Prof. Yann Thoma

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

Programmation Concurrente (PCO)
Semestre printemps 2014-2015
Contrôle continu 2
08.06.2015

Prénom: VALENTIN D.

Nom: MINDER

— Aucune documentation n'est permise, y compris la feuille de vos voisins
— La calculatrice n'est pas autorisée
— Aucune réclamation ne sera acceptée en cas d'utilisation du crayon
— Ne pas utiliser de couleur rouge

Question	Points	Score
1	20	15
2	20	20
3	20	14
Total:	60	49

Note: 5,1

Question 1: 20 points

Considérons une ligne de trams comportant une section critique, c'est-à-dire un tronçon du réseau à voie unique. Simuler à l'aide de ~~sémaphores~~ variables conditions le contrôle de cette partie de voie, en assurant les points suivants :

— Un tram ne peut s'engager sur la voie unique que si cette dernière est libre ou si celui-ci se dirige dans le même sens que les autres trams déjà présents sur ce tronçon ;
— Le nombre de trams présents simultanément sur ce tronçon de voie est limité à la valeur passée au constructeur ;
— Lorsqu'un tram quitte la voie unique, si d'autres trams sont en attente pour la même direction, l'un d'eux pourra y accéder, sinon un tram venant en sens inverse pourra le faire.

Complétez le code suivant, en vous basant sur les variables conditions. Les fonctions `acces` et `sortie` prennent en paramètre le sens d'arrivée du tram, qui peut prendre la valeur 0 (arrivée depuis le Nord) ou 1 (arrivée depuis le Sud).

Rappel sur les ~~sémaphores Posix~~ variables conditions QT :
— `cond.wait(&mutex)` : Mise en attente sur la condition ;
— `cond.wakeOne()` : Signale la condition.

```
class Troncon
{
public:
    Troncon(int maxTrams);
    virtual ~Troncon();
    void acces(int sens);
    void sortie(int sens);
}
```

#include <QWaitCondition>
#include <QMutex>

```
class TronconImpl : public Troncon {
private:
    int maxTrams;
    int currentDirection = -1; // -1 : libre, 0 : depuis le Nord, 1 : depuis le Sud
    QMutex* mutex; // protège currentDirection et currentNbTrams
    int currentNbTrams = 0;
    QWaitCondition*[] waitIsDir; // tableau de QWaitCondition
                                 // attend que les trams arrivent
                                 // depuis le Nord : [0], depuis le Sud : [1]

public:
    TronconImpl(int maxTrams) : maxTrams(maxTrams) {
        //équivalent à : this->maxTrams = maxTrams;
        mutex = new QMutex(); waitIsDir = new QWaitCondition[2];
        waitIsDir[0] = new QWaitCondition();
        waitIsDir[1] = new QWaitCondition();
    }
    virtual ~TronconImpl() {
        delete mutex;
        delete waitIsDir;
    }
}
```

```
virtual void acces (int sens) {
    mutex -> lock();
    if (currentNbTrams == 0) {          // premier tram : on accède à la voie
        currentDirection = sens;
        currentNbTrams ++;
        mutex -> unlock();
    } else if (currentNbTrams < maxTrams &&
               currentDirection == sens) {   // tram suivant dans le même sens et il y a de la place -> on accède
        currentNbTrams ++;
        mutex -> unlock();
    } else {                            // sinon, on ne passe pas, on se met en attente sur la waitCondition qui correspond au sens de parcours
        dans un while() -> waitIsDir[sens].wait(&mutex);
        currentDirection = sens;        // on a hérité son sens du tunnel (important, car on pourrait être le premier après un changement de sens ...
        currentNbTrams ++;
        mutex -> unlock();
    }
}
virtual void leave (int sens) {
    mutex -> lock();
    currentNbTrams --;
    if (currentNbTrams == 0) {          // s'il n'y a plus de trams sur la voie : on libère éventuellement des trams de l'autre sens (au max maxTrains) car ils peuvent s'engager à leur tour
        currentDirection = 1-sens;      // plus simple de le faire ici : le changement de sens intervient quand le dernier train quitte.
        for(int i=0; i<maxTrains; i++) {
            waitIsDir[1-sens].wakeOne();
        }
    } else {                            // si on était pas le dernier sur la voie, on libère éventuellement un tram de son propre sens
        waitIsDir[sens].wakeOne();
    }
    mutex -> unlock();
}
```

Question 2: 20 points

Nous désirons réaliser une barrière de synchronisation un peu particulière. Des threads de deux types (0 et 1) vont attendre sur cette barrière, et lorsque le Nième thread arrive, alors les threads en attente doivent être relâchés, selon un ordre particulier. Ecrivez le code implémentant la classe `BarriereAB` en exploitant des variables conditions. La valeur passée au constructeur correspond à la taille de la barrière. Les $n-1$ premiers threads doivent être mis en attente, et le $n$ième relâche les autres. Le relâchement doit se faire de manière à d'abord relâcher tous les threads de la même classe que le $n$ième puis tous les autres.

Vous avez en plus une contrainte supplémentaire : un thread ne peut appeler qu'une seule fois la fonction `wakeOne()` (et il est interdit d'utiliser `wakeAll()`).

```
class BarriereAB
{
public:
    BarriereAB(int n);
virtual void arrive(int classe);
}
```

virtual ~BarriereAB();

voir explications plus loin.

```
class BarriereABImpl : public BarriereAB {
private:
    int n;
    int[] currentWaiting;          // tableau de compteur nombre de thread en attente sur chaque classe
    QMutex * mutex;
    QWaitCondition[] * waitForClass;   // tableau de QWaitCondition [0] : QWait sur la classe 0
                                       [1] : idem, classe 1
public:
    BarriereABImpl(int n) : n(n) {   // équiv à : this->n = n
        mutex = new QMutex();
        waitForClass = new QWaitCondition[2];
        waitForClass[0] = new QWaitCondition();
        waitForClass[1] = new QWaitCondition();
        currentWaiting[0] = currentWaiting[1] = 0;
    }
virtual ~BarriereABImpl() {
        delete mutex;
        delete waitForClass;
    }
virtual void arrive(int classe)
1.  mutex->lock();
2.  currentWaiting[classe]++;
3.  if (currentWaiting[0] + currentWaiting[1] < n) {   // si le total de thread en attente (des 2 types) n'atteint pas n, on se met en attente sur le QWait de sa classe
4.      waitForClass[classe].wait(&mutex);
5.  }
6.  currentWaiting[classe]--;   // à la libération (ou si total=n) on libère un autre de sa classe (s'il y en a encore) sinon 1 de l'autre classe
7.  if (currentWaiting[classe] > 0) {
8.      waitForClass[classe].wakeOne();
9.  } else {
10.     waitForClass[1-classe].wakeOne();
11. }
n.  mutex->unlock()
}
```

(voir explications plus loin)

Question 2 : Explications

- les N-1 premier thread entrant vont
  - incrémenter le nombre de waitings sur leur classe (L2)
  - se bloquer sur la waitCondition de leur classe (L4)

  car la condition (L3) ne sera satisfaite (il n'y a pas encore eu N threads de tous type).
- Le Ne thread passe tout droit à la condition (L3)
  Il incrémente (L2) et décrémente (L6) le nombre en attente sur sa classe
  - s'il y en a encore en attente sur sa classe (L7), il en libère un (L8)
  - sinon (L5), il libère un de l'autre classe (L10)

Les threads libérés opèrent de la même manière depuis (*)

Ceci garantit que tous les threads libérés seront d'abord de la classe du Ne, puis de l'autre classe.

Note: mutex->Unlock() n'est fait qu'à la fin, afin que tous se libèrent avant qu'il n'y ait un nouvel arrivant
(on pourrait aussi ajouter une condition isLiberation pour empêcher d'autres entrées pendant la libération)

Question 3: 20 points

Un ingénieur n'ayant pas suivi le cours PCO a été mandaté pour écrire le code d'un buffer de nourriture. Ce tampon multiple doit contenir des éléments de nourriture, des producteurs doivent pouvoir y placer de la nourriture et des consommateurs en récupérer. Une contrainte existe toutefois sur les consommateurs qui ne sont en fait intéressés que par un seul type de nourriture. Ils devront donc attendre que le prochain élément soit du type qui les intéresse.

Après une semaine de dur labeur, l'ingénieur a proposé le code suivant :

```
typedef enum {Pasta = 0, Rice = 1, Tomato = 2, NBFOODTYPES = 3} FoodType;

typedef struct
{
    FoodType type;
} Food;

class FoodBuffer
{
protected:
    Food *array;
    int size;
    int writePointer, readPointer, nbTot;
public:
    FoodBuffer(int size) : size(size), writePointer(0),readPointer(0), nbTot(0)
    {
        array = new Food[size];
    }

    void put(Food food)
    {
        while (nbTot == size) ;
        array[writePointer] = food;
        writePointer = (writePointer + 1) % size;
        nbTot ++;
    }

    Food get(FoodType type)
    {
        Food result;
        while ((nbTot == 0) || (array[readPointer].type != type)) ;
        result = array[readPointer];
        readPointer = (readPointer + 1) % size;
        nbTot --;
        return result;
    }
};
```

Votre oeil d'expert en programmation concurrente devrait légèrement se plisser en voyant ce code, et c'est avec raison que vous proposez à votre chef de le modifier afin de le rendre plus sûr et plus efficace. En utilisant des sémaphores Qt, modifiez cette classe de manière à en faire un code exploitable correctement et efficacement par une application multi-threadée.

Rappel sur les sémaphores Qt :

— `QSemaphore::QSemaphore(n)` : correspond à initialiser le sémaphore à `n` (`n` doit être ≥ 0) ;
— `QSemaphore::acquire()` : correspond à `P(sémaphore)` ;
— `QSemaphore::release()` : correspond à `V(sémaphore)`.

Voir 1.9

MINDER VALENTIN

Question 3.

```
class FoodBufferConcurrent : public FoodBuffer {
protected:
    QSemaphore * mutex;
    QSemaphore * waitForNotEmpty;
    QSemaphore * waitForNotFull;

public:
    FoodBufferConcurrent (int size) ...   // liste d'initialisation comme avant
    {
        nbWaitForNotEmpty = nbWaitForNotFull = 0;
        mutex = new QSemaphore(1); // ouvert !
        waitForNotEmpty = new QSemaphore(0); // idem pour l'autre
    }
    ~FoodBuffer () { delete mutex; delete waitForNotEmpty; delete waitForNotFull; }

virtual void put (Food food) {
    mutex -> acquire ();
    if (nbTot == size) { nbWaitNotFull ++;
        mutex -> release ();
        waitForNotFull -> acquire (); // réception de mutex
    }
    array [writePtr] = food;
    writePtr = (writePtr + 1) % size;
    nbTot ++;
    if (nbWaitForNotEmpty != 0) waitForNotEmpty.release (); // libération d'un client en attente...
    mutex -> release ();
}

virtual Food get (FoodType type) {
    mutex -> acquire (); nbWaitForNotEmpty ++;
    while ((nbTot == 0) || array [readPtr].type != type) {
        mutex -> release ();
        waitForNotEmpty -> acquire ();
        mutex -> acquire ();
    }
    nbWaitForNotEmpty --;
    result = array [readPtr];
    readPtr = (readPtr + 1) %
    nbTot --;
    if (nbWaitForNotFull != 0) {
        nbWaitForNotFull --;
        waitForNotFull -> release (); // transmission de mutex
    } else {
        mutex -> release ()
    }
}
```

section critique identique

section critique identique

[il faudrait réveiller le bon consommateur]

[si je n'ai pas fait de transmission (je voulais pas écrire la condition sur foodtype 2x)]

Voir au verso ?

⚠ attention la version proposée au verso risque de se bloquer si personne ne vient demander ce qui se trouve à la position reader (même si dans les suivantes, il y a ce qu'il faut pour satisfaire ces clients actuels...)
→ une solution serait de parcourir toutes les cases pour essayer de satisfaire un client (mais il pourrait ensuite y avoir des cases "vides" au milieu
→ ~~changer la logique complète~~ (tableau de cases circulaire))

- De plus, le client libéré n'est peut-être pas intéressé par la food produite, mais un autre client en attente, ici
→ libération / reblocage des clients successifs (si le client libéré ne prend pas, alors il libère un autre...)

p. ex., dans get

```
if (nbTot == 0)                                   // bloqué car vide
    mutex->unlock
    waitForNotEmpty.acquire();
if(                                               // si pas la bonne → on bloque
while (true) {                                    // bloqué car pas la bonne nourriture
    mutex->release
    waitForNotEmpty.acquire()
    mutex->acquire()
    if(bonne food){                               // si c'est la bonne nourriture, on sort
        break;
    } else { // on re-bloque et débloque un autre du prochain tour de while(true)
        waitForNotEmpty.release();
    }
}
```