

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	1.3
2	0.6
3	0
4	0
Total	1.9
Note	3.1

+1

+ 0.2

Question 1 (1,6 points)

Répondez aux questions posées.

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

Une action qui n'est pas divisible.

1.3
1.6

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

Il est nécessaire lorsque le processus a un parent et que la terminaison correcte du thread est importante. ?
que signifie "correcte" ici

-0.1

(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)

Une préemption arrive lorsque l'ordonnanceur remet le thread à l'état prêt sans que le processus n'en soit conscient.

Il est possible de l'éviter par la synchronisation des tâches : le thread passe à un état endormi en attendant un événement ou le relâchement d'un mutex.
ou {
oui mais le but normalement voulu est de vouloir "monopoliser" le processeur au détriment des autres tâches. Votre réponse est équivalente à terminer une tâche pour empêcher d'être préempté.

(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 ✓ car un verrou est un sémaphore binaire. Il suffit d'ajouter une variable comptant le nombre de thread.

(non il faut aussi d'autres verrous)

non il faut aussi autre chose.

La réponse souhaitée est : " car il est possible de simuler un sémaphore et ses différentes fonctions par des verrous, comme c'est fait dans les notes de cours. "

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*.)

Si *intention[0]* est à *true* et que
intention[1] et *intention[2]* le sont aussi

T0
intention[0] = true
tour = 1
while (intention[0] || intention[2])
 && tour != 0

T1
intention[1] = true
tour = 2
while (intention[2] || intention[0])
 && tour != 1

T2
intention[2] = true
tour = 0
while (intention[0] || intention[1])
 && tour != 2

S'il y a un changement de contexte entre
tour et while de T0 puis T1 puis T2 :

intention[0] true
tour = 1

intention[1] = true
tour = 2

intention[2] = true
tour = 0

while ...
sort de la boucle
rentre en SC

while ...
sort de la boucle
rentre en SC

exclusion non
garantie ✓

Justification
0.2/1

3/8 ~~tour=0 ≠ 1~~
donc toujours
dans la boucle.

0.4

(CEZ 4/18/11)

MettreEnCouple(~~bool~~ groupe)

if (attente[!groupe] == 0

 attente[groupe]++;
 "attendre"

else

 attente[!groupe]--;
 "réveiller"

if (groupe != 'A') {

 if (attenteGripB == 0)

 attenteGripA++;
 attenteA();

else

 attenteGripB--;
 reveillerB();

{ else }

int a;

void toto() {
 for (int i = 0; i < 3; i++)
 a++; // ~~i++~~
}

~~bo~~ int attente[?];
sem_t attenteGrip[?];
sem_t mutex;

init() {

0, 1, 2, 3, ..., n-1, 1, 2, 3, ..., n, ②
↓

Question 3 (1,4 points)

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.

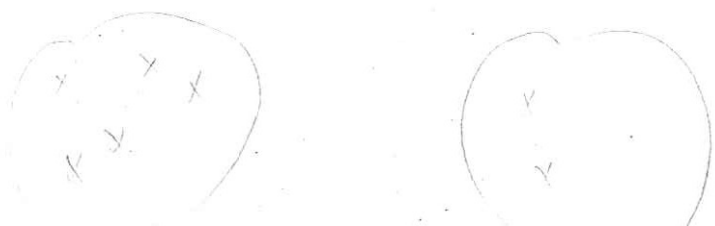
En utilisant uniquement des sémaphores Posix, proposez une implémentation de la procédure demandée. Votre implémentation doit aussi fournir une procédure permettant d'initialiser vos variables partagées.

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 :



```

int n = min(nbGroupe1, nbGroupe2)
bool groupe = false
sem_t[nbGroupe1] sem1;
sem_init(&sem1[ ], 0, );
sem_t[nbGroupe2] sem2;
  
```

doit être dans une fonction

ce n'est pas de cette façon qu'un tableau en C se définit.

```

void MettreEnCouple(bool groupe) {
  
```

Il n'y a pas grand chose ici où je peux vous donner des points

0/14

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
    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 (pris > 0) {                            // 5
        pthread_mutex_unlock(v->mutex);       // 6
        return false; // indique que VERROU n'est pas obtenu // 7
    }
    else {                                     // 8
        pris = 1;                             // 9
        pthread_mutex_unlock(v->mutex);       // 10
        pthread_mutex_lock(v->verrou);        // 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
    pris += 1;                               // 15
    pthread_mutex_lock(v->verrou);            // 16
    pthread_mutex_unlock(v->mutex);           // 17
}

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

pas fait.

Corrections du test n°1

Question 1

3) Prémption → se faire voler le processeur à son insu. Transition pour interrompre la tâche en cours de son exécution.

Non il n'est pas possible de les éviter à moins d'être super-user d'un super OS créé par nos soins.

Question 2

Principe poursuivi : quand il y a un conflit pour l'accès (c'est-à-dire plusieurs tâches voulant 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 deux autres tâches puissent mettre "tour" à l'indice de celui qui est dedans.

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

tâche1		tâche2
intention[1] = true	→	intention[2] = true
		tour = 0
tour = 2	→	sort du while !

Question 3

Exercice supplémentaire :

attendre sur l'autre → écrire son pthread_t → attendre que l'autre écrive son pthread_t →

lire le pthread_t de l'autre → sortir

```
pthread_t MettreEnCouple(bool groupe) {
    pthread_t autrePid;
    sem_wait(&mutex);
    if (nbAttente[!groupe] > 0) {        // il faut réveiller le membre
        pid = pthread_self();
        sem_post(&pidPret[!groupe]);
        sem_post(&attente[!groupe]);
        sem_wait(&pidPret[groupe]);
        autrePid = pid;
        sem_post(&mutex);
    }
    else {
        nbAttend[groupe]++;
        sem_post(&mutex);
        sem_wait(&attente[groupe]);
        autrePid = pid;
        pid = pthread_self();
        sem_post(&pidPret);
    }
}
```

```

    }
    return autrePid;
}
sem_t attente[2], mutex, pidPret;
unsigned nbAttente[2];
pthread_t pid;

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

```

Question 4

On essaye de simuler le comportement d'un verrou qui aurait, en plus, un trylock. On ne doit pas attendre pour savoir si le verrou est pris ni attendre pour l'obtenir car on risque de se faire passer devant.

On compte le nombre de tâches en attente...

- pris est un champ de la structure de données → pris est considéré comme non déclaré : v->pris
- croiser 16 et 17 car sinon on a une attente de libération du mutex
- il peut y avoir un changement de contexte entre 11 et 12 → inverser 11 et 12 car sinon le verrou peut être pris à l'insu.

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 *stupide mais directe*

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

void Initialiser1(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 Initialiser2(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)`.