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.

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*.

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.

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.

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.

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.

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 ?

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 ?

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.

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] 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`.

```
      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;
      int * ret;
public :
      SleepJob(int arg, int * ret) : arg(arg), ret(ret) {}
      void run () {
            // TODO : compléter
      }
      ~SleepJob(){}
};
```

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

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) ;
      void start (int nbthread);
      void submit (Job * job) ;
      void stop() ;
      ~Pool() ;
};
```

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

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.

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

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.

TME 5 : Parallélisation d'une application

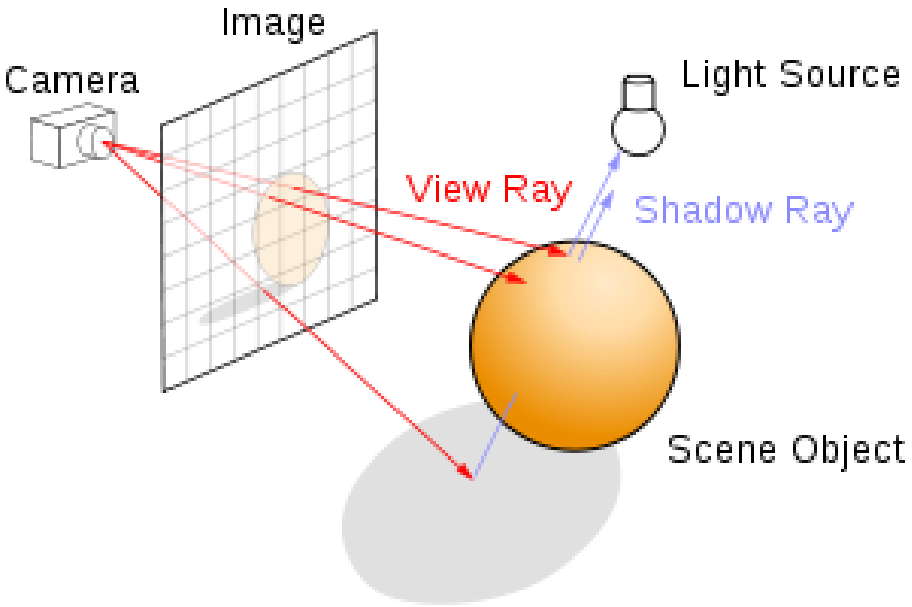
Objectifs pédagogiques :

- **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.

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

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

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

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