TD 5 : Conditions, Pool de thread

Objectifs pédagogiques :

- **condition variable**
- **wait/notify**
- **pool de thread**

Introduction

Dans ce TD, nous allons réaliser en C++ une classe gérant un pool de thread. Plutôt que de créer un thread pour chaque tâche à traiter, le pool en initialise un certain nombre initialement (souvent adapté aux ressources CPU) et ils traitent des tâches en continu. Forcément tous ces threads collaborent, il va nous falloir des `<condition_variable>` pour traiter les notifications entre thread.

L'objectif de l'exercice est de bien comprendre comment réaliser les synchronisations utiles ; la lib standard du C++ ne propose pas actuellement cet utilitaire.

Pour être sûr de ne pas entrer en conflit avec d'autres applications, nous utiliserons le namespace `pr` pour notre implémentation.

1.1 Queue<T>.

1.1.1 Mise en place : buffer circulaire

Un buffer circulaire est une structure de donnée efficace pour partager des informations sur un support de taille limitée. Le buffer alloue un tableau (ou un vector) de T * de taille $ALLOCSZ$. La structure de données est FIFO, on va appeler la classe `Queue`.

Il offre deux principales opérations : `T * pop ()` (extrait le prochain élément du buffer) et `void push(T * t)` (insère un élément dans le buffer). Il dispose d'un indice *begin* désignant respectivement la prochaine case à lire et d'une taille sz donnant le remplissage actuel.

On propose de partir de la réalisation suivante conçue pour stocker des pointeurs. Les pointeurs sont censés être valides ; c'est donc au client qui fait push (le *producteur*) de faire un `new` pour obtenir le pointeur à insérer. Le client qui fait pop (le *consommateur*) doit donc symétriquement `delete` l'entrée quand il a terminé de s'en servir [1].

Queue.h

```
1  #ifndef SRC_QUEUE_H_
2  #define SRC_QUEUE_H_
3
4  #include <string> //size_t
5
6  template <typename T>
7  class Queue {
8        T ** tab;
9        const size_t allocsize;
10       size_t begin;
11       size_t sz;
12 public :
13       Queue (size_t maxsize) :allocsize(maxsize),begin(0),sz(0) {
14             tab = new T* [maxsize];
15             memset(tab, 0, maxsize * sizeof(T*));
16       }
```

[1] On aurait pu plutôt utiliser des `unique_ptr<T>`, qui correspondent à la sémantique voulue (passage de responsabilité d'un bloc mémoire), mais ce n'est pas l'objectif pédagogique de la séance donc on ne l'a pas fait ici.

```
17      size_t size() const {
18            return sz;
19      }
20      T* pop () {
21            T* ret = tab[begin];
22            tab[begin] = nullptr;
23            sz--;
24            begin = (begin+1) % allocsize;
25            return ret;
26      }
27      void push (T* elt) {
28            tab[(begin + sz)%allocsize] = elt;
29            sz++;
30      }
31      ~Queue() {
32            for (size_t i = 0; i < sz ; i++) {
33                  size_t ind = (begin + i) % allocsize;
34                  delete tab[ind];
35            }
36            delete[] tab;
37      }
38 };
39
40 #endif /* SRC_QUEUE_H_ */
```

Question 1. Expliquez le fonctionnement de la classe et de son destructeur. Donnez le code des méthodes privées `bool full() const` et `bool empty() const` rendant vrai si respectivement la queue est pleine ou vide.

Il faut commenter et décrypter le paragraphe sur les pointeurs et qui détient la mémoire.

Parler du destructeur = si la mémoire est dans la queue, et qu'on détruit la queue, c'est à elle de détruire les entrées.

Les memset 0 et remise à zéro sur pop ne sont pas strictement nécessaires, car le destructeur est bien écrit. Avec ces mise à zéro on pourrait just delete toutes les entrées.

Les déref de nullptr sont des fautes franches plus faciles à isoler et comprendre que des fautes sur des vieilles addresses qui ont été valides et qui trainent encore vaguement accessibles. Sous debugger ce sera donc appréciable que la mémoire soit à zéro, et ça ne coûte rien.

On peut suggérer que des unique_ptr seraient appropriés pour ce genre de comportement (passage de responsabilité de bloc mémoire) mais ce n'est pas le sujet principal du TD. donc on ne le fait pas ici.

Algo, pas grand chose à dire, l'indice begin = prochain indice d'une case en principe non vide où lire la valeur.

On incrémente modulo la taille d'allocation.

La case où écrire la prochaine valeur se calcule avec le remplissage actuel et un modulo allocsz.

on peut dessiner un cas à 3 cases et faire tourner un peu jusqu'à avoir begin < end (des cases à cheval entre fin et début du buffer).

On fait tourner sans faute pour l'instant, les push précèdent toujours un pop.

empty et full dans le corrigé suivant. Ils sont triviaux mais augmentent la lisibilité des tests de synchro dans le code par la suite.

Queue.h

```
1       bool empty() const {
2             return sz == 0;
3       }
4       bool full() const {
5             return sz == allocsize;
6       }
```

La queue actuellement n'est pas protégée contre les débordements de sous ou sur capacité.

Question 2. Modifiez le comportement pour que `push` rende un booléen : *false* et pas d'effet si la queue est pleine et *true* si l'insertion est réussie. Modifiez aussi `pop` pour qu'il rende un *nullptr* si la queue est vide (au lieu de la corrompre). On ajoute donc au contrat de la classe qu'il est interdit d'insérer des *nullptr*.

Donc on introduit une API pour un comportement "normal" en contexte non multi thread.
Naturellement ce comportement n'est pas bloquant.

Queue.h

```
1       T* pop() {
2             /// *******
3             if (empty()) {
4                   return nullptr;
5             }
6             /// *********
7             auto ret = tab[begin];
8             tab[begin] = nullptr;
9             sz--;
10            begin = (begin + 1) % allocsize;
11            return ret;
12      }
13      bool push(T* elt) { /// signature
14            if (full()) { /// test
15                  return false;
16            }
17            tab[(begin + sz) % allocsize] = elt;
18            sz++;
19            return true; /// return
20      }
```

donc on teste avec nos deux méthodes full et empty.

1.1.2 Une Classe MT-safe de synchronisation

La queue actuellement n'est pas protégée contre les accès multi-thread.

Question 3. Ajoutez un *mutex* et les synchronisations utiles pour protéger la classe contre les accès concurrents.

On a déjà vu le principe et la forme du pattern qui consiste à ajouter un `mutable mutex` à la classe, et à ajouter des `unique_lock` aux méthodes.

Il ne nous faut pas nécessairement de mutex sur l'API privée de la classe. Cela évite même d'introduire un recursive mutex, pour traiter le fait qu'une méthode de la classe par exemple en invoque une autre privée.

On utilise des unique_lock dans chaque méthode de préférence au lock/unlock à la main (RAII).

Ici la classe in extenso, mais on a juste ajouté : un attribut mutex et les unique lock au début des trois méthodes size/pop/push.

Queue.h

```
1 #ifndef SRC_QUEUE_H_
```

```
#define SRC_QUEUE_H_

#include <cstdlib>
#include <mutex>


template <typename T>
class Queue {
      T ** tab;
      const size_t allocsize;
      size_t begin;
      size_t sz;
      mutable std::mutex m;

      // fonctions private, sans protection mutex
      bool empty() const {
            return sz == 0;
      }
      bool full() const {
            return sz == allocsize;
      }
public:
      Queue(size_t size) :allocsize(size), begin(0), sz(0) {
            tab = new T*[size];
            memset(tab, 0, size * sizeof(T*));
      }
      size_t size() const {
            std::unique_lock<std::mutex> lg(m);
            return sz;
      }
      T* pop() {
            std::unique_lock<std::mutex> lg(m);
            if (empty()) {
                  return nullptr;
            }
            auto ret = tab[begin];
            tab[begin] = nullptr;
            sz--;
            begin = (begin + 1) % allocsize;
            return ret;
      }
      bool push(T* elt) {
            std::unique_lock<std::mutex> lg(m);
            if (full()) {
                  return false;
            }
            tab[(begin + sz) % allocsize] = elt;
            sz++;
            return true;
      }
      ~Queue() {
            // ?? lock a priori inutile, ne pas detruire si on travaille encore avec
            for (size_t i = 0; i < sz; i++) {
                  auto ind = (begin + i) % allocsize;
                  delete tab[ind];
            }
            delete[] tab;
      }
};
```

```
61
62
63
64 #endif /* SRC_QUEUE_H_ */
```

La queue actuellement ne constitue pas un mécanisme de *synchronisation*. Un thread consommateur qui attend une donnée pourrait devoir tourner en attente active sur *pop*. Au contraire, on souhaite introduire un comportement bloquant qui vient se substituer à celui proposé à la question 2.

Question 4. Ajoutez une condition pour bloquer tout thread qui tenterait un *pop* sur une queue vide ou un *push* sur une queue pleine. Symétriquement débloquer les threads éventuellement bloqués si l'on *push* sur queue vide ou que l'on *pop* sur queue pleine.

Donc une seule condition à ce stade. On se substitue au mécanisme non bloquant, i.e. push rend toujours vrai, pop rend toujours un élément.

Attributs :

Queue.h

```
1 class Queue {
2       T ** tab;
3       const size_t allocsize;
4       size_t begin;
5       size_t sz;
6       mutable std::mutex m;
7       std::condition_variable cv;
```

push et pop :

Queue.h

```
1       T* pop() {
2             std::unique_lock<std::mutex> lck(m);
3             while (empty()) {
4                   cv.wait(lck);
5             }
6             if (full()) {
7                   cv.notify_all();
8             }
9             auto ret = tab[begin];
10            tab[begin] = nullptr;
11            sz--;
12            begin = (begin + 1) % allocsize;
13            return ret;
14      }
15      bool push(T* elt) {
16            std::unique_lock<std::mutex> lg(m);
17            while (full()) {
18                  cv.wait(lg);
19            }
20            if (empty()) {
21                  cv.notify_all();
22            }
23            tab[(begin + sz) % allocsize] = elt;
24            sz++;
25            return true;
26      }
```

Question 5. Malgré le fait qu'il puisse y avoir des Threads bloqués en wait sur la condition, d'autres threads peuvent quand même acquérir le lock. Expliquez pourquoi.

> wait fait de façon atomique un unlock avant de dormir.
> Au réveil il reprend le lock atomiquement.
> Donc on dort sur wait, mais SANS le lock à la main.

1.1.3 Discussion, variantes

Question 6. Quel est l'intérêt d'utiliser deux conditions différentes pour séparément traiter les producteurs et les consommateurs bloqués ? Expliquez comment écrire une version avec deux conditions distinctes.

> Alors l'intérêt en principe on ne notifie que des gens intéressés, i.e. pas les producteurs si on push, pas les consommateurs si on pop.
> MAIS, pas clair ici que l'on puisse avoir au même instant des consommateurs et des producteurs simultanément bloqués. Les deux conditions sont réellement exclusives l'une de l'autre. Donc pas d'intérêt pratique.
> On fait quand même l'exercice demandé.
> Même avec plusieurs conditions dans la classe, on veut a priori toutes les contrôler avec le même mutex, ça ne pose pas de problème et c'est nécessaire de protéger la condition bool sur laquelle le wait porte.
> Il faut passer un unique_lock, on bascule donc là dessus si on a fait lock "à la main" plus haut.

Queue.h

```
class Queue {
      T ** tab;
      const size_t allocsize;
      size_t begin;
      size_t sz;
      mutable std::mutex m;
      std::condition_variable cv_plein;
      std::condition_variable cv_vide;
```

push et pop :

Queue.h

```
      T* pop() {
            std::unique_lock<std::mutex> lck(m);
            while (empty()) {
                  cv_vide.wait(lck);
            }
            if (full()) {
                  cv_plein.notify_all();
            }
            auto ret = tab[begin];
            tab[begin] = nullptr;
            sz--;
            begin = (begin + 1) % allocsize;
            return ret;
      }
      bool push(T* elt) {
            std::unique_lock<std::mutex> lg(m);
            while (full()) {
                  cv_plein.wait(lg);
```

```
            }
            if (empty()) {
                  cv_vide.notify_all();
            }
            tab[(begin + sz) % allocsize] = elt;
            sz++;
            return true;
      }
```

Question 7. On propose de se limiter à `notify_one` quand on **change** l'état de la queue de vide à non-vide, ou de plein à non-plein. Quel serait l'effet de cette modification sur la version à une seule condition, et sur celle avec deux conditions ?

Donc notify_one ne réveille qu'un seul thread bloqué.

Par rapport à notify_all, Si tous les threads qui attendent en face sont identiques/symétriques/peuvent traiter la notification, c'est ok en général, Ca évite que tout le monde se réveille alors qu'un seul va pouvoir bosser et les autres vont retourner wait.

Par contre, si la nature des threads en attente est différente, le plus souvent notify one n'est pas une bonne idée.

De ce point de vue all/one :

- Sur la version à une seule condition, on doit immédiatement se méfier : sur la condition techniquement on pourrait avoir et des consommateurs et des producteurs, mais on ne sait pas qui on va réveiller avec un notify_one.
- Ici, sur la version à deux conditions donc, ça parait ok, on a soit un paquet de consommateurs bloqués sur empty, soit un paquet de producteurs bloqués sur full, mais jamais de mélange.

Par contre si on adopte cette stratégie notify_one, il est essentiel de notifier à chaque pop/push, pas seulement quand l'état change.

Sinon on pourrait par exemple push trois éléments d'affilée mais ne réveiller qu'un seul consommateur, ce n'est pas ce qu'on souhaite ici.

Question 8. On propose pour être plus efficace de faire le wait du `pop` dans un test `if (empty()) cv.wait(m) ;`. Que penser de cette solution non standard ?

On fait un `if (cond) cv.wait(m) ;` c'est bad.

Mauvaise idée. Déjà la question le dit c'est non standard, il doit y avoir une raison...

Donc un scenario bien foireux :

C0 pop sur vide (bloque), P0 push (notify C0), C1 pop (rien), C0 finit son pop c'est vide !

Morale : ne pas trop réfléchir, suivre les schémas de synchro bien tracés.

Toujours faire un while ou la version lambda de wait.

Il faut un beau while, c'est le cas général, le cas notify all qui foire s'appelle "spurious wake", il est traité simplement par un *while*.

1.1.4 Terminaison

Question 9. La fin d'un programme utilisant une Queue pose actuellement un problème : que faire des threads bloqués dans *pop* sur la condition "file vide" ?

Ecrivez une fonction `void setBlocking(bool isBlocking)` dans la Queue, qui a pour effet de changer son comportement :

- Le `pop` redevient non bloquant (comme en question 2), et rend la valeur `nullptr` si la queue est vide.
- Tout thread bloqué en attente doit être réveillé et doit prendre en compte la nouvelle sémantique.

Du coup, si la valeur de retour du *pop* est un *nullptr*, le client sait qu'il doit s'arrêter.

NB: par symétrie on peut aussi rendre le push non bloquant, mais ce n'est pas strictement nécessaire.

Donc il nous faut un nouveau booléen pour isblocking dans la classe.
Le wait dans pop change : on attends !empty || !isBlocking
Si au réveil on a "empty and !isBlocking", on rend nullptr.
Le setblocking se passe sous mutex normal, et notifie la condition en broadcast.

Queue.h

```
class Queue {
      T ** tab;
      const size_t allocsize;
      size_t begin;
      size_t sz;
      bool isBlocking;
      mutable std::mutex m;
      std::condition_variable cv;
```

le pop :

Queue.h

```
      void setBlocking(bool isBlocking) {
            std::unique_lock<std::mutex> lck(m);
            this->isBlocking = isBlocking;
            cv.notify_all();
      }
      T* pop() {
            std::unique_lock<std::mutex> lck(m);
            while (empty() && isBlocking) {
                  cv.wait(lck);
            }
            if (empty()) {
                  return nullptr;
            }
            if (full()) {
                  cv.notify_all();
            }
            auto ret = tab[begin];
            tab[begin] = nullptr;
            sz--;
            begin = (begin + 1) % allocsize;
            return ret;
      }
```

le push (non demandé)

Queue.h

```
      bool push(T* elt) {
            std::unique_lock<std::mutex> lg(m);
            while (full() && isBlocking) {
                  cv.wait(lg);
            }
```

```
6             if (full()) {
7                   return false;
8             }
9             if (empty()) {
10                  cv.notify_all();
11            }
12            tab[(begin + sz) % allocsize] = elt;
13            sz++;
14            return true;
15      }
```

C'est assez fréquent quand on a des condition_variable de devoir prévoir une façon de terminer propre, i.e. débloquer les threads qui sont bloqués dessus.

C'est nécessaire de "join" tous les threads lancés avant la fin du programme, on ne pourra pas le faire s'ils sont bloqués sur une cond.

Donc on enrichit la condition booléenne utilisée pour le wait, et on notifyAll pour permettre aux threads bloqués de s'échapper.

1.2 Pool de Thread

On va maintenant réaliser une classe Pool gérant un pool de thread. Cette classe gère un ensemble de threads, ce qui évite de payer l'instanciation d'un thread pour chaque traitement à réaliser en parallèle, et permet de dimensionner le degré de concurrence.

On commence par se donner une façon de définir une tâche qu'on pourra soumettre au pool.

1.2.1 Job abstrait et concret

On introduit une classe abstraite pure (ou interface) `Job` pour encapsuler un traitement à spoumettre au Pool.

Job.h

```
1  #pragma once
2
3  class Job {
4  public:
5        virtual void run () = 0;
6        virtual ~Job() {};
7  };
```

La classe `Job` est munie d'une méthode abstraite `virtual void run() =0;`.

Le marqueur "=0" indique que la méthode n'a pas d'implantation (méthode abstraite). Le mot clé "virtual" indique que cette opération peut être redéfinie dans une classe fille [2].

Elle doit aussi porter un destructeur `virtual  Job(){}` vide, comme c'est une classe de base pour de l'héritage.

Question 10. Compléter le code d'un job concret `SleepJob`, qui possède un entier (argument) représentant les arguments passés au Job, et un pointeur d'entier (result) où il doit écrire son résultat. Le traitement lui-même est simulé par un sleep ici.

```
1  class SleepJob : public Job {
2        int calcul (int v) {
```

[2] En Java, une méthode peut être redéfinie sauf si elle est déclarée `final`. En C++, une méthode ne peut **pas** être redéfinie sauf si elle est déclarée `virtual`.

```
            std::cout << "Computing for arg =" << v << std::endl;
            // traiter un gros calcul
            this_thread::sleep_for(1s);
            int ret = v % 255;
            std::cout << "Obtained for arg =" << arg << " result " << ret << std::endl;
            return ret;
      }
      int arg;
      int * ret;
public :
      SleepJob(int arg, int * ret) : arg(arg), ret(ret) {}
      void run () {
            // TODO : compléter
      }
      ~SleepJob(){}
};
```

Donc on commente le fait que Job doit porter ses arguments et l'endroit où poser son retour, vu la signature sans rien de run().

Un job plus complet pourrait aussi notifier/modifier un compteur de job faits, pour que quelqu'un qui attende la fin soit au courant. On reviendra sur cette idée avec les <future>.

main.cpp

```
      void run () {
            * ret = calcul(arg);
      }
```

Question 11. Ecrire un main qui crée un job concret et l'exécute.

```
int ret; // résultat du calcul
Job * pjob = new SleepJob(42,&ret);
pjob->run();
cout << ret << endl;
```

1.2.2 Une classe Pool

La classe Pool possède en attributs une `Queue<Job>` et un `vector<thread>`. Elle offre comme API :

- Construction avec un entier pour la taille d'allocation de la Queue
- `void start (int NBTHREAD)` : instancie NBTHREAD threads, et les met en boucle sur la queue à traiter des jobs
- `void submit (Job * job)` : ajoute un job à la queue (peut être bloquant dans notre implémentation).

Pool.h

```
class Pool {
      Queue<Job> queue;
      std::vector<std::thread> threads;
public:
      Pool(int qsize) ;
```

```
6       void start (int nbthread);
7       void submit (Job * job) ;
8       void stop() ;
9       ~Pool() ;
10  };
```

Question 12. Ecrire une fonction `void poolWorker(Queue<Job> & queue)` qui sera le corps des threads gérés par le Pool. Les threads tournent en boucle sur le comportement suivant :

- extraire un Job de la queue
- le traiter

Version de base :

```
1  // fonction passee a ctor de thread
2  void poolWorker(Queue<Job> & queue) {
3  while (true) {
4  Job * j = queue.pop();
5
6  j->run();
7  // ne pas oublier de delete !
8  delete j;
9  }
10 }
```

Question 13. A l'aide de la primitive `setBlocking` de `Queue`, ajouter une opération `void stop()` au Pool, qui doit libérer et join tous les threads crées par start. On attendra qu'ils aient fini leur Job en cours le cas échéant.

Donc on ajoute au corps de la fonction worker :

Pool.h

```
1       // fonction passee a ctor de thread
2       void poolWorker(Queue<Job> * queue) {
3             while (true) {
4                   Job * j = queue->pop();
5                   // NB : ajout en fin de TD pour la terminaison propre
6                   if (j == nullptr) {
7                         // on est non bloquant = il faut sortir
8                         return;
9                   }
10                  j->run();
11                  delete j;
12            }
13      }
```

Et une méthode stop dans Pool:

Pool.h

```
1             queue.setBlocking(false);
2             for (auto & t : threads) {
3                   t.join();
4             }
5             threads.clear();
```

```
6       }
7       ~Pool() {
```

La classe Pool, in extenso, non demandé en TD.

Pool.h

```
1  /*
2   * Pool.h
3   *
4   * Created on: Oct 12, 2018
5   * Author: ythierry
6   */
7
8  #ifndef SRC_POOL_H_
9  #define SRC_POOL_H_
10
11 #include "QueueBlock.h"
12 #include "Job.h"
13 #include <vector>
14 #include <thread>
15
16 namespace pr {
17
18       // fonction passee a ctor de thread
19       void poolWorker(Queue<Job> * queue) {
20             while (true) {
21                   Job * j = queue->pop();
22                   // NB : ajout en fin de TD pour la terminaison propre
23                   if (j == nullptr) {
24                         // on est non bloquant = il faut sortir
25                         return;
26                   }
27                   j->run();
28                   delete j;
29             }
30       }
31
32 class Pool {
33       Queue<Job> queue;
34       std::vector<std::thread> threads;
35 public:
36       Pool(int qsize) : queue(qsize) {}
37       void start (int nbthread) {
38             threads.reserve(nbthread);
39             for (int i=0 ; i < nbthread ; i++) {
40                   threads.emplace_back(poolWorker, &queue);
41             }
42       }
43       // Ajout a la fin du TD, pour la terminaison
44       void stop() {
45             queue.setBlocking(false);
46             for (auto & t : threads) {
47                   t.join();
48             }
49             threads.clear();
50       }
```

```
51        ~Pool() {
52              stop();
53        }
54        void submit (Job * job) {
55              queue.push(job);
56        }
57  };
58
59  } /* namespace pr */
60
61  #endif /* SRC_POOL_H_ */
```

1.2.3 Progression du calcul

Un client utilisant le thread pool souhaite soumettre N jobs, puis attendre qu'ils soient tous traités pour arrêter le Pool (on connait N). On propose de coder une classe `Barrier` pour satisfaire ce besoin.

Question 14. Proposez une classe Barrier, munie en attribut d'un mutex, d'un compteur, d'un N attendu, et d'une condition variable. Elle propose l'api :

- constructeur prenant N en argument
- `void done()` incrémente le compteur, notifie la condition si N atteint
- `void waitFor()` attends sur la condition que le compteur atteigne N

Appelons ça une barrière.

Donc on ajoute un mutex, un condition var et un compteur partagé de jobs faits. A la construction on lui passe le N.

On a deux opérations sur barrier : waitFor() et done().

Waitfor attends (wait) que le compteur atteigne N

done incrémente et notifie si N atteint.

On passe un pointeur vers la barrière quand on crée les Job.

Queue.h

```
1  #ifndef __BARRIER_H__
2  #define __BARRIER_H__
3
4  #include <mutex>
5  #include <condition_variable>
6
7
8  class Barrier {
9        int counter;
10       const int MAX;
11       std::mutex m;
12       std::condition_variable cv;
13 public :
14       Barrier(int max) : counter(0), MAX(max) {}
15
16       void done() {
17             std::unique_lock<std::mutex> l(m);
18             counter++;
19             if (counter == MAX)
20                   cv.notify_all();
```

```
	}
	void waitFor() {
		std::unique_lock<std::mutex> l(m);
		while (counter != MAX) {
			cv.wait(l);
		}
	}
};

#endif
```

Question 15. Modifiez le Job concret pour qu'il notifie la barrière avec *done* quand il a fini. Ecrivez ensuite un main qui crée un pool, le démarre avec K threads, soumet N jobs, attend qu'ils soient finis, arrête le pool, affiche la valeur des résultats et se termine.

Queue.h

```
#include "Pool.h"
#include "Barrier.h"
#include "Job.h"
#include <iostream>
#include <thread>
#include <vector>

using namespace std;
using namespace pr;

class SleepJobBarrier : public Job {
	int calcul (int v) {
		std::cout << "Computing for arg =" << v << std::endl;
		// traiter un gros calcul
		this_thread::sleep_for(1s);
		int ret = v % 255;
		std::cout << "Obtained for arg =" << arg << " result " << ret << std::endl;
		return ret;
	}
	int arg;
	Barrier * barrier;
	int * ret;
public:
	SleepJobBarrier(int arg, Barrier * bar, int * ret) : arg(arg), barrier(bar), ret(
	    ret) {}
	void run() {
		*ret = calcul(arg);
		barrier->done();
	}
	~SleepJobBarrier() {}
};

int main() {
	const int NBTHREAD = 20;
	const int NBJOB = 60;

	Pool pool(15); // queue size
	vector<int> results(NBJOB);
	pool.start(NBTHREAD);
```

```
39        Barrier b(NBJOB);
40        for (int i = 0; i < NBJOB; i++) {
41              pool.submit(new SleepJobBarrier(i, &b, &results[i]));
42        }
43
44        b.waitFor();
45        pool.stop();
46
47        for (int &i : results) {
48              cout << i << ",";
49        }
50        cout << endl;
51
52        return 0;
53  }
```

TME 5 : Parallélisation d'une application

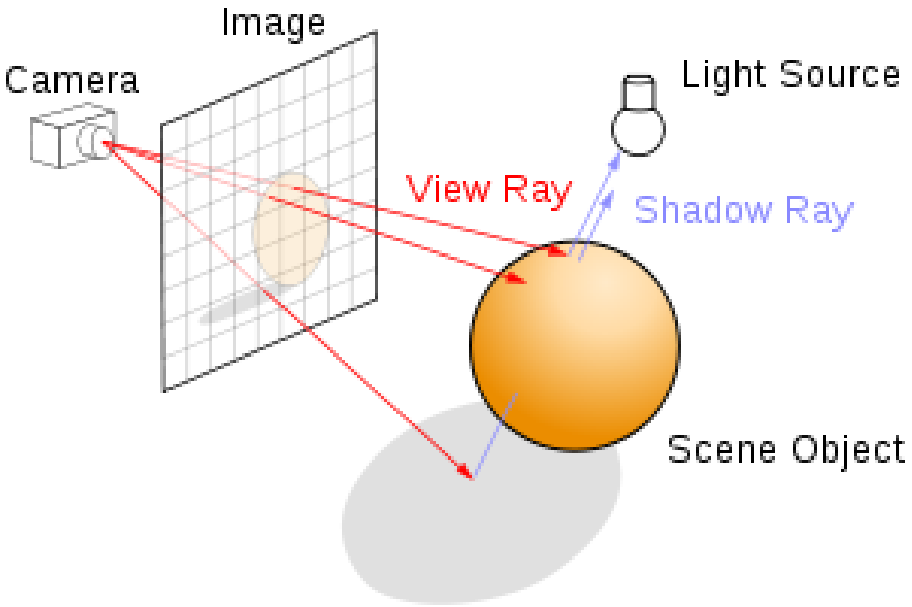
Objectifs pédagogiques :

- **condition, mutex, barrière**
- **pool de threads**
- **parallélisation d'une application**

1.1 Objectif

On vous fournit le code d'un Ray tracer très basique, qui sait dessiner des scenes représentant des Spheres colorées avec un éclairage.

Pour cela, le code calcule la couleur de chaque pixel de l'image à l'aide d'un rayon tiré de l'observateur (la caméra) vers les points de l'écran. On a pour cela actuellement une double boucle imbriquée qui calcule la couleur des pixels pour chaque position dans l'écran.



Votre mission (si vous l'acceptez) est de paralléliser ce code. A priori la tâche est assez facile : la couleur de chaque pixel peut tout à fait être calculée en parallèle. Le calcul de la couleur nécessite un accès en lecture seule sur l'état de la scene qui contient les sphères.

Pour réaliser ce travail, on propose de s'appuyer sur les classes Queue, Pool et Job définies dans le TD 4.

1.2 Mise en place

Question 1. Vérifiez votre email `@etu.upmc.fr` : vous devez avoir une invitation gitlab sur le dépot :`http://pscr-gitlab.lip6.fr`.

Activez votre compte en vous connectant sur `https://pscr-gitlab.lip6.fr/`, votre login = numéro d'étudiant.

Authentifiez-vous, puis faire un "fork" du projet : `https://pscr-gitlab.lip6.fr/ythierry/TME5` pour obtenir une copie du dépot de sources contenant le Ray Tracer à améliorer.

Question 2. Cloner ensuite votre fork du projet `https://pscr-gitlab.lip6.fr/XXXXXX/TME5` (où XXXX est votre numéro d'étudiant) dans votre espace de travail.

Sous eclipse,

- sur la page Gitlab de votre nouveau projet, sous le bouton "Clone or download", copier

l'adresse dans le presse-papier (Ctrl-C)

- Ouvrir la perspective Git (bouton coin en haut à droite)
- Dans "Window->Préférences->General->Network connections", sélectionner le provider "Manual" (au lieu de "native"), puis éditer HTTPS, pour utiliser "proxy" sur port "3128".
- A gauche, choisir "Clone a git Repo", si on a copié l'adresse plus haut il pre-remplit les champs
- compléter avec votre login/pass github, et cocher "use secure store".
- On voit maintenant le projet a gauche, ouvrir le projet et sélectionner le dossier "Working Tree", puis clic-droit "Import as Project"
- Rebasculer en perspective C/C++

En ligne de commande,

- positionner les proxy : `export https_proxy=https://proxy:3128`
- dans un répertoire vide, cloner le projet : `git clone https://pscr-gitlab.lip6.fr/XXXXX/TME5.git` (où XXX est votre numéro étudiant).

Question 3. Après le clone, lancer la configuration du projet `cd TME5 ; autoreconf -vfi ; ./configure ; make` dans un terminal. Sous Eclipse, faire "File->Refresh" après cette opération, puis "Project->Build Project" le projet doit être reconnu/fonctionner.

Question 4. Lancer le programme "src/tme5" et admirer l'image générée "toto.ppm" à la racine du projet.

Question 5. Créez la classe Queue avec son comportement final dans le TD : une seule condition, comportement bloquant par défaut, possibilité de basculer à non bloquant si on le souhaite.

NB: Une version de la classe (MT safe, mais comportement non bloquant) est fournie.

Question 6. Créez la classe abstraite (interface) Job et la classe Pool avec son comportement final dans le TD : construction avec la taille de la queue, start, stop.

> cf corrigé du TD

Question 7. Créez la classe Barrier, qui permet d'attendre la fin des jobs

> cf corrigé du TD

Question 8. Créez un Job concret qui contient essentiellement le corps de la boucle imbriquée sur les pixels.

> Ca nous donne ça plus ou moins :

Queue.h

```
#include "Vec3D.h"
#include "Pool.h"
#include "Barrier.h"
#include "Rayon.h"
#include "Scene.h"
#include <iostream>
#include <algorithm>
#include <fstream>
#include <limits>
```

```
#include <random>

using namespace std;
using namespace pr;




class DrawJobBarrier : public Job {
      int foo(int v) {
            // traiter un gros calcul
            this_thread::sleep_for(1s);
            return v % 255;
      }
      const Vec3D & screenPoint;
      Color & pixel;
      const Scene & scene;
      const vector<Vec3D> & lights;
      Barrier * barrier;
public:
      DrawJobBarrier(const Vec3D & screenPoint, Color & p, const Scene & s, const vector
          <Vec3D> & l,Barrier * b) : screenPoint(screenPoint),pixel(p),scene(s),lights(
          l),barrier(b) {}
      void run() {
            // le rayon a inspecter
            Rayon ray(scene.getCameraPos(), screenPoint);
            Color finalcolor;
            auto minz = std::numeric_limits<float>::max();
            for (const auto & obj : scene) {
                  // rend la distance de l'objet a la camera
                  auto zinter = obj.instersects(ray);
                  // si intersection plus proche ?
                  if (zinter < minz) {
                        minz = zinter;
                        // pixel prend la couleur de l'objet
                        finalcolor = obj.getColor();

                        auto camera = scene.getCameraPos();
                        // calcul du rayon et de sa normale a la sphere
                        // on prend le vecteur de la camera vers le point de l'ecran (
                            dest - origine)
                        // on le normalise a la longueur 1, on multiplie par la distance
                             à l'intersection
                        Vec3D rayInter = (ray.dest - ray.ori).normalize() * zinter;
                        // le point d'intersection
                        Vec3D intersection = rayInter + camera;
                        // la normale a la sphere au point d'intersection donne l'angle
                            pour la lumiere
                        Vec3D normal = obj.getNormale(intersection);
                        // le niveau d'eclairage total contribue par les lumieres 0
                            sombre 1 total lumiere
                        double dt = 0;
                        // modifier par l'eclairage la couleur
                        for (const auto & light : lights) {
                              // le vecteur de la lumiere au point d'intersection
                              Vec3D tolight = (light - intersection);
                              // si on est du bon cote de la sphere, i.e. le rayon n'
                                  intersecte pas avant de l'autre cote
                              if (obj.instersects(Rayon(light,intersection)) >= tolight.
```

```
                                length() - 0.05 ) { // epsilon 0.05 for double issues
62                                dt += tolight.normalize() & normal ; // l'angle (
                                      scalaire) donne la puissance de la lumiere
                                      reflechie
63                            }
64                        }
65                        // eclairage total
66                        finalcolor = finalcolor * dt + finalcolor * 0.2; // *0.2 =
                              lumiere speculaire ambiante
67                }
68            }
69            pixel = finalcolor;
70            barrier->done();
71      }
72      ~DrawJobBarrier() {}
73  };
74
75
76
77  // NB : en francais pour le cours, preferez coder en english toujours.
78  // pas d'accents pour eviter les soucis d'encodage
79
80  int main () {
81        // on pose une graine basee sur la date
82        default_random_engine re(std::chrono::system_clock::now().time_since_epoch().count
              ());
83        // definir la Scene
84        Scene scene (1000,1000);
85        const int NBSPHERES = 500;
86
87        std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();
88
89        // on remplit la scene de spheres colorees de taille position et couleur aleatoire
90        uniform_int_distribution<int> distrib(0, 200);
91        uniform_real_distribution<double> distribd(-200, 200);
92        for (int i = 0; i < NBSPHERES; i++) {
93              // position autour de l'angle de la camera
94              // rayon entre 3 et 33, couleur aleatoire
95              // distrib(re) rend un entier aleatoire extrait de re
96              scene.add(Sphere({50+distribd(re),50 + distribd(re),40 + distribd(re) },
                    double(distrib(re)%30) + 3.0, Color::random()));
97        }
98        // quelques spheres de plus pour ajouter du gout a la scene
99        scene.add(Sphere({50,50,40},15.0,Color::red));
100       scene.add(Sphere({100,20,50},55.0,Color::blue));
101
102       // les points de l'ecran
103       const Scene::screen_t & screen = scene.getScreenPoints();
104       // lumieres
105       vector<Vec3D> lights;
106       lights.reserve(3);
107       lights.emplace_back(Vec3D(50, 50, -20));
108       lights.emplace_back(Vec3D(50, 50, 120));
109       lights.emplace_back(Vec3D(200, 0, 120));
110
111       // Les couleurs des pixels
112       Color * pixels = new Color[scene.getWidth() * scene.getHeight()];
113
114
```

```
115
116       const int NBTHREAD = 8;
117       const int NBJOB = scene.getWidth() * scene.getHeight();
118
119       Pool pool(15); // queue size
120       vector<int> results(NBJOB);
121       pool.start(NBTHREAD);
122       Barrier b(NBJOB);
123       // pour chaque pixel, calculer sa couleur
124       for (int x =0 ; x < scene.getWidth() ; x++) {
125             for (int y = 0 ; y < scene.getHeight() ; y++) {
126                   // le point de l'ecran en coordonnees 3D
127
128                   pool.submit(new DrawJobBarrier(screen[y][x],pixels[y*scene.getHeight()
                          + x],scene,lights,&b));
129             }
130       }
131
132       b.waitFor();
133       pool.stop();
134
135       std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
136          std::cout << "Total time "
137                  << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).
                        count()
138                  << "ms.\n";
139
140       // ppm est un format ultra basique
141       ofstream img ("toto.ppm");
142       // P3 signifie : les pixels un par un en ascii
143       img << "P3" << endl; // ascii format, colors
144       // largeur hauteur
145       img << scene.getWidth() << "\n"<< scene.getHeight() << "\n" << "255" << endl;
146       // tous les pixels au format RGB
147       for (int y = 0 ; y < scene.getHeight() ; y++) {
148             for (int x =0 ; x < scene.getWidth() ; x++) {
149                   Color & pixel = pixels[x*scene.getHeight() + y];
150                   img << pixel << '\n';
151             }
152       }
153       // oui ca fait un gros fichier :D
154       img.close();
155
156       return 0;
157 }
```

Question 9. Assembler ces éléments pour paralléliser le code. On créera deux threads par CPU sur la machine environ.

> TODO

Question 10. Faire varier le grain du parallélisme, e.g. si un job consiste à calculer la couleur d'un seul pixel, modifier votre code pour calculer la couleur des pixels d'une ligne entière dans le job.

On doit avec un paramétrage adapté tourner quasiment N fois plus vite sur une machine avec N coeurs.

1.3 Rendu du travail

Question 11. Faites un push de vos modifications vers votre dépot.

Sous Eclipse, dans la perspective C/C++ normale :

- Window->Show view->Git Staging
- dans cette fenêtre on a trois parties : les fichiers modifiés mais non inclus dans le commit, les fichiers "indexés" c'est à dire inclus dans le prochain commit, la zone de saisie du message de commit.
- On double clic un fichier pour avoir une comparaison à l'original
- Clic droit "Add to index" si ça a l'air bien + ajouter un commentaire
- on finit avec un Commit (local) ou un Commit And Push (aussi repercuté sur Github)
- Si l'on n'a pas encore push, clic droit sur le projet, "Team->push to Upstream".

En ligne de commande :

- git add fichiers_modifies
- git commit -m 'message de commit'
- git push