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

Prof. Yann Thoma

Programmation Concurrente (PCO)
Semestre printemps 2017-2018
Contrôle continu 1
17.04.2018

Prénom: Joel

Nom: Solai

— 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	10	5
2	15	12
3	10	8
4	10	10
Total:	45	35

Note: 8/7 5,4

Rappel sur les classes 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);`
— `QMutex::lock()` : verouille un mutex;
— `QMutex::unlock()` : déverouille un mutex;
— `QThread::start()` : lance un thread;
— `QThread::wait()` : attend qu'un thread se termine.

Attention. Il est interdit d'utiliser `tryLock()` et `tryAcquire()`, ainsi que `acquire(n)` et `release(n)` avec $n > 1$.

Joel Schär

Question 1: 10 points

Partant du constat que nous ne possédons que des sémaphores capables d'exécuter un seul `acquire()` et un seul `release()` à la fois, nous aimerions créer une classe `SuperSemaphore` offrant une méthode du type acquire prenant en paramètre le nombre de décrémentations à faire, `multiAcquire(unsigned int n)`, et une méthode `multiRelease(unsigned int n)` prenant en paramètre le nombre d'incrémentations à faire.

A vous de proposer une implémentation. Celle-ci doit évidemment se faire sans utiliser les méthodes des sémaphores Qt avec un `n` plus grand que 1. Dans ce cas précis nous ne sommes pas intéressés aux performances de votre code.

Pour anticiper votre question : oui, vous êtes en train d'implémenter une version sûre des méthodes des sémaphores Qt `acquire(n)` et `release(n)`.

```
class SuperSemaphore
{

public:

    SuperSemaphore(unsigned int initValue = 0);
    void multiAcquire(unsigned int n = 1);
    void multiRelease(unsigned int n = 1);
};
```

Question 2: 15 points

C'est le printemps, et les crapauds et grenouilles souffrent des traversées de routes qui ont tendance à les décimer. Heureusement, des amoureux de la nature ont creusé des tunnels leur permettant de traverser sans risque.

Nous sommes intéressés à modéliser ce tunnel à batraciens permettant de laisser passer un certain volume d'animal. Dans la population d'intérêt nous disposons de grenouilles et de crapauds, qui ont des volumes de respectivement 100 et 1'000 cm^3. Le tunnel ne tolère que 50'000 cm^3 de batraciens (l'entassement ne les dérange visiblement pas). Les règles suivantes doivent être appliquées :

1. Le volume des batraciens présents dans le tunnel ne doit pas excéder la capacité maximale du tunnel.
2. Les grenouilles ont réellement envie d'emprunter le tunnel. Elles se mettent donc en attente si le tunnel est déjà plein.
3. Les crapauds ont horreur d'attendre. S'il n'y a pas de place pour eux dans le tunnel, ils repartent.

La classe `Tunnel` doit offrir des fonctions appelées par les threads grenouilles (frog) et crapauds (toad). La fonction `frogIn()` est potentiellement bloquante, et la fonction `toadIn()` retourne un booléen qui vaut **`true`** si le crapaud peut entrer dans le tunnel, et **`false`** sinon.

A vous d'implémenter cette classe, en vous aidant de sémaphores.

```
class Tunnel
{
public:
    Tunnel();
    void frogIn();  // called by a frog to go in (potentially blocking)
    void frogOut(); // called by a frog when leaving the tunnel
    bool toadIn();  // called by a toad when wanting to go in the tunnel
    void toadOut(); // called by a toad when leaving the tunnel
};
```

Joel Schär

```
void frogIn(){
    mutex.acquire()
    if(volInTunnel + FROG_VOL <= TUN_VOL)
        volInTunnel += FROG_VOL;
        mutex.release();
    } else {
        nbWaiting++;
        mutex.release();
        waiting.acquire();
        volInTunnel += FROG_VOL;
    }
}
```

(pas besoin d'acquire, car transmis)

Joli !

```
void frogOut(){
    mutex.acquire()
    volInTunnel -= FROG_VOL;
    if(nbWaiting == 0){
        mutex.release();
    } else {
        nbWaiting--;
        waiting.release();
    }
}
```

(mutex transmis => pas de release)

```
bool toadIn(){
    mutex.acquire();
    if(volInTunnel + TOAD.VOL <= TUN_VOL){
        volInTunnel += TOAD_VOL;
        mutex.release();
        return true;
    }
    else{
        mutex.release();
        return false;
    }
}
```

```
void toadOut(){
    mutex.acquire()
    volInTunnel -= TOAD_VOL;
    if(nbWaiting == 0)
        mutex.release();
    else{
        nbWaiting--;
        waiting.release();
    }
}
```

la sortie d'un toad peut laisser entrer plusieurs grenouilles

← peut-être plusieurs

Question 3: 10 points

Pour cette question, nous avons 2 catégories de threads : des threads hydrogènes (H) et des threads oxygènes (O). Ces threads s'unissent pour se transformer en eau (H_2O), autrement dit, dès qu'un des threads manquants à former un thread H_2O arrive, celui-ci détruit 2 threads d'hydrogène et transforme le thread oxygène en eau.

Les deux méthodes proposées par la classe `H20` seront appelées respectivement par les threads `Oxygen` et `Hydrogen`. Il peut y avoir un nombre quelconque de threads de chaque type, la classe pouvant donc supporter un nombre indéfini d'appel à ses deux fonctions.

La version proposée ci-dessous l'a été par un ingénieur ne connaissant rien à la programmation concurrente. Modifiez ce code en utilisant les sémaphores afin de le rendre sûr, correct et efficace.

```
#include <QThread>

class H20
{
protected:
    int nbOxygen;
    int nbHydrogen;

public:

    H20() : nbOxygen(0), nbHydrogen(0)
    {

    }

    void arrivesOxygen()
    {
        nbOxygen ++;
        while (nbHydrogen < 2) ;
        nbOxygen --;
        // That's a miracle, I'm now water !!
    }

    void arrivesHydrogen()
    {
        nbHydrogen ++;
        while ((nbHydrogen < 2) || (nbOxygen < 1)) ;
        nbHydrogen --;
        // Let's kill the thread
        QThread::currentThread()->exit(0);
    }

};
```

arrivesOxygen() {
fifo.acquire

Joel Schär

```
class H2O {
protected:
    int nbOxygen;
    int nbHydrogen;
    QSemaphore mutex;
    QSemaphore waitForHydrogen;
    QSemaphore waitForOxygen;
public:
    H2O: nbOxygen(0), nbHydrogen(0), mutex(1), waitForHydrogen(0),
         waitForOxygen(0){ }

    void arrivesOxygen(){
        mutex.acquire();
        nbOxygen++;
        waitingForOxygen.release();
        while(nbHydroge<2){   mutex.release;
            waitingToHydroge.acquire();
        ?   mutex.acquire();
        nbOxygen--;
        mutex.release();

    }
    void arrivesHydrogen(){
        mutex.acquire();
        nbHydrogen++;
        waitingForHydrogen.release();
        while(nbOxygen<1){
            waitForOxygen.acquire();   ← bloqué à mutex...
        ?
        nbHydrogen--;
        mutex.release;

}
```

Question 4: 10 points

Soit la classe `MyClass` suivante :

```
class MyClass
{
protected:
    int value;
    bool alert;


public:

    MyClass() : value(0), alert(false) {}

    void function1(OtherClass *obj)
    {
        value = obj->getValue();
        value = value + 1;
        if (value > 1000)
            alert = true;
    }




    int getVal()
    {
        if (alert)
            return -1;
        else
            return value;
    }



    void compute()
    {
        if (value > 1000) {
            value -= 100;
        }
        else {
            value ++;
            if (value > 1000)
                alert = true;
        }
    }

};
```

Réécrivez [1] la classe en la rendant *thread-safe*, c'est-à-dire que ses méthodes publiques soient appelables depuis plusieurs threads en faisant que la sémantique des différentes méthodes soit respectée, tout en la laissant la plus efficace possible en ce qui concerne le temps de traitement.

Attention, la fonction `OtherClass::getValue()` est très gourmande en termes de temps d'exécution. Vous pouvez aussi considérer qu'elle est thread-safe et qu'elle peut donc être appelée par plusieurs threads en parallèle.

1. A droite du code existant, par exemple.

↓ suite

Joel Schär

```
void compute(){
  mutex.acquire();
  if (value > 1000) {
    value -= 100;
  } else {
    value++;
    if (value > 1000)
      alert = true;
  }
  mutex.release();
}
```