SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 1 PSCR
Master 1 Informatique Nov 2023

UE 4I400

Année 2023-2024

2 heures – **Tout document papier autorisé**
Tout appareil de communication électronique interdit (téléphones. . .)
Clé USB en lecture seule autorisée.

Introduction

- Le barème est sur 24,5 points et est donné à titre indicatif.
- Dans le code C++ demandé vous prendrez le temps d'assurer que la compilation fonctionne.
- On vous fournit une archive contenant un projet eclipse CDT par exercice, qu'il faudra modifier. Décompresser cette archive dans votre home, de façon à avoir un dossier `~/exam/` et les sous dossiers `~/exam/exo1/src` ...
- Pour importer ces projets dans Eclipse, le plus facile :"File->Import->General->Existing Projects into Workspace",
- Si vous préférez utiliser un autre IDE ou la ligne de commande, on vous fournit dans chaque répertoire source un Makefile trivial.
- A la fin de la séance, fermez simplement votre session. On ira par script récupérer les fichiers dans ce fameux dossier `~/exam/`. Assurez vous donc de bien suivre ces instructions.

Le sujet est composé d'exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite. Pour la majorité des questions il s'agit de fournir un code compilable correct.

1 Indexation de fichiers

On considère le problème suivant: étant donné un ensemble de mots clés et un ensemble de fichiers, on souhaite construire un index, c'est à dire associer à chaque mot clé la liste des fichiers auquel il appartient.

1.1 unordered_map, unordered_set (4 points)

Question 1. A l'aide des classes de la lib standard, implanter ce comportement dans "exo1". On complétera le "main.cpp" fourni qui est conçu pour être très simple (variables globales précédent les fonctions qui s'en servent). On dispose d'un cadre dans "utils.hh" pour invoquer une fonction sur chaque mot rencontré dans un fichier.

NB: Même si on recommande `std::unordered_map` et `std::unordered_set`, et plus spécifiquement un `unordered_map<string,unordered_set<string>` associant des listes de noms de fichier à chaque mot clé. Cependant on acceptera les réponses moins efficaces utilisant e.g. `vector`.

main.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>

#include <unordered_map>
#include <unordered_set>


using namespace std;

#include "utils.hh"

// define data structures

unordered_set<string> keywords;
unordered_map<string,unordered_set<string>> index;

// we read a keyword : add to keywords
void onKeyWordEncountered(const std::string& word, const std::string& filename) {
   // std::cout << "saw " << word << " in " << filename << std::endl;
   keywords.insert(word);
}

// we read a word : if it is a keyword, add file to files containing keyword
void onWordEncounteredInFile(const std::string& word, const std::string& filename) {
      if (keywords.find(word) != keywords.end()) {
            // hit, this word is a keyword
            // NB : index[word] is an empty unordered_set<string> if word was not in the
                 map
            index[word].insert(filename);
      }
}

int main (int argc, const char **argv) {
      if (argc < 3) {
            std::cerr << "Invoke with keyword file as first argument followed by the
                files to index." << std::endl;
            std::cerr << "e.g. indexer data/keywords.txt data/*" << std::endl;
      } else {
            std::cout << "Reading keywords from " << argv[1] << std::endl;
            std::cout << "Indexing files " ;
            for (int i=2; i < argc ; i++) {
                  std::cout << argv[i];
            }
            std::cout << std::endl;
      }

      // parse arguments to main
      processFile(argv[1], onKeyWordEncountered);

      for (int i=2; i < argc ; i++) {
            processFile(argv[i], onWordEncounteredInFile);
      }


      for (const auto & keyword : keywords) {
            std::cout << "key:" << keyword << std::endl;
```

```
            std::cout << "Found in : ";
            const auto & files = index[keyword];
            for (const auto & file : files) {
                  std::cout << file << " ";
            }
            std::cout << std::endl;
      }

      return 0;
}
```

Barème :

- 20 % déclaration des structures de données
- 20 % fonction invoquée pour les keyword, les range dans la structure de donnée, e.g. insere avec un set vide dans la map.
- 30 % fonction invoquée pour les word, 10% teste si c'est un keyword, 20% ajoute dans la table (beaucoup de variantes possibles).
- 20 % a chage du résultat
- 10 % compile, tourne sans faute majeure.

Résultat attendu (exo1 à exo3):

```
> exo1 keywords poeme*
Reading keywords from keywords
Indexing files poeme1.txtpoeme2.txtpoeme3.txtpoeme4.txtpoeme5.txt
key:coeur
Found in : poeme2.txt
key:paysage
Found in : poeme1.txt
key:visage
Found in : poeme1.txt
key:Baudelaire
Found in : poeme5.txt poeme4.txt poeme3.txt poeme2.txt poeme1.txt
key:spleen
Found in :
```

1.2 Multi-threading (4 points)

Question 2. Modifier à présent le programme pour le rendre concurrent. On souhaite engendrer un thread pour chaque fichier argument. Bien entendu, il faut protéger les structures de données manipulées contre les accès concurrents. On recommande d'écraser le contenu du fichier "mainMT.cpp" avec le contenu de votre main actuel, puis d'ajouter les éléments de synchronisation utiles et la création des thread (un par fichier argument). La lecture des mots clés restera dans le thread principal. On prendra garde à sortir proprement. A ce stade on impose une solution avec un seul mutex.

mainMT.cpp

```
#include <iostream>
#include <fstream>
#include <string>

#include <unordered_map>
#include <unordered_set>
#include <thread>
```

```cpp
#include <mutex>
#include <vector>


using namespace std;

#include "utils.hh"

// define data structures

mutex m; // NEW

unordered_set<string> keywords;
unordered_map<string,unordered_set<string> > index;

// we read a keyword : add to keywords
void onKeyWordEncountered(const std::string& word, const std::string& filename) {
   // std::cout << "saw " << word << " in " << filename << std::endl;
   keywords.insert(word);
}

// we read a word : if it is a keyword, add file to files containing keyword
void onWordEncounteredInFile(const std::string& word, const std::string& filename) {
      if (keywords.find(word) != keywords.end()) {
            unique_lock<mutex> l(m); // NEW

            // hit, this word is a keyword
            // NB : index[word] is an empty unordered_set<string> if word was not in the
                 map
            index[word].insert(filename);
      }
}

int main (int argc, const char **argv) {
      if (argc < 3) {
            std::cerr << "Invoke with keyword file as first argument followed by the
                files to index." << std::endl;
            std::cerr << "e.g. indexer data/keywords.txt data/*" << std::endl;
      } else {
            std::cout << "Reading keywords from " << argv[1] << std::endl;
            std::cout << "Indexing files " ;
            for (int i=2; i < argc ; i++) {
                  std::cout << argv[i];
            }
            std::cout << std::endl;
      }
      // parse arguments to main
      processFile(argv[1], onKeyWordEncountered);


      vector<thread> threads; // NEW
      threads.reserve(argc-2);

      for (int i=2; i < argc ; i++) {
            threads.emplace_back(processFile,argv[i], onWordEncounteredInFile); // NEW
      }

      for (auto & t:threads) { // NEW
            t.join();
```

```
65        }
66
67        for (const auto & keyword : keywords) {
68              std::cout << "key:" << keyword << std::endl;
69              std::cout << "Found in : ";
70              const auto & files = index[keyword];
71              for (const auto & file : files) {
72                    std::cout << file << " ";
73              }
74              std::cout << std::endl;
75        }
76
77        return 0;
78  }
```

Barème en delta sur la classe précédente, dans le corrigé les lignes ajoutées sont annotée avec NEW:

- 10 % on déclare un mutex
- 40 % l'accès au map est protégé par mutex dans la deuxième fonction (traiter mot) quand on insère.
- 10 % si on ne verrouille que au moment d'insérer, pas pendant qu'on cherche si c'est un keyword
- 20 % on lance un thread par fichier
- 10 % on wait
- 10 % compile et tourne sans faute majeure

1.3 Synchronisation fine (4 points)

Question 3. (5 points) Modifier à présent le programme pour effectuer une synchronisation plus fine, dans laquelle on se donne un mutex pour chaque mot clé, de manière à pouvoir verrouiller l'ensemble des noms de fichiers associés à un mot clé donné, au lieu de devoir tout protéger ("big fat lock" comme la question précédente). De nouveau on peut commencer par écraser le contenu de "mainMTfin.cpp" avec votre version de la question précédente. Pour cette question, on recommande de revoir vos structures de donnée, on souhaite que la table qui associe l'ensemble des fichiers aux mots clés soit stable pendant que l'on itère les fichiers en concurrence. On recommande de

- définir un `unordered_map<string,int>` "keyIndex" associant à chaque mot clé un indice unique de 0 à (nombre de mots clé -1);
- définir un `vector<mutex>` "mutexes" avec (nombre de mots clé) entrées
- définir un `vector<unordered_set<string>` "files" tel que à l'indice i on trouve les fichiers contenant le mot clé d'indice i dans "keyIndex".
- Mettre à jour le code en conséquence (affichages, fonctions exécutées par les threads...)

. On fera attention à initialiser ces structures dans la phase séquentielle de parse des mots clés, et donc avant de lancer le premier thread.

mainMTfin.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  #include <unordered_map>
6  #include <unordered_set>
7  #include <thread>
```

```cpp
#include <mutex>
#include <vector>


using namespace std;

#include "utils.hh"

// define data structures
unordered_map<string,int> keyIndex;

vector<unordered_set<string>> index;

vector<mutex> mutexes;

// we read a keyword : add to keywords
void onKeyWordEncountered(const std::string& word, const std::string& filename) {
   // std::cout << "saw " << word << " in " << filename << std::endl;
   int id = keyIndex.size();
   keyIndex.insert(make_pair(word,id));
}

void onWordEncounteredInFile(const std::string& word, const std::string& filename) {
      auto it = keyIndex.find(word);
      if (it != keyIndex.end()) {
            int nkey = it->second;
            {
                  // lock for insertion
                  unique_lock<mutex> l(mutexes[nkey]);

                  // files associe, peut etre vide
                  index[nkey].insert(filename);
            }
      }
}

int main (int argc, const char **argv) {
      if (argc < 3) {
            std::cerr << "Invoke with keyword file as first argument followed by the
                files to index." << std::endl;
            std::cerr << "e.g. indexer data/keywords.txt data/*" << std::endl;
      } else {
            std::cout << "Reading keywords from " << argv[1] << std::endl;
            std::cout << "Indexing files " ;
            for (int i=2; i < argc ; i++) {
                  std::cout << argv[i];
            }
            std::cout << std::endl;
      }
      // parse arguments to main
      processFile(argv[1], onKeyWordEncountered);

      // initialize using number of keywords
      mutexes = vector<mutex>(keyIndex.size());
   index = vector<unordered_set<string>>(keyIndex.size());


      vector<thread> threads;
      threads.reserve(argc-2);
```

```
66
67        for (int i=2; i < argc ; i++) {
68              threads.emplace_back(processFile,argv[i], onWordEncounteredInFile);
69        }
70
71        for (auto & t:threads) {
72              t.join();
73        }
74
75        for (const auto & kwindex : keyIndex) {
76              std::cout << "key:" << kwindex.first << std::endl;
77              std::cout << "Found in : ";
78              int nkey = kwindex.second;
79              for (const auto & file : index[nkey]) {
80                    std::cout << file << " ";
81              }
82              std::cout << std::endl;
83        }
84
85        return 0;
86  }
```

Barème :

- 20 % déclaration des structures de données avec des index int
- 10 % un vector de mutex
- 10 % mise à jour de la collecte des kw dans la première méthode
- 10 % initialisation du vector de mutex et du vector<set<string» avec le nombre de kw
- 20 % lock fin dans la deuxième méthode, protège uniquement la partie critique
- 20 % mise à jour des a chages
- 10 % compile et tourne sans faute

1.4 Parallélisme contraint (4 points)

Question 4. Modifier le fichier fourni “mainContraint.cpp" pour que :

- Le programme engendre au total N threads qui vont exécuter la fonction “work”.
- A un instant donné, il ne doit pas y avoir plus de K threads actifs
- On prendra garde à sortir proprement

. NB : cet exercice est indépendant des questions précédentes et peut être traité séparément.

mainContraint.cpp

```
1   #include <thread>
2   #include <cstring>
3   #include <iostream>
4   #include <vector>
5
6   using namespace std;
7
8   int work() {
9         std::this_thread::sleep_for(std::chrono::seconds(1)); // Simulate work
10        return 42;
11  }
12
```

```cpp
int main (int argc, const char **argv) {
      if (argc < 3) {
            std::cerr << "Invoke N and K" << std::endl;
            exit(1);
      }
      int N=atoi(argv[1]);
      int K=atoi(argv[2]);

      // lancer N thread
      vector<thread> threads;
      threads.reserve(argc-2);

      int active = 0;
      int lastJoin=0;

      for (int i=0; i < N ; i++) {
            threads.emplace_back(work);
            active++;
            if (active >=K) {
                  // garantir au plus K thread actifs.
                  threads[lastJoin].join();
                  lastJoin++;
            }
      }

      // sortie propre
      for (int i=lastJoin; i < N ; i++) {
            auto & t=threads[i];
            t.join();
      }

      return 0;
}
```

Barème (TODO) en delta sur la classe précédente:

- 20 % le main crée N thread au total
- 10 % on join tous les threads crées
- 30 % on comptabilise le nombre de thread actifs et on essaie de s'en servir
- 30 % on limite e ectivement à K threads actifs simultanément
- 10 % compile et tourne sans faute (on ne donne pas si c'est juste N thread sans rien de plus)

1.5 Une classe de synchronisation (4 points)

Question 5. Ecrire une classe générique "`Result<T>`" munie de :

- un attribut "valeur" de type T
- `bool set(const T & val)` qui met à jour la valeur stockée si elle n'est pas encore positionnée et rend true, ou rend false sinon (et ne fait rien).
- `const T & get()` qui rend la valeur stockée, **en attendant qu'elle soit disponible**.

Cette classe sera utilisée dans un contexte concurrent, ou un thread fera le "set" et un ou plusieurs autres threads attendent d'accéder à la valeur calculée avec "get". Compléter le fichier "Result.cpp" pour définir la classe demandée et dans le main créer deux thread qui interagissent avec.

Result.cpp

```cpp
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <thread>

template <typename T>
class Result {
private:
   T valeur;
   bool isValueSet = false;
   std::mutex mtx;
   std::condition_variable cv;

public:
   bool set(const T& val) {
      std::lock_guard<std::mutex> lock(mtx);
      if (!isValueSet) {
         valeur = val;
         isValueSet = true;
         cv.notify_all(); // Notify all waiting threads
         return true;
      }
      return false;
   }

   const T& get() {
      std::unique_lock<std::mutex> lock(mtx);
      cv.wait(lock, [this]{ return isValueSet; }); // Wait until the value is set
      return valeur;
   }
};

void producerThread (Result<int>& result) {
   std::this_thread::sleep_for(std::chrono::seconds(1)); // Simulate work
   result.set(42);
}

void consumerThread (Result<int>& result) {
   std::cout << "Value received: " << result.get() << std::endl;
}

int main() {
   Result<int> result;

   // TODO : creer un thread qui invoque "producerThread" et un thread qui invoque "
       consumerThread"

   // Thread that sets the value
   std::thread setter(producerThread, std::ref(result));

   // Thread that gets the value
   std::thread getter(consumerThread, std::ref(result));

   setter.join();
   getter.join();

   return 0;
```

```
57 }
```

Barème (TODO)

- 20 % la classe stocke un mutex et une cond var
- 10 % un attribut pour déterminer si on a déjà a ecté
- 35 % set; sous mutex (10), distingue si set (10), notifie (15)
- 25 % get; sous mutex (10), wait correctement (15)
- 10 % compile et tourne sans faute

1.6 Question compréhension (4,5 point)

Question 6. Répondez dans "questions.txt" aux questions suivantes:

1. Quel est l'intérêt en pratique d'un pool de thread par rapport à l'instanciation d'un thread par tâche à traiter ?
2. Expliquez comment communiquer une donnée particulière entre deux threads. Cette stratégie est elle encore valable dans un contexte multi-processus ?
3. Expliquez quand et de quelle manière sont traités les signaux reçus en fonction du "masque" du processus.

1. L'instanciation coute cher et prend du temps et des ressources; la réutilisation d'un thread déjà existant est donc préférable quand elle est possible. C'est d'autant plus le cas si les taches à éxécuter sont relativement rapides. Le pool permet aussi de dimensionner les ressources (nombre de threads = niveau de parallelisme) séparément du niveau de concurence (granularité des taches).
2. Il faut simplement se synchroniser, e.g. avec des mutex et si besoin des condition variable. Les threads partagent naturellement leur vision de la mémoire. En multi processus, on est forcé de s'appuyer sur les IPC pour communiquer, e.g. file, pipe, shm, sem, socket...
3. A l'élection d'un processus, le système scanne le bitmask de signaux recus, si un signal est "recu" et non "masqué", on exécute le traitement associé (handler ou ignorer ou tuer le processus)

Barème (TODO)

1. 70 % meilleure utilisation des ressources, instancier un thread coute cher. 30 % on évoque un des autres points, e.g. controle du nombre de threads
2. 50 % atomic, mutex, condition variable + mémoire partagée; 50 % IPC en multi process
3. 30 % les signaux sont gérés à l'élection de la tache, 30 % les handlers, 40 % le masque