Prof. Yann Thoma

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

# Programmation concurrente 1 (PCO1)
## semestre printemps 2009-2010
## Contrôle continu 1
## 19.04.2010

4,2

Prénom:

Nom

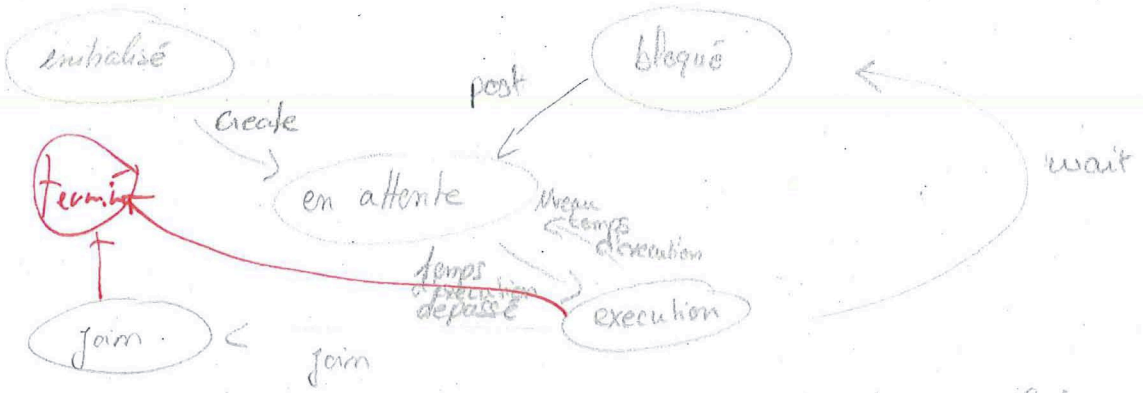
- Aucune documentation n'est permise, y compris la feuille de vos voisins

**Question 1:** (10 pts) 6

1. Listez les états d'une tâche, et indiquez, pour chaque état, quel est sa signification.
2. Dressez le graphe du cycle de vie d'une tâche, en indiquant ce qui provoque les transitions.
3. Dans ce contexte, que se passe-t'il lors de l'exécution d'un `lock(&mutex)` si le mutex est fermé ?
4. Toujours dans ce contexte, que se passe-t'il lors de l'exécution de `unlock(&mutex)` ?

1) Une tâche peut être :
- en attente : attend de pouvoir s'executé (une autre tâche s'exécute)
- en execution : son code est entrain de s'executer
- initialisé : On vient de créer cette tâche
- bloqué : Attend une condition precise
...

2)




3) La fonction qui effectue ce code devient bloqué et s'ajoute à la liste d'attente, dans l'ordre d'arrivé jusqu'à obtenir un unlock (&mutex) On passe de l'état d'execution à l'état bloqué

4) La première fonction tâche dans l'attente est debloqué (état bloqué à en attente).

true true
false false
true true
etat (false false) 0 1

**Question 2:** (10 pts) 10

Soit le listing suivant :

```
0  bool etat[2] = {false,false};
1  void *T(void *arg)
2  {
3    int t=(int)arg;
4    while (true) {
5      etat[t] = true;
6      while (etat[1-t]) {
7        etat[t] = false;
8        while (etat[1-t])
9          ;
10       etat[t] = true;
11     }
12     /* section critique */
13     etat[t] = false;
14     /* section non-critique */
15   }
16 }
```

true true
false false
true true

Deux tâches sont créées avec comme argument 0 et 1. Est-ce que l'accès à la section critique est géré correctement ? Cochez le(s) case(s) posant problème et commentez.

| Exclusion mutuelle | Interblocage | Famine | Couplage trop fort |
|---|---|---|---|
| | X | X | |

Si oui, démontrez que c'est le cas. Si non, exposez un scénario illustrant le problème.

Famine : la ligne 6 test l'état de l'autre tâche. Ainsi, si la tâche 1 laisse passer la tâche 2, et que celle ci s'execute en continue jusqu'a la ligne 5, la tache 2 s'executera à nouveau sans que la tache 1 ne se soit executé.

Interblocage : Si les tâches 1 et 2 s'executent exactement en même temps de la ligne 4 à la ligne 11 (ligne 4 tache 1, ligne 4 tache 2, ligne 5 tache 1, ligne 5 tache 2, etc)
les 2 tâches n'arriveront jamais en section critique.

Exclusion mutel : Cela correspond au fait que deux taches puissent acceder une même ressource en même temps.

Couplage trop fort : Il faut obligatoirement que les deux tâches s'execute. L'une d'elles ne pourra pas continuer sans l'autre (synchro par exemple)

## Question 3: (10 pts) 6

Modifiez le programme suivant pour que le programme principal lance l'exécution du traitement des threads après un certain temps, et que les deux instructions `a=a+y` et `a=a+w` s'exécutent avant les deux instructions `b=b+z` et `b=b+v`. Vous devez évidemment garantir le bon fonctionnement du programme. Le tout doit se faire en utilisant des verrous.

Rappel sur les verrous Posix :

- `pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER` : correspond à initialiser le verrou comme étant ouvert ;
- `pthread_mutex_lock(&m)` : verrouille le verrou s'il était ouvert, ou bloque l'appelant sinon ;
- `pthread_mutex_unlock(&m)` : déverrouille le verrou s'il était fermé ou réveille une tâche en attente s'il en existe une.

```
int a,b,v,w,y,z;
// déclarations
pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER; // ouvert
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER; // ouvert
pthread_mutex_t debut = PTHREAD_MUTEX_INITIALIZER; // ouvert


void *fonctionA(void *arg) {
  pthread_mutex_lock(&debut);
  pthread_mutex_unlock(&debut);

  a=a+y;
  pthread_unlock(&mutexB);
  pthread_lock(&mutexA);

  b=b+z;

}

void *fonctionB(void *arg) {
  pthread_mutex_lock(&debut);
  pthread_mutex_unlock(&debut);

  a=a+w;
  pthread_mutex_unlock(&mutexA);
  pthread_mutex_lock(&mutexB);

  b=b+v;

}

int main (int argc, char *argv[])
{
  pthread_t threadA,threadB;
  pthread_mutex_lock(&mutexB); // fermé
  pthread_mutex_lock(&mutexB); // fermé
  pthread_mutex_lock(&debut); // fermé

  pthread_create(&threadA, NULL, fonctionA, NULL);
  pthread_create(&threadB, NULL, fonctionB, NULL);
  usleep(10000000);
  // lancement de l'exécution
  pthread_mutex_unlock(&debut);

  pthread_join(threadA,NULL);
  pthread_join(threadB,NULL);
  return 0;
}
```

Synchro avec timer

attend sur la fct B

exclusion mutuelle !

attend sur fonct° A

## Question 4: (20 pts) 7C

Nous désirons modéliser une file d'attente à un guichet CFF. La file est tout d'abord unique, puis se sépare en deux, pour atteindre deux guichets. Devant chaque guichet, jusqu'à 5 personnes peuvent attendre. Lorsqu'une nouvelle personne arrive, si la file unique est occupée, elle se place en fin de file. Si la file unique est vide, elle se place dans la plus petite file présente devant un guichet. Lorsqu'un client arrive au guichet, il libère évidemment une place dans la file de ce guichet et une personne de la file principale peut y prendre place.
Proposez l'implémentation des fonctions `accesGuichet()` et `quitteGuichet(int guichet)`, ainsi que `initialiseGuichet()`, à l'aide de sémaphores. La fonction `accesGuichet()` retourne le numéro de guichet que la personne aura atteint (0 ou 1). Ce numéro doit être passé en paramètre de `quitteGuichet(int guichet)`. (placez votre code sur une des pages à disposition).

Rappel sur les sémaphores Posix :
- `sem_init(&a,0,n)` : correspond à initialiser le sémaphore a à n (n doit être $\geq 0$) ;
- `sem_wait(&a)` : décrémente a puis bloque l'appelant si `a < 0` ;
- `sem_post(&a)` : incrémente a puis réveille une tâche si $a \leq 0$.

par le sem

```
#define NUM_THREADS 100

// déclarations à compléter
// Exemple: sem_t semaphore;
void initialiseGuichet() {
  // à compléter
}

int accesGuichet() {
  // à compléter
}

void quitteGuichet(int guichet) {
  // à compléter
}

void *client(void *arg) {
  int guichetatteint;
  while(true) {
    guichetatteint=accesGuichet();
    usleep(1000);
    quitteGuichet(guichetatteint);
    usleep(1000);
  }
}

int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int t;
  initialiseGuichet();
  for(t=0; t<NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, client, NULL);
  for(t=0; t<NUM_THREADS; t++)
    pthread_join(threads[t],NULL);
  pthread_exit(NULL);
}
```

sem_t mutex;
sem_t fileUnique, sem_t file1, sem_t file2;
sem_init(&fileUnique, 0, NUM_Thread);
sem_init(&file1, 0, 5);
sem_init(&file2, 0, 5);
int NbClientFile1 = 0;
int NbClientFile2 = 0;
sem_init(&mutex, 0, 1);
int NbClientFileCommune

72

```
int accessGuichet() {
    sem_wait(&fileUnique); // On attend notre tour dans une file
    // On a été réveillé
    sem_wait(&mutex);
    if (nbClientFile1 < 5) {          <- on remplit d'abord la 1ère file jusqu'à la dernière
        nbClientFile1++; // On s'ajoute
        sem_post(&mutex); // on relâche le mutex
        sem_wait(&file1); // on attend notre tour
        sem_post(&guichetUnique);
    }
    else if (nbClientFile2 < 5) {
        nbClientFile2++;
        sem_post(&mutex);
        sem_wait(&file2);
        sem_wait(&mutex);
        nbClientFile2--;
        sem_post(&guichetUnique);
        sem_post(&mutex);
        return 2;
    }
    return Erreur;
}

void quitteGuichet(guichetAtteint
{
    if (guichetAtteint == 1)
    {

    }
    else
    {

    }
}
```

```
sem_wait(&mutex);
nbClientFile1--;
sem_post(&mutex);
return 1;
```

Si on fait ça car le reste est dans quitte guichet

il faut des post sur file1 et file2