SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 1 PSCR
Master 1 Informatique Nov 2019

UE 4I400

Année 2019-2020

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

Introduction

- Le barème est sur 20 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", Pointer le dossier `/exam`, Eclipse doit voir 4 projets, tous les importer (sans cocher "copy 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.
- Vous n'avez pas un accès complet à internet, mais si vous configurez le proxy de votre navigateur (proxy, port 3128, tous les protocoles) vous aurez accès au site `https://cppreference.com`

Le sujet est composé de trois 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 Table de hash concurrente (5 points)

1.1 Multi-thread safe

On considère la table de hash fournie suivante.

HashMap.h

```
1  #pragma once
2  #include <forward_list>
3  #include <vector>
4
5  namespace pr {
6
7  template <typename K, typename V>
8  class HashMap {
9
```

```
public:
      class Entry {
      public :
            const K key;
            V value;
            Entry(const K &k,const V& v):key(k),value(v){}
      };
private :

      typedef std::vector<std::forward_list<Entry> > buckets_t;
      // stockage pour la table de buckets
      buckets_t buckets;
      // nombre total d'entrées dans la table
      size_t sz;

public:
      HashMap(size_t size): buckets(size),sz(0) {
            // le ctor buckets(size) => size cases, initialisées par défaut.
      }

      V* get(const K & key) {
            size_t h = std::hash<K>()(key);
            size_t target = h % buckets.size();
            for (Entry & ent : buckets[target]) {
                  if (ent.key==key) {
                        return & ent.value;
                  }
            }
            return nullptr;
      }

      bool put (const K & key, const V & value) {
            size_t h = std::hash<K>()(key);
            size_t target = h % buckets.size();
            for (Entry & ent : buckets[target]) {
                  if (ent.key==key) {
                        ent.value=value;
                        return true;
                  }
            }
            sz++;
            buckets[target].emplace_front(key,value);
            return false;
      }

      size_t size() const { return sz ; }
};

} /* namespace pr */
```

Question 1. (1,5 points) A l'aide d'un seul mutex, modifier cette classe afin de la rendre utilisable dans un contexte multi-threadé concurrent (rendre la classe *MultiThread-safe*).

HashMap.h

```
#ifndef SRC_HASHMAP_H_
#define SRC_HASHMAP_H_

#include <cstddef>
```

```cpp
#include <ostream>

#include <forward_list>
#include <vector>
#include <mutex>

namespace pr {

template <typename K, typename V>
class HashMap {
	mutable std::mutex m;
public:
	class Entry {
	public :
		const K key;
		V value;
		Entry(const K &k,const V& v):key(k),value(v){}
	};
private :

	typedef std::vector<std::forward_list<Entry> > buckets_t;
	// storage for buckets table
	buckets_t buckets;
	// total number of entries
	size_t sz;

public:
	HashMap(size_t size): buckets(size),sz(0) {
		// le ctor buckets(size) => size cases, initialisées par défaut.
	}

	V* get(const K & key) {
		std::unique_lock<std::mutex> l(m);
		size_t h = std::hash<K>()(key);
		size_t target = h % buckets.size();
		for (Entry & ent : buckets[target]) {
			if (ent.key==key) {
				return & ent.value;
			}
		}
		return nullptr;
	}

	bool put (const K & key, const V & value) {
		std::unique_lock<std::mutex> l(m);
		size_t h = std::hash<K>()(key);
		size_t target = h % buckets.size();
		for (Entry & ent : buckets[target]) {
			if (ent.key==key) {
				ent.value=value;
				return true;
			}
		}
		sz++;
		buckets[target].emplace_front(key,value);
		return false;
	}

	size_t size() const { std::unique_lock<std::mutex> l(m); return sz ; }
```

```
64
65 };
66
67 } /* namespace pr */
68
69 #endif /* SRC_HASH_H_ */
```

Barème :

- 20 % la classe stocke un mutex mutable
- 30 % put protégée
- 30 % get protégée
- 20 % size protégée

1.2 Verrouillage fin

Il est actuellement impossible d'accéder simultanément à des entrées de la table, même si elles sont différentes. De plus, on estime que la partie la plus coûteuse du code est l'itération sur la liste chaînée contenue dans un *bucket* cible (lignes 38 et 49 du code fourni).

On souhaite au contraire construire une version MT-safe de la map, qui introduit un mutex différent pour chaque *bucket* et réalise une protection plus fine. On ne veut interdire que les accès concurrents à un **bucket donné**. Donc deux accès concurrents (*put* et/ou *get*) à des entrées logées dans des buckets différents doit rester possible.

De plus, pour protéger les accès à la taille *size* de la table de hash, on propose d'introduire un `atomic`.

Question 2. (3,5 points) Suivant ces instructions, dans un nouveau fichier `MultiHashMap.h`, programmer cette synchronisation fine.

HashMap.h

```
1  #ifndef SRC_HASHMAP_H_
2  #define SRC_HASHMAP_H_
3
4  #include <cstddef>
5  #include <ostream>
6
7  #include <forward_list>
8  #include <vector>
9  #include <mutex>
10 #include <atomic>
11
12 namespace pr {
13
14 template <typename K, typename V>
15 class HashMap {
16       mutable std::vector<std::mutex> muts;
17 public:
18       class Entry {
19       public :
20             const K key;
21             V value;
22             Entry(const K &k,const V& v):key(k),value(v){}
23       };
24 private :
```

```
25
26      typedef std::vector<std::forward_list<Entry> > buckets_t;
27      // storage for buckets table
28      buckets_t buckets;
29      // total number of entries
30      std::atomic<size_t> sz;
31
32  public:
33      HashMap(size_t size): muts(size),buckets(size),sz(0) {
34            // le ctor buckets(size) => size cases, initialisées par défaut.
35      }
36
37      V* get(const K & key) {
38            size_t h = std::hash<K>()(key);
39            size_t target = h % buckets.size();
40            std::unique_lock<std::mutex> l(muts[target]);
41            for (Entry & ent : buckets[target]) {
42                  if (ent.key==key) {
43                        return & ent.value;
44                  }
45            }
46            return nullptr;
47      }
48
49      bool put (const K & key, const V & value) {
50            size_t h = std::hash<K>()(key);
51            size_t target = h % buckets.size();
52            std::unique_lock<std::mutex> l(muts[target]);
53            for (Entry & ent : buckets[target]) {
54                  if (ent.key==key) {
55                        ent.value=value;
56                        return true;
57                  }
58            }
59            sz++;
60            buckets[target].emplace_front(key,value);
61            return false;
62      }
63
64      size_t size() const { return sz ; }
65  };
66
67  } /* namespace pr */
68
69  #endif /* SRC_HASH_H_ */
```

Barème :

- 20 % la classe stocke un vector<mutex>
- 30 % put protégée
- 30 % get protégée
- 20 % size protégée

2 Sémaphore (9 points)

NB: les parties de cet exercice sont indépendantes ; on peut en particulier supposer la classe sémaphore élaborée en partie 1 correcte pour répondre aux parties 2 et 3, ou répondre à ces parties

sans utiliser le sémaphore de la partie 1.

2.1 Classe Sémaphore

Un sémaphore est un objet utilisé pour la synchronisation dans les applications concurrentes. Le sémaphore contient un compteur (int), qui représente les ressources disponibles, défini à la construction. Le compteur ne peut pas passer en négatif ; un thread qui essaie d'acquérir plus de ressources que ce qui est disponible se bloque.

Le sémaphore offre l'API suivante :

- `void acquire(int qte)` : (souvent noté P dans la littérature) tente d'acquérir `qte` occurences de la ressource (i.e. décrémente le compteur). S'il n'y a pas suffisament de ressources disponibles, cette opération est bloquante ; le thread qui invoque `acquire` restera bloqué jusqu'à ce que l'acquisition soit possible.
- `void release(int qte)` : (souvent noté V dans la littérature) relâche ou produit `qte` occurences de la ressource (i.e. incrémente le compteur). Cette opération doit aussi réveiller les threads qui sont bloqués dans `acquire` le cas échéant.

Question 3. (3 points) Complétez la classe sémaphore.

Semaphore.h

```
#pragma once

namespace pr {

class Semaphore {
      int compteur;
public :
      Semaphore(int initial);
      void acquire(int qte);
      void release(int qte);
};

}
```

Semaphore.h

```
#pragma once
#define SEMAPHORE_H_

#include <thread>
#include <mutex>
#include <condition_variable>

namespace pr {

class Semaphore {
      int val;
      std::mutex m;
      std::condition_variable cv;
public :
      Semaphore(int val):val(val) {}
      void acquire(int qte) {
            std::unique_lock<std::mutex> l(m) ;
            cv.wait(l,[&]{ return val >=qte;});
            val -= qte;
```

```
20        }
21        void release(int qte) {
22              {
23                    std::unique_lock<std::mutex> l(m);
24                    val += qte;
25              }
26              cv.notify_all();
27        }
28  };
29
30  }
```

Barème :

- 10 constructeur
- 10 attributs mutex et cond
- 50 sur acquire : 20 lock du mutex couvre toute l'action, 20 wait correct avec un while ou lambda, 10 décrément
- 30 sur release : 15 lock correct couvrant ou non la notification, 15 le notify_all systématique

2.2 Alternance entre deux threads

NB : Comme l'exercice contient deux programmes indépendants, mais que eclipse ne veut qu'un programme par projet, sous eclipse, renommez un des deux *main* en *main*2. Si vous n'utilisez pas eclipse, le makefile produit deux binaires.

On fournit le programme suivant à compléter :

- Le main doit instancier un thread qui exécute `pinger` et un thread qui exécute `ponger`.
- Le programme ne doit pas contenir de variables globales.
- Le programme doit se terminer quand on a vu `NBITER` répétitions de "ping pong " sur la console. La trace doit commencer par "ping" et se terminer par "pong".
- La solution attendue n'utilise **que** la classe sémaphore comme mécanisme de synchronisation (ni mutex, ni condition). Cependant les solutions qui utilisent directement ces primitives seront acceptées.

Question 4. (3 points) Suivant ces instructions, complétez le programme pour réaliser une alternance "ping pong ping...".

Indices :

- Avec un seul sémaphore, l'interprétation du compteur de "ressource" est ambigü ; quand il est à 1 est-ce le tour de faire "ping" ou "pong" ?
- Le code de pinger et ponger vont rester symétriques, les initialisations peuvent cependant être asymétriques pour assurer que "ping" précède le premier "pong".

pingpong.cpp

```
1  #include "Semaphore.h"
2  #include <iostream>
3
4  void pinger(int n) {
5        for (int i=0; i < n ; i++) {
6              std::cout << "ping ";
7        }
8  }
9
10 void ponger(int n) {
11       for (int i=0; i < n ; i++) {
```

```
            std::cout << "pong ";
      }
}

int main () {
      // a faire varier si on le souhaite
      const int NBITER = 20;

      // TODO : instancier un thread pinger et un thread ponger avec n=NBITER

      // TODO : terminaison et sortie propre

      return 0;
}
```

pingpong.cpp

```
#include "Semaphore.h"
#include <thread>
#include <vector>
#include <iostream>

// Fichier à completer
void pinger(int n, pr::Semaphore & sema, pr::Semaphore & semb) {
      for (int i=0; i < n ; i++) {
            sema.acquire(1);
            std::cout << "ping ";
            semb.release(1);
      }
}

void ponger(int n, pr::Semaphore & sema, pr::Semaphore & semb) {
      for (int i=0; i < n ; i++) {
            semb.acquire(1);
            std::cout << "pong ";
            sema.release(1);
      }
}

int main () {
      // a faire varier
      const int NBITER = 20;

      pr::Semaphore sema(1);
      pr::Semaphore semb(0);

      std::vector<std::thread> threads;
      // instancier NBREAD threads lecteurs
      threads.emplace_back(pinger,NBITER, std::ref(sema), std::ref(semb));
      // instancier NBWRITE threads écrivains
      threads.emplace_back(ponger,NBITER, std::ref(sema), std::ref(semb));

      // sortie propre
      for (auto & t: threads)
            t.join();

      return 0;
}
```

- _
  _
  _
-
-

2.3 Lecteurs et Ecrivains

On considère le programme suivant, qui représente des threads lecteurs et écrivains qui travaillent simultanément sur une donnée partagée (classe `Data`).

readwrite.cpp

```cpp
#include "Semaphore.h"
#include <thread>
#include <vector>

// TODO : classe à modifier
class Data {
	std::vector<int> values;
public :
	int read() const {
		if (values.empty())
			return 0;
		else
			return values[rand()%values.size()];
	}
	void write() {
		values.push_back(rand());
	}
};

// Pas de modifications dans la suite.
void worker(Data & data) {
	for (int i=0; i < 20 ; i++) {
		auto r = ::rand() % 1000 ; // 0 to 1 sec
		std::this_thread::sleep_for (std::chrono::milliseconds(r));
		if (r % 2)
			auto lu = data.read();
		else
			data.write();
	}
}

int main () {
	// a faire varier
	const int NBTHREAD=10;

	// le data partagé
	Data d;

	std::vector<std::thread> threads;
```

```
40        for (int i=0; i < NBTHREAD; i++)
41              threads.emplace_back(worker,std::ref(d));
42
43        for (auto & t: threads)
44              t.join();
45        return 0;
46  }
```

Question 5. (3 points) Toujours à l'aide de la classe sémaphore, modifiez la classe `Data` pour que :

- plusieurs lecteurs (au maximum 256) puissent exécuter le code de `read` en concurrence,
- un seul écrivain à la fois ne doit pouvoir exécuter `write`,
- lecteurs et écrivains soient mutuellement exclusifs
- La solution attendue n'utilise **que** la classe sémaphore comme mécanisme de synchronisation (ni mutex, ni condition). Cependant les solutions qui utilisent directement ces primitives seront acceptées.

Indice : on introduira au moins un sémaphore initialisé avec la valeur fournie 256 dans la solution.

readwrite.cpp

```cpp
#include "Semaphore.h"
#include <thread>
#include <vector>

// classe à modifier
class Data {
      std::vector<int> values;
      mutable pr::Semaphore sem;
public :
      Data():sem(256){}
      int read() const {
            sem.acquire(1);
            int toret;
            if (values.empty())
                  toret = 0;
            else
                  toret = values[rand()%values.size()];
            sem.release(1);
            return toret;
      }
      void write() {
            sem.acquire(256);
            values.push_back(rand());
            sem.release(256);
      }
};

// Pas de modifications dans la suite.
void worker(Data & data) {
      for (int i=0; i < 20 ; i++) {
            auto r = ::rand() % 1000 ; // 0 to 1 sec
            std::this_thread::sleep_for (std::chrono::milliseconds(r));
            if (r % 2)
                  auto lu = data.read();
            else
                  data.write();
      }
}
```

```cpp

int main () {
	// a faire varier
	const int NBTHREAD=10;

	// le data partagé
	Data d;

	std::vector<std::thread> threads;
	for (int i=0; i < NBTHREAD; i++)
		threads.emplace_back(worker,std::ref(d));

	for (auto & t: threads)
		t.join();
	return 0;
}
```

Barème :

- 30 % la classe data porte un sémaphore (20%), déclaré mutable (10%)
- 10 % initialisation à 256 dans le ctor
- 30 % P(1), V(1) encadre le code lecteur
- 30 % P(256), V(256) encadre le code écrivain

Une solution (correcte) qui utilise mutex/conditions au lieu de la classe sémaphore est notée sur 80 % des points.

3 Itérateurs concaténés (6 points)

Question 6. (6 points) Ecrivez une classe `concat` qui puisse s'utiliser avec le code suivant :

concat.cpp

```cpp
#include "concat.h"
#include <vector>
#include <iostream>
#include <string>

using namespace std;
using namespace pr;

int main () {
	vector<string> v1;
	v1.push_back("abc"); v1.push_back("def");

	vector<string> v2;
	v2.push_back("ghi"); v2.push_back("klm");

	// sans faire de copies !
	concat conc = concat(v1,v2);
	for (const string & s : conc) {
		cout << s << ":";
	}
	cout << endl;
	return 0;
}
```

Indices :

- On vous fournit un fichier `concat.h` à compléter.
- La classe à réaliser ne fonctionnera que pour deux `vector<string>`. Ceci permet d'éviter les problèmes liés à la généricité (`typename`, syntaxe...) ; le code demandé n'a donc pas besoin d'introduire d'argument template.
- La classe `concat` à la construction doit stocker des pointeurs ou des références vers ses constituants (des `vector<string>`). Elle ne doit en aucun cas copier des données.
- La classe `concat` est donc itérable ; elle doit proposer `begin` et `end` rendant des itérateurs correctement positionnés au début du premier vecteur et au delà de la fin du second respectivement.
- Un itérateur sur une concaténation fonctionne de la manière suivante :
  - Il stocke un itérateur sur un des deux vecteurs qu'on a concaténés (la position courante).
  - Il stocke un pointeur ou une référence vers son contexte (l'instance de `concat` dans laquelle il itère)
  - Déréférencer l'itérateur correspond à un accès à la position courante
  - Pour incrémenter ou décaler l'itérateur on se décale sur la position courante, si l'on a atteint la fin du premier vecteur, on se place au début du second

Fichier fourni :

concat.h

```
#pragma once

#include <vector>
#include <string>

namespace pr {

class concat {
      // TODO : attributs stockant ref ou pointeurs vers les constituants v1,v2
public:
      concat(std::vector<std::string> & v1, std::vector<std::string> & v2);

      class iterator {
            // TODO : attributs
      public:
            // TODO : signature du constructeur
            iterator(/* A COMPLETER */);
            // TODO : contrat itérateur
            std::string & operator*();
            iterator & operator++();
            bool operator!=(const iterator & other) const;
      };

      iterator begin() ;
      iterator end() ;
};

}
```

concat.cpp

```
#pragma once

#include <vector>
#include <string>
```

```
5
6  namespace pr {
7
8  class concat {
9        std::vector<std::string> & v1;
10       std::vector<std::string> & v2;
11 public:
12       concat(std::vector<std::string> & v1, std::vector<std::string> & v2):v1(v1),v2(v2)
              {}
13
14       class iterator {
15             std::vector<std::string>::iterator ite;
16             concat & context;
17       public:
18             iterator(const std::vector<std::string>::iterator & ite, concat & context):
                  ite(ite),context(context) {}
19             std::string & operator*() {
20                   return *ite;
21             }
22             iterator & operator++() {
23                   ++ite;
24                   if (ite == context.v1.end()) {
25                         ite = context.v2.begin();
26                   }
27                   return *this;
28             }
29             bool operator!=(const iterator & other) const {
30                   return ite != other.ite;
31             }
32       };
33
34       iterator begin() {
35             return iterator(v1.begin(),*this);
36       }
37       iterator end() {
38             return iterator(v2.end(),*this);
39       }
40 };
41
42 }
```

Barème :

- 20 % la classe stocke par ref ou pointeur les deux vector, positionnés dans le ctor
- 20 % begin et end corrects
- 20 % attributs de l'itérateur
- 20 % operator* et operator!=
- 20 % operator++