ITR TD4

Le TD suivant à pour but de proceder à une encapsulation des taches posix pour securiser leurs utilisation. On utilise donc une architecture de classe permettant d'automatiser au maximum leurs fonctionnement et de mettre à profit les constructeurs/destructeurs pour automatiser les creation/destructions de mutex.

Classe Thread

On programme une classe Thread. On utilise le constructeur pour mettre en place la politique d'ordonnancement Posix:

```
Thread::Thread(int schedPolicy) : m_schedPolicy(schedPolicy)
{
    pthread_t m_tid;
    pthread_attr_t m_attr;
    pthread_attr_init(&m_attr);
    pthread_attr_setschedpolicy(&m_attr, schedPolicy);
    pthread_attr_setinheritsched(&m_attr, PTHREAD_EXPLICIT_SC
}

```

Cette classe met en place une methode virtuelle pure `run()` qui va etre overridé par les sous-classes de threads spécifiques:

```
protected:
    virtual void run() = 0; // fonction virtuelle pure. C'est
```

Neammoins, il nous faut une interface commune. On implemente donc une methode call_run qui permet depuis l'objet Thread d'appeler la fonction run:

```
void* Thread::call_run(void* thread)
{
    Thread* ptr_thread= (Thread*)thread;
    ptr_thread->run();
}
```

Pour mettre en place notre incrementeur, on developpe une classe Incr qui derive de Thread `class Incr : public Thread` qui elle possede une fonction run réelle:

```
void Incr::run()
{
    if(m_pCounter->getMutexUse() == true)
    {
        for(int i=0; i <  m_pCounter->get_nLoops(); i++)
        {
            m_pCounter->incrementSafe();
        }
    }
    else
    {
        for(int i=0; i < m_pCounter->get_nLoops(); i++)
        {
            m_pCounter->incrementUnsafe();
        }
    }
}
```

On ajoute à la classe Thread un ensemble de fonction de parametrage:

- Un join avec Timeout qui vient surcharger le join habituel :

```
    void Thread::join(double timeout_ms)
{
    struct timespec abstime;
    clock_gettime(CLOCK_REALTIME, &abstime);
    long secs = timeout_ms / 1000;
    abstime.tv_sec += secs;
    abstime.tv_nsec += (timeout_ms - secs *1000) * 1000000;
    if(abstime.tv_nsec >= 1000000000)
    {
        abstime.tv_sec += 1;
        abstime.tv_nsec %= 1000000000; //TODO utiliser classe
    }
    pthread_timedjoin_np(m_tid, NULL, &abstime);
}
```

- Une option de parametrage de la taille de la pile:

```
void Thread::setStackSize(size_t stackSize)
{
    pthread_attr_setstacksize(&m_attr, stackSize);
    //printf("Thread stack size successfully set to %li bytes
}
```

- Une fonction d'endormissement du thread:

```
void Thread::sleep(double delay_ms)
{
    const double mille = 1000;
    struct timespec tim;
    tim.tv_sec = delay_ms / mille;
    tim.tv_nsec = (delay_ms - (delay_ms / mille) * mille) * m
    nanosleep(&tim, NULL);
}
```

On teste finalement ces classes sur notre programme précedant:

```
#include "Incr.h"
#include "Thread.h"
#include "Mutex.h"
#include "Lock.h"
#include <vector>
#include <iostream>
#include <stdio.h>

```

```
using namespace std;

int main(int argc, char* argv[])
{
    int nLoops = 0;
    int nTask=0;
    if(argc > 2)
    {
        sscanf(argv[1], "%d", &nLoops);
        sscanf(argv[2], "%d", &nTask);

        Incr::Counter counter(nLoops, true);

        int schedPolicy;
        schedPolicy = SCHED_RR;
        vector<Incr*> myVect;

        for(int i=0; i<nTask; i++)
        {
            Incr* ptrIncr = new Incr(&counter, schedPolicy);
            myVect.push_back(ptrIncr);
        }

        for(int i=0; i < nTask; i++)
        {
            cout << "main(): creating thread, " << i << endl;
            myVect[i]->start(42);
        }

        for(int i=0; i < nTask; i++)
        {
            myVect[i]->join();
        }

        for(int i=0; i < nTask; i++)
        {
            delete myVect[i];
        }


        myVect.clear();
        cout << "Le compteur vaut: " << counter.getValue() <<
        return 0;
    }
    return -1;
}
```

[Corentin, il ne manque pas la fonction call_run ici? Le thread tourne-t-il vraiment?]

Classes Mutex et Lock

De meme, nous allons maintenant créer une classe mutex pour en faciliter la manipulation. Le constructeur de cette classe automatise la création d'un mutex posix:

```
Mutex::Mutex(bool isInversionSafe)
{
    pthread_mutex_t mid;
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    //TODO iniatiliser et destroy attr
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE)
    if(isInversionSafe == true)
    {
        pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INH
    }
    pthread_mutex_init(&mid, &attr);
    pthread_mutexattr_destroy(&attr);
}
```

On donne l'option de se proteger de l'inversion de priorité par heritage de priorité.

Le destructeur automatise la destruction de l'objet Posix mutex:

```
Mutex::~Mutex()
{
    pthread_mutex_destroy(&mid);
}
```

Le Lock est une classe qui possede un objet de la classe mutex. Son principal interet est de liberer le mutex dans son destructeur. Ainsi, en fin de pile dans une fonction, quand le lock est detruit, son mutex est automatiquement libéré ce qui evite des erreurs qui pourraient bloquer l'ensemble des taches necessitant le mutex.

Cette classe ne possède donc que trois méthode:

```
Lock::Lock(Mutex* mutex): m(mutex)
{
    m->lock();
}

Lock::Lock(Mutex* mutex, double timeout_ms): m(mutex)
{
    if(m->lock(timeout_ms) == false)
        throw std::runtime_error("There was a runtime error")
}

Lock::~Lock()
{
    m->unlock();
}
```

Pour l'utiliser, il suffit de modifier legerement la classe Incr, et de lui donner un objet mutex en argument.

Dès lors on peut ecrire une fonction d'incrementation protégée par mutex;

```
double Incr::Counter::incrementSafe()
{
    try
    {
        Lock lock(&mutex);
        value += 1;
    }
    catch(std::exception& e)
    {
        std::cerr << "Error:" << e.what() << std::endl;
    }
    return value;
}
```

On voit ici l'interet de la classe lock. Son utilisation est etrememement aisée etant donnée qu'on a pas besoin de le detruire manuellement.

Classe Condition

On ecrit une classe Condition derivant de la classe Mutex. Cette classe implemente le système d'attente et de notification. Les methodes principale de cette classe sont

l'attente de notification, l'attente avec timeout, la notification, et le broadcast de notification

```
void Condition::wait()
{
    pthread_cond_wait(&cid, &mid);
}

bool Condition::wait(double timeout_ms)
{
    struct timespec abstime;
    clock_gettime(CLOCK_REALTIME, &abstime);
    long secs = timeout_ms / 1000;
    abstime.tv_sec += secs;
    abstime.tv_nsec += (timeout_ms - secs *1000) * 1000000;
    if(abstime.tv_nsec >= 1000000000)
    {
        abstime.tv_sec += 1;
        abstime.tv_nsec %= 1000000000; //TODO utiliser classe
    }
    if (pthread_cond_timedwait(&cid, &mid, &abstime) == ETIME
    {
        return false;//TODO faire un throw plutôt que return
    }
    return true;
}

void Condition::notify()
{
    pthread_cond_signal(&cid);
}

void Condition::notifyAll()
{
    pthread_cond_broadcast(&cid);
}
```

De la meme manière que pour Lock, on utilise le destructeur pour automatiquement detruire l'objet posix:

```
Condition::~Condition()
{
    pthread_cond_destroy(&cid);
}
```

On utilise cette classe pour proteger le champ Started de Thread. On modifie ainsi la methode start pour tenir compte de cette condition. On empeche donc de relancer une tache non terminée:

```
bool Thread::start(int priority)
{
    Lock lock(&condition);
    if(started)
    {
        return false;
    }
    started = true;
    sched_param schedParam;
    schedParam.sched_priority = priority;
    pthread_attr_setschedparam(&m_attr, &schedParam);
    pthread_create(&m_tid, &m_attr, call_run, this);
    return true;
}
```

Classe Semaphore

Le semaphore correspond à la metaphore de la "boite à jetons". Il implemente les methodes suivantes:

- give() ou on lui ajoute un jeton

```
void Semaphore::give()
{
    Lock lock(&condition);
    if(counter < maxCount)
    {
        counter += 1;
        condition.notifyAll();
    }
    else
    {
        while (counter >= maxCount)
        {
            condition.wait(); //wait libère le mutex en étant
        }
        counter += 1;
        condition.notifyAll();
    }
}
```

Cette fonction locke l'acces à la condition du semaphore, incremente puis notifie les autres taches qui peuvent attendre le jeton. Dans le cas ou le compteur est plein, l'appel est bloquant. Lock est detruit à la fin de la fonction, ce qui libère la condition.

- take() ou on lui retire un jeton

On l'implemente avec et sans timeout.

```
void Semaphore::take()
{
    Lock lock(&condition);
    if(counter > 0)
    {
        counter -= 1;
        condition.notifyAll();
    }
    else
    {
        bkdTasks += 1;
        while (counter == 0)
        {
            condition.wait(); //wait libère le mutex en étant
        }
        bkdTasks -= 1;
        counter -= 1;
        condition.notifyAll();
    }
}
```

De la meme manière que pour give, on locke la condition puis accède au compteur. Il est à noter que si le compteur de jetons du sémaphore est à zéro, l'appel de take() est bloquant. On compte egalement (si il y en a) le nombre de taches bloquée.

- flush() qui libère l'ensemble des jetons

```


void Semaphore::flush()
{
    Lock lock(&condition);
    counter += bkdTasks;
    condition.notifyAll();
}
```

On locke la condition, ajoute un nombre de jeton correspondant aux taches bloquée, puis libère.

Classe Fifo multitâches

La figure 5 spécifie l'interface d'une classe template Fifo. L'appel à pop() doit être bloquant si la fifo est vide ; l'appel bloquant doit comprendre une version avec timeout.

Programmez la classe Fifo en utilisant le conteneur C++ std::queue et testez-la en y accédant de manière concurrente par de multiples tâches productrices et consommatrices. Pour cela, utilisez une fifo de nombres entiers, faites produire par chaque tâche productrice une série d'entiers de 0 à n et mettez un place un mécanisme pour vérifier que tous les entiers produits par les tâches productrices ont bien été reçus par les tâches consommatrices.