant sur un verrou.
L'implémentation proposée ci-dessous comporte un certain nombre d'erreurs et qu'il faut corriger afin d'obtenir le verrou ayant la fonctionnalité souhaitée.

Rappel sur les verrous Posix :
- `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER` : déclare et initialise le verrou `m` ;
- `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`.

```
typedef struct VERROU {
  pthread_mutex_t mutex;
  pthread_mutex_t verrou;
  int pris;
} 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 VerrouInit */

bool Verrou_Essaie(VERROU *v) {
  // Verrouille VERROU seulement s'il est disponible et
  // le laisse tel quel autrement (correspond à trylock).
  pthread_mutex_lock(v->mutex);                                  // 4
  if (v->pris > 0) {                                             // 5
     pthread_mutex_unlock(v->mutex);                             // 6
     return false; // indique que VERROU n'est pas obtenu       // 7
  }                                                              // 8
  else {                                                         // 9
  v->pris += 1;                                                  // 10
     pthread_mutex_unlock(v->mutex);                             // 11
     pthread_mutex_lock(v->verrou);                              // 12
     return true; // indique que VERROU est obtenu               // 13
  }
}

void Verrou_Verrouille(VERROU *v) {
  // Verrouille inconditionnellement VERROU (correspond à lock).
  pthread_mutex_lock(v->mutex);                                  // 14
  v->pris += 1;                                                  // 15
  pthread_mutex_lock(v->verrou);                                 // 16
  pthread_mutex_unlock(v->mutex);                                // 17
}

void Verrou_Deverrouille(VERROU *v) {
  // Déverrouille VERROU (correspond à unlock).
  pthread_mutex_lock(v->mutex);                                  // 18
  v->pris -= 1;                                                  // 19
  pthread_mutex_unlock(v->verrou);                               // 20
  pthread_mutex_unlock(v->mutex);                                // 21
}
```

on a oublié partout de passer par pointeur quand on fait des init lock unlock.

non car pris=0 => passage à 1.

1) échanger ligne 11 et 12 pour que le pris += 1 et le lock du verrou soit atomiques. 0.4

2) l.10 += 1 pour dire que y en a 1 de plus

3) à chaque fois que l'on utilise la variable "pris" on doit mettre V -> pris car c'est un champs de la struct. et pas juste une variable globale. 0.1

0.5
0.6

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

Solutions possibles de la question 3 du contrôle continu du mardi 19 avril 2011

Solution 1

```
sem_t attente[2], mutex;
int nbAttente[2];

void Initialise1(void) {
  int i;
  for (i = 0; i < 2; i += 1) {
     nbAttente[i] = 0;
     sem_init(&attente[i],0,0);
  }
  sem_init(&mutex,0,1);
}

void MettreEnCouple1(bool groupe) {
  int i = !groupe; // Groupe inverse
  sem_wait(&mutex);
  if (nbAttente[i] > 0) {  // Membre autre groupe présent?
     nbAttente[i] -= 1;    // OUI: le réveiller
     sem_post(&mutex);
     sem_post(&attente[i]);
  }
  else {
     nbAttente[1-i] += 1;  // NON: attendre
     sem_post(&mutex);
     sem_wait(&attente[1-i]);
  }
}
```

Solution 2

```
sem_t attente[2];  // Compteur de présence et attente

void Initialise2(void) {
  sem_init(&attente[0],0,0);
  sem_init(&attente[1],0,0);
}

void MettreEnCouple2(bool groupe) {
  int i = !groupe; // groupe inverse
  sem_post(&attente[!i]);
  sem_wait(&attente[i]);
}
```

Solution 3

```
sem_t attente, mutex;
int nbAttente;
bool groupeEnAttente;

void Initialise3(void) {
  nbAttente = 0;
  sem_init(&attente,0,0);
  sem_init(&mutex,0,1);
}

void MettreEnCouple3(bool groupe) {
  bool g = !groupe;   // Groupe inverse
  sem_wait(&mutex);
  if (nbAttente > 0 && g == groupeEnAttente) {
     nbAttente -= 1;  // 1 membre du groupe inverse est présent
     sem_post(&mutex);
     sem_post(&attente);
  }
  else { // Attendre sur un membre du groupe inverse
     nbAttente += 1;
     groupeEnAttente = !g; // Indique le groupe si nbAttente=1
     sem_post(&mutex);
     sem_wait(&attente);
  }
}
```

Exercice supplémentaire

Nous souhaitons reprendre le même problème, mais cette fois-ci, l'appelant de la fonction `MettreEnCouple` devrait connaître l'identifiant de la tâche avec laquelle il forme un couple :

```
pthread_t MettreEnCouple(bool groupe);
```

L'identifiant d'un thread s'obtient par la fonction `pthread_t pthread_self(void)`.