TD 4 : 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>.

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 de deux indices deb et fin désignant respectivement la prochaine case à lire et la prochaine case à remplir.

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.

Queue.h

```
#ifndef SRC_QUEUE_H_
#define SRC_QUEUE_H_

#include <string> //size_t

template <typename T>
class Queue {
      T ** tab;
      const size_t allocsize;
      size_t begin;
      size_t sz;
public :
      Queue (size_t maxsize) :allocsize(maxsize),begin(0),sz(0) {
            tab = new T* [maxsize];
            memset(tab, 0, maxsize * sizeof(T*));
      }
      size_t size() const {
            return sz;
      }
      T* pop () {
            auto ret = tab[begin];
```

```
            tab[begin] = nullptr;
            sz--;
            begin = (begin+1) % allocsize;
            return ret;
      }
      void push (T* elt) {
            tab[(begin + sz)%allocsize] = elt;
            sz++;
      }
      ~Queue() {
            for (size_t i = 0; i < sz ; i++) {
                  auto ind = (begin + i) % allocsize;
                  delete tab[ind];
            }
            delete[] tab;
      }
};

#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*.

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.

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 pour être plus efficace, en plus d'utiliser deux conditions distinctes, 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, et de faire le wait du `pop` dans un test `if (empty()) cv.wait(m) ;`. Que penser de cette solution non standard ?

1.2 Pool de Thread

On va maintenant réaliser une classe Pool gérant un pool de thread.

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

Question 8. Définir une classe `Job` munie d'une méthode abstraite `virtual void run() =0;`. Elle doit aussi porter un destructeur `virtual  Job(){}` vide comme c'est une classe de base pour de

l'héritage.

Question 9. Définir un job concret, qui possède un entier (argument) représentant les arguments passés au Job, et un pointeur d'entier (result) où il doit écrire son résultat. L'action de *run* consiste à afficher "début sur args =...", dormir un peu, écrire dans *res la valeur modulo 256, afficher "fini sur arg... res vaut ...".

La classe Pool possède en attributs une `Queue<Job>` et un `vector<thread>`. La classe Pool 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).

Question 10. Proposez une réalisation de cette classe

Question 11. La fin d'un programme utilisant une Queue pose actuellement un problème : que faire des threads éventuellement bloqués sur la condition ? Ecrivez une fonction `void setBlockingPop(bool isBlocking)` dans la Queue, qui a pour effet de changer le comportement de la queue. Le `pop` redevient non bloquant (comme en question 2), et rend la valeur `nullptr` si la queue est vide. Tout thread en attente doit être réveillé et prendre en compte la nouvelle sémantique.

Question 12. A l'aide de cette nouvelle primitive, 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.

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 réaliser une classe Barrier pour réaliser ce besoin.

Question 13. 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 14. Modifiez le Job concret pour qu'il notifie la barrière avec *done* quand il a fini, et le main pour qu'il attende tous les jobs qu'il crée, arrête le pool, puis affiche la valeur des résultats.

TME 4 : Parallélisation d'une application

Objectifs pédagogiques :

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

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 la scene 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. Créez un compte github, authentifiez-vous, puis suivre le lien pour obtenir une copie du dépot de sources contenant le Ray Tracer à améliorer.

`https://classroom.github.com/a/xXBIwV0Y`

Question 2. Cloner ce nouveau projet dans votre espace de travail.

Sous eclipse,

- sur la page Github 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", puis éditer HTTP et HTTPS et SOCKS, 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++

Question 3. 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.

Question 4. 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 5. Créez la classe Barrier, qui permet d'attendre la fin des jobs

Question 6. Créez un Job concret qui contient essentiellement le corps de la boucle imbriquée sur les pixels.

Question 7. Assembler ces éléments pour paralléliser le code. On créera deux threads par CPU sur la machine environ.

1.3 Rendu du travail

Question 8. 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".