HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Prof. Yann Thoma

Programmation Concurrente (PCO)
Semestre printemps 2015-2016
Contrôle continu 2
06.06.2016

Prénom:

Nom:

— 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	12
2	20	20
3	20	15
Total:	60	47

Note: 4,9

Question 1: 20 points

Nous désirons modéliser l'accès à un tunnel double. Les véhicules sont représentés par des threads qui devront faire appel à deux fonctions : `carAccess()` et `carLeave()`. La première pour l'accès au tunnel, et la deuxième lorsqu'il quitte le tunnel. Les points suivants doivent être assurés : no vehicule par tube

- Les véhicules sont de deux types différents : type 1 (grand véhicule), et type 2 (petit véhicule)
- Le tunnel est en fait un tunnel double, composé de 2 tubes. Le tube 1 permet de faire passer tous les véhicules, et le tube 2 seulement les petits véhicules
- Les grands véhicules sont prioritaires sur les petits véhicules
- Si un petit véhicule ne peut accéder à aucun tube à son arrivée, alors au final il empruntera de toute façon le tube 2
- La fonction `carAccess()` prend en paramètre le type du véhicule et retourne le numéro du tube emprunté par le véhicule
- La fonction `carLeave()` prend en paramètre le tube par lequel le véhicule est passé
- L'attente est modélisée par une suspension du thread

Ecrivez le code de la classe `DoubleTunnel`, en exploitant des sémaphores.

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)`.

```
class DoubleTunnel
{
public:
    DoubleTunnel();
    virtual ~DoubleTunnel();
    int carAccess(int vType);
    void carLeave(int way);
```

privé :

```
QSemaphore *tubes; mutex
int *nbreAttente, max, *current;

};

DoubleTunnel(): max(10), mutex(0) {
    mutex.release();
    tubes = new QSemaphore[2];
    nbreAttente = new int[2]; current = new int[2];
}
```

Question 2: 20 points

Nous désirons réaliser la modélisation d'un arrêt de bus. Les threads `Personne` devront, lorsqu'ils arrivent à l'arrêt, attendre le bus, sauf s'il y a plus de 30 personnes qui attendent déjà. Dans ce cas, elles continueront leur chemin. L'attente du bus reviendra à suspendre le thread. Les threads `Bus` devront, lorsqu'ils arrivent, embarquer les personnes présentes, mais au maximum 10. L'embarquement reviendra simplement à réveiller les threads personnes concernés.

Les threads `Personne` appeleront la méthode `peopleArrives()` alors que les bus appeleront la méthode `busArrives()`. La méthode `peopleArrives()` retourne un booléen qui doit être à vrai lorsque la personne peut attendre le bus et à faux si ce n'est pas le cas.

En utilisant uniquement des variables condition, écrivez la classe `BusStop` pour répondre à cette question.

Rappel sur les variables conditions Qt :

- `cond.wait(&mutex)` : Mise en attente sur la condition ;
- `cond.wakeOne()` : Signale la condition ;
- L'utilisation de `cond.wakeAll()` est autorisé.

```
#include <QMutex>
#include <QWaitCondition>

class BusStop {
public:
  BusStop();
  ~BusStop();

  bool peopleArrives();
  void busArrives();
};
```

Le code suivant illustre un exemple d'utilisation :

```
BusStop *b;

// exemple de tâche
void Personne::run() {
    if (b->peopleArrives()) {
      // On est dans le bus
    }
    else {
        // On n'est pas dans le bus
    }
    ...
}

// exemple de tâche
void Bus::run() {
  while (1) {
      b->busArrives();
        ...
      }
}

// programme principal
int main (int argc, char *argv[]) {
    b = new BusStop();
    // création de threads
    ...
}
```

```
class BusStop {

private:
    @UnixCondition cond;
    @Mutex mutex;
    int attente, max;
public:
  BusStop() {
    attente = 0;
    max = 30;
  }
  bool peopleArrives()
  {  mutex.lock();
     if(attente == max)
     {  mutex.unlock();
        return false;
     }
     attente++;
     cond.wait(&mutex);
     mutex.unlock;
     return true;
}
void busArrive() {
    mutex.lock();
    if(attente < no){attente = 0; cond.wakeAll();}
    else{ for(int i=0; i<no; i++) cond.wakeOne();
          attente -= no;
        }
    mutex.unlock();
}
```

Question 3: 20 points

Nous sommes intéressés à réaliser un tampon multiple un peu particulier, où il existe des producteurs de haute priorité et d'autres de basse priorité. Si plusieurs producteurs veulent déposer un élément dans le tampon, alors ceux de haute priorité doivent passer avant les autres.

Ecrivez le code de la classe `PriorityBuffer`, en vous basant sur les variables conditions. La fonction `put` prend notamment en paramètre la priorité du producteur, et le constructeur prend la taille du tampon en paramètres.

Rappel sur les variables conditions Qt :

— `cond.wait(&mutex)` : Mise en attente sur la condition ;
— `cond.wakeOne()` : Signale la condition ;
— L'utilisation de `cond.wakeAll()` est autorisé.

```
template<class ITEM>
class PriorityBuffer
{
public:
    typedef enum {HIGH = 0, LOW = 1} PriorityType;

public:
    PriorityBuffer(int size);
    void put(ITEM item, PriorityType priority);
    ITEM get();
```

```
private:
    QMutex mutex;
    QWaitCondition * condProd, condCons;
    int writeP, readP, size; current, * attente;
    ITEM *data;
};

PriorityBuffer(int size) : size(size) {
    writeP = readP = current = 0;
    condProd = new QWaitCondition[2];
    data = new ITEM[size];
    attente = new int[2];
}
void put(ITEM item, PriorityType priority)
{
    mutex.lock();
    while(current >= size){
        condProd[priority].wait(&mutex);
    }
    attente[priority]++;
    data[writeP] = item; writeP = (writeP+1) % size;
    current++; condCons.wakeOne();
    mutex.unlock();
}
```

```
ITEM get()
{
    ITEM result;
    mutex.lock();
while if (current==0)
        condCons.wait(&mutex);
    result = data[readP];
    readP = (readP+1) % BUFFER_SIZE;
    current--;
    if (attente[HIGH]>0){
        condProd[HIGH].wakeOne();
        attente[HIGH]--;
    }else{
        condProd[LOW].wakeOne();
        attente[LOW]--;
    }
    mutex.unlock();
    return result;
}
```

```
        if (nbreAttente[way-1] > 0)
        {
            nbreAttente[way-1]--;
            tubes[way-1].release();
        }
    }
    mutex.release();
```

```
int carAccess(int vtype) {
    int num = 1;
    mutex.acquire();
    while((current[vtype-1] >= max and vtype == 1) or
                (current[vtype-1] >= max and
                 current[0] >= max))
    {
        nbreAttente[vtype-1]++;
        mutex.release();
        tubes[vtype-1].acquire();
    }
    mutex.acquire();
    current[vtype-1]++;
    num = current[0] < max ? 1 : 2;
    mutex.release;
    return num;
}
void carLeave(int way){
    mutex.acquire();
    current[way-1]--;
    if (way == 1) {
        if (nbreAttente[way-1] > 0)
            tubes[way-1].release();
            nbreAttente[way-1]--;
        } else if (nbreAttente[way] > 0){
            nbreAttente[way]--;
            tube[way].release();
        }
    } else {
```