TD 9 : Future, Promise, Task

Objectifs pédagogiques :

- **future, promise, async en C++**
- **parallélisme de tâches**

Introduction

Dans ce TD, nous allons étudier des briques participant à la mise en place d'un modèle de concurrence basé sur les tâches, partiellement décorrélé du modèle de parallélisme.

1 Future, Promise

L'objectif de cet exercice est de réaliser nos propres versions de `future` et `promise` simplifiées par rapport au standard (pas de gestion des exceptions) en appui sur les primitives de synchronisation classiques des thread (mutex, condition).

On propose d'implanter trois classes template : `shared_result`, `promise`, `future`.

1.1 Shared Result

La classe `shared_result` est le coeur de notre synchronisation ; c'est un conteneur qui *peut* héberger un objet de type T. Son API est la suivante :

- `T get ()` rend la valeur stockée, bloquant si elle n'est pas disponible.
- `void set(T &val)` positionne la valeur, débloque le client bloqué sur `get` le cas échéant.
- `bool is_set () const` (non bloquant) rend vrai si `set` a été invoqué sur cet objet.

Question 1. Proposez, en appui sur les primitives de synchronisation des threads C++11, une implantation pour cette classe.

1.2 Future

La classe `future` représente le canal de lecture pour un client qui attend un résultat. On construit un objet `future` en lui passant l'adresse d'un `shared_result`. Son API est la suivante :

- `T get ()` : bloquant si pas encore disponible
- `bool isAvailable() const` : vrai si le résultat est prêt

Question 2. Proposez une implantation pour cette classe.

1.3 Promise

La classe `promise` représente le canal en écriture vers la donnée partagée, dans lequel le thread serviteur va poser le résultat de son calcul. Quand on construit un objet `promise`, il crée une instance de `shared_result`. Son API est la suivante :

- `future<T> getFuture()` construit et rend un objet future, attaché au résultat.
- `void set_value(const T & r)` affecte une valeur au résultat

Question 3. Proposez une implantation pour cette classe.

1.4 Gestion mémoire

Question 4. A quel moment faut-il désallouer le `shared_result` qui est pointé par le couple promise/future ? En utilisant les pointeurs intelligents de `<memory>` proposez une réalisation de ce comportement.

Question 5. Si on suppose que le résultat est une classe qui alloue de la mémoire (e.g. une String), combien de copies sont payées dans l'implantation actuelle pour transmettre le résultat du thread serviteur au thread client ? Comment pourrait-on contourner ce problème ?

Question 6. Si on souhaite supporter le scenario où plusieurs clients attendent le résultat calculé par le serviteur, le standard propose `shared_future`. Expliquez les différences avec le `future` simple.

2 Pool de thread et Future

On revient sur la réalisation d'un pool de thread, comme au TD4.

Pool.h

```
#ifndef SRC_POOL_H_
#define SRC_POOL_H_

#include "Queue.h"
#include <vector>
#include <thread>

namespace prTD4 {

class Job {
public:
      virtual void run () = 0;
      virtual ~Job() {};
};

// fonction passee a ctor de thread
inline void poolWorker(Queue<Job> * queue) {
      while (true) {
            Job * j = queue->pop();

            // pour la terminaison propre
            if (j == nullptr) {
                  // on est non bloquant = il faut sortir
                  return;
            }

            j->run();
            delete j;
      }
}

class Pool {
      Queue<Job> queue;
      std::vector<std::thread> threads;
public:
      Pool(int qsize) : queue(qsize) {}
      void start (int nbthread) {
            threads.reserve(nbthread);
            for (int i=0 ; i < nbthread ; i++) {
                  threads.emplace_back(poolWorker, &queue);
            }
      }
```

```
      void stop() {
            // débloque les threads bloqués sur pop
            queue.setBlocking(false);
            // on attend qu'ils aient fini leur job actuel
            for (auto & t : threads) {
                  t.join();
            }
            threads.clear();
      }
      ~Pool() {
            stop();
      }
      void submit (Job * job) {
            queue.push(job);
      }
};

} /* namespace pr */

#endif /* SRC_POOL_H_ */
```

On veut réaliser une version `Pool<T>` auquel on puisse soumettre des `Job<T>` qui définit une seule opération `T run()` (donc au lieu de la signature `void run()` proposée). Pour éviter de tomber trop profondément dans les problèmes d'instantiation de template, tous les `Job<T>` soumis à un `Pool<T>` donné auront donc le même type de retour T.

La nouvelle signature de l'opération `void submitJob (Job *job)` est `future<T> submitJob(Job<T> *job)`.

Question 1. Modifier le Pool de thread et les définitions liées (Job, fonction exécutée par les thread...) pour obtenir ce nouveau comportement.

Question 2. Expliquer comment utiliser ce nouveau mécanisme dans un main.

3 Taches et Threads, Concurrence et Parallélisme

Question 1. Discutez des différences au niveau syntaxe (invocation, utilisation) et sémantique (créations de thread, parallélisme effectif) entre le système d'exécution asynchrone que l'on vient de construire et les comportements (en fonction de la policy `launch_deferred|launch_async`) de la fonction `async` proposée dans le standard.

Question 2. Comment dimensionner un pool de thread utilisé dans le contexte d'un serveur TCP ? Dans le contexte d'un ray tracer concurrent ? Que peut-on en déduire vis à vis de la seule fonction `async` proposée par le standard ?

Question 3. Essayez de proposer un mécanisme qui permettrait avec un nombre fixe de threads de traiter un nombre plus grand de connexions simultanées dans un serveur TCP.

4 TME 9

Dans cette séance :

- Finaliser le TME8, on souhaite atteindre la partie où l'on utilise Protobuf.
- On peut introduire les mécanismes vu dans le TD9 autour de notre pool de thread. Dans ce cas, utilisez les version `std::promise, std:::future` au lieu de celles implantées "à la main" dans le TD.
- Expérimentez avec async, par exemple, modifier le TME4 pour comparer l'utilisation du Pool

de thread à celle de "async". Faites varier le grain du parallélisme, essayez d'améliorer les performances en déployant vos connaissances en parallélisme.