SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 2 PR
Master 1 Informatique Jan 2019

UE 4I400

Année 2018-2019

2 heures – **Tout document papier autorisé**
Tout appareil de communication électronique interdit (téléphones...)

Introduction

- Le barème est sur 22 points et est donné à titre indicatif et susceptible d'être modifié.
- Dans le code C++ demandé vous vous efforcerez d'ecrire du code correct et compilable, mais les petites typos ne seront pas sanctionnées.
- De même les problèmes d'include et de namespace ne sont pas pertinents (pas la peine de qualifier les `std::`, ni de citer tous les `#include`)
- On suppose également que **tous les appels système se passent bien**, on ne demande pas de contrôler les valeurs de retour (donc pas de `perror` ou `errno.h`).

Le sujet est composé de trois exercices indépendants qu'on pourra traiter dans l'ordre qu'on souhaite.

1 Synchronisations (11 points)

Notions testées : concurrence et synchronisations.

On considère une variante du problème du dîner des philosophes, proposé par Dijkstra. On considère N threads, qui travaillent ensemble sur un vecteur tab de N cases.

Chaque thread d'indice i a le comportement suivant (répété 100 fois):

- Il "réflechit" pendant un temps variable, ce qui est représenté par l'invocation à `bool think()`, une fonction qui ne sera pas détaillée dans ce sujet.
- S'il est satisfait du résultat de sa réflexion (`think` a rendu vrai), il échange la valeur de la case $tab[i]$ avec celle de $tab[(i+1)\%N]$. On a donc une gestion circulaire du tableau.

Question 1. (3 points) Complétez la fonction `main` qui crée `tab` (initialisé avec comme contenu les entiers de 0 à $N-1$), engendre les N threads "philosophe", attends leur terminaison et enfin affiche le contenu de tab sur la sortie standard. Donnez la signature de la fonction `philosophe` cohérente avec ce main, et compléter son corps.
NB : on ne demande pas encore de coder de synchronisations.

main.cpp

```cpp
const int N = 5;

// temps de reflexion arbitraire
// rend true ou false de façon arbitraire
// la fonction ne sera pas détaillée.
bool think ();

// fonction executée par chacun des thread
// Signature à fournir en Q1.
???? philosophe ( ???? ) {
      for (int iter = 0 ; iter < 100 ; iter++) {
            if (think()) {
                  // echanger les cases d'indice i et (i+1) % N de tab.
            }
      }
}

// Question 1 : compléter le corps de main.
int main() {
      // declarer tab
      // initialiser tab avec les entiers 0 à N-1
      // engendrer N threads exécutant "philosophe" ; leur passer leur indice de création et tab
      // attendre la terminaison de tous les threads
      // afficher tab
      return 0;
}
```

main.cpp

```cpp
#include <vector>
#include <thread>
#include <cmath>
#include <iostream>

using namespace std;

const int N = 5;

// temps de reflexion arbitraire
// rend true ou false de façon arbitraire
// la fonction ne sera pas détaillée.
bool think () {
      std::this_thread::sleep_for( std::chrono::milliseconds((rand() % 1000)) );
      return rand() % 2 == 0;
}

// fonction executée par chacun des thread
// Signature à fournir en Q1 : 15 %
void philosophe (int i, vector<int> & tab) {
      for (int iter = 0 ; iter < 10 ; iter++) {
            if (think()) {
                  // echanger les cases d'indice i et (i+1) % N de tab. : 20 %
                  int c = tab[i];
                  tab[i] = tab[(i+1) % N];
                  tab[(i+1) % N] = c;
            }
```

```
28        }
29  }
30
31  // Question 1 : compléter le corps de main.
32  int main() {
33        // declare et init (sans faute mémoire) : 10%
34        // declarer tab
35        vector<int> tab (N,0);
36        // initialiser tab avec les entiers 0 à N-1
37        for (int i=0;i<N;i++) { tab[i] = i ;}
38
39
40        // engendrer N threads exécutant "philosophe" : 15%
41        vector<thread> threads;
42        threads.reserve(N);
43        for (int i=0;i<N;i++) {
44              // leur passer leur indice de création et tab : 20 %
45              threads.emplace_back(thread(philosophe,i,ref(tab)));
46        }
47        // attendre la terminaison de tous les threads : 10%
48        for (int i=0;i<N;i++) {
49              threads[i].join();
50        }
51        // afficher tab 10%
52        for (int i=0;i<N;i++) {
53              cout << tab[i] << ", ";
54        }
55        cout << endl;
56        return 0;
57  }
```

Barème :

15 % la signature de philosophe rend void, et prend bien un indice de création i et le vecteur par référence ou pointeur

20 % le swap de deux cases du tableau, a priori avec un temporaire

10 % déclaration et initialisation du vecteur

15 % on engendre bien n threads

20 % passage des paramètres à la fonction thread, avec un std::ref si nécessaire.

10 % join des n threads

10 % affichage du vecteur

Question 2. (2 points) Le programme actuel, sans synchronisations est évidemment incorrect. Ajoutez des synchronisations pour protéger l'accès aux cases du tableau, tout en **maximisant** la concurrence : deux philosophes qui ne sont pas adjacents doivent pouvoir travailler en même temps. Vous expliquerez comment déclarer ces éléments (dans le main) et vous donnerez les modifications à réaliser dans la fonction `philosophe`.

main.cpp

```
1  #include <vector>
2  #include <thread>
3  #include <cmath>
4  #include <iostream>
5  #include <mutex>
6
```

```cpp
using namespace std;

const int N = 5;

// temps de reflexion arbitraire
// rend true ou false de façon arbitraire
// la fonction ne sera pas détaillée.
bool think () {
      std::this_thread::sleep_for( std::chrono::milliseconds((rand() % 1000)) );
      return rand() % 2 == 0;
}

// fonction executée par chacun des thread
void philosophe (int i, vector<int> & tab, vector<mutex> & mutexes) {
      for (int iter = 0 ; iter < 10 ; iter++) {
            if (think()) {
                  mutexes[i].lock();
                  mutexes[(i+1)%N].lock();

                  // echanger les cases d'indice i et (i+1) % N de tab.
                  int c = tab[i];
                  tab[i] = tab[(i+1) % N];
                  tab[(i+1) % N] = c;

                  mutexes[i].unlock();
                  mutexes[(i+1)%N].unlock();
            }
      }
}

// Question 1 : compléter le corps de main.
int main() {
      // declarer tab
      vector<int> tab (N,0);
      // initialiser tab avec les entiers 0 à N-1
      for (int i=0;i<N;i++) { tab[i] = i ;}
      // engendrer N threads exécutant "philosophe" ; leur passer leur indice de cré
          ation et tab
      vector<thread> threads;
      threads.reserve(N);
      // initialiser les mutex
      vector<mutex> mutexes (N);
      for (int i=0;i<N;i++) {
            threads.emplace_back(thread(philosophe,i,ref(tab),ref(mutexes)));
      }
      // attendre la terminaison de tous les threads
      for (int i=0;i<N;i++) {
            threads[i].join();
      }
      // afficher tab
      for (int i=0;i<N;i++) {
            cout << tab[i] << ", ";
      }
      cout << endl;
      return 0;
}
```

Barème :
20 % idée : on introduit N mutex ou sémaphores initialisés à 1

10 % syntaxe de déclaration et initialisation cohérente

20 % on les passe à la fonction philosophe/constructeur de thread (ils ne sont pas déclarés/instanciés dans philosophe par exemple)

50 % on lock avant de swap les deux mutex, on delock après. 20 % si on encadre le test de think. 0 % si on encadre la boucle. 0 % si on ne prend pas deux locks simultanément à un moment.

Question 3. (1 point) Le programme actuellement se bloque parfois avant de se terminer, expliquez pourquoi en exhibant une séquence d'exécution possible du programme.

Interblocage circulaire, tout le monde a une "fourchette" à la main.

Barème :

50 % la séquence est exhibée de façon raisonnablement lisible, on comprend dans la réponse que ce scenario est du à l'ordonnanceur, qu'on ne contrôle pas...

50 % on conclut avec l'explication de l'interblocage / on explique correctement l'interblocage circulaire.

Pour remédier à ce problème, vous allez coder un système de tickets de manière à ce que au plus $N-1$ philosophes puissent commencer à travailler en même temps. Chaque philosophe prend un ticket avant de commencer à travailler, et le rend avant d'entamer une nouvelle étape de réflexion. Dans les questions suivantes, on demande de développer une classe `TicketBarrier`, son constructeur prenant un entier (le nombre de tickets disponibles), ses opérations `void enter()` et `void leave()`.

Question 4. (1 point) Donnez les modifications à apporter au code du programme pour utiliser cette classe `TicketBarrier`. La classe elle-même sera implantée dans les questions suivantes.

Instancier dans le main + passer aux philosophes.

main.cpp

```
1       // engendrer N threads exécutant "philosophe" ; leur passer leur indice de cré
            ation et tab
2       vector<thread> threads;
3       threads.reserve(N);
4       // initialiser les mutex
5       vector<mutex> mutexes (N);
6
7       // AJOUT : tickets
8       TicketBarrier tb (N-1);
9
10      for (int i=0;i<N;i++) {
11            threads.emplace_back(thread(philosophe,i,ref(tab),ref(mutexes),ref(tb)));
12      }
```

Modifier le corps de boucle dans philosophe pour utiliser la barrière.

main.cpp

```
1   // AJOUT : paramètre tb
2   void philosophe (int i, vector<int> & tab, vector<mutex> & mutexes, TicketBarrier & tb)
        {
3       for (int iter = 0 ; iter < 10 ; iter++) {
4             if (think()) {
5                   tb.enter(); // NEW
6
7                   mutexes[i].lock();
8                   mutexes[(i+1)%N].lock();
```

```
9
10                  // echanger les cases d'indice i et (i+1) % N de tab.
11                  int c = tab[i];
12                  tab[i] = tab[(i+1) % N];
13                  tab[(i+1) % N] = c;
14
15                  mutexes[i].unlock();
16                  mutexes[(i+1)%N].unlock();
17
18                  tb.leave(); // NEW
19              }
20      }
21  }
```

Barème :

40 % Instancier dans le main, et passer l'objet (unique) à tous les threads (dont 20% sur la mise à jour de la signature et invocation à thread)

60 % La barrière encadre le corps du "swap", il est dans le test if (think). 30 % si on encadre aussi le test de think. 0 % si on encadre la boucle. 0 % s'il ne protège pas l'acquisition du premier mutex de la question précédente.

Question 5. (2 points) Ecrire une version de la classe `TicketBarrier` en n'utilisant que les primitives fournies dans `std::` par C++11.

main.cpp

```
1  class TicketBarrier {
2        const int K;
3        int cur;
4        mutex m;
5        condition_variable cv;
6  public :
7        TicketBarrier(int K):K(K),cur(0){};
8
9        void enter() {
10             unique_lock<mutex> l (m);
11             while (cur >= K) { cv.wait(l) ; }
12             cur ++;
13       }
14       void leave() {
15             unique_lock<mutex> l (m);
16             cur--;
17             cv.notify_all();
18       }
19 };
```

Barème :

30 % On a un compteur, un mutex, un condition_variable. Le corrigé utilise deux compteurs pour plus de lisibilité mais on peut faire avec un seul qu'on décrémente.

10 % tous les accès au compteur sont correctement protégés par le mutex

30 % enter est bloquant quand il le faut

30 % leave notifie (quand il le faut, ou comme dans le corrigé à chaque fois)

Question 6. (2 points) Ecrire une version de la classe `TicketBarrier` en n'utilisant que des primitives POSIX.

main.cpp

```
class TicketBarrier {
      sem_t sem;
public :
      TicketBarrier(int K){
            sem_init (&sem, 0, K);
      };
      void enter() {
            sem_wait(&sem);
      }
      void leave() {
            sem_post(&sem);
      }
};
```

20 % la classe ne porte que un sémaphore. 10 % si on a aussi mis un compteur.
20 % initialisation du sémaphore à la valeur souhaitée.
30 % enter bloque quand il le faut.
30 % leave notifie au bon moment.

2 Invocation Distante (8 points)

On considère l'interface `IValue` et sa réalisation triviale `Value` suivante :

IValue.h

```
#pragma once

class IValue {
public :
      virtual void set (int val) = 0;
      virtual int get () const = 0;
      virtual ~IValue() {}
};

class Value : public IValue {
      int val;
public :
      Value (int v=0):val(v) {}
      void set (int v) { val = v; }
      int get () const { return val; }
      ~Value() {}
};
```

Notre objectif est de permettre un accès par un client à une instance de `Value` distante, c'est à dire hebergée sur une autre machine serveur, en suivant les principes d'un proxy distant. On considère que les machines sont connectées par un réseau IPv4 classique. L'exercice est centré sur le réseau, et ne nécessite donc pas de mécanismes liés à la concurrence.

Question 1. (1 point) Selon que l'utilisateur du logiciel passe les addresses de machine sous la forme "`hote.reseau.domaine`" (e.g. "`www.google.fr`") ou sous la forme d'une ip en quatre parties "`a.b.c.d`" (e.g. "192.168.0.12"), quelle primitive(s) de conversion faut-il utiliser pour obtenir une addresse IP au format machine (utilisable dans un `struct sockaddr_in` par exemple) ? Expliquez rapidement la différence entre ces familles de fonctions.

On suppose dans les questions suivantes que l'on dispose d'une fonction `in_addr_t getIP (const string & host)` implantant ce service de conversion.

> Le format machine pour une IP = un entier non signé, 4 octets, dans le format "network".
>
> Pour les addresses "192.168.0.12", aller vers ces 4 octets = coder sur un octet l'entier qu'on parse entre chaque point. Simple fonction de conversion ou parse, nature proche d'un "atoi/itoa". "inet_ntoa" pour network to string, et "inet_addr" pour l'autre direction.
>
> Pour les addresses "`www.google.fr`" c'est différent, on doit interroger le système de nommage DNS. Il y a plusieurs API pour ce faire, on a utilisé "getaddrinfo" et "getnameinfo" dans l'UE.
>
> Barème
>
> 30 % la conversion est triviale pour les IP numériques
>
> 20 % on cite une fonction de la famille "inet_*"
>
> 30 % il faut passer par le DNS pour les IP nommées
>
> 20 % on cite une fonction d'interaction DNS.

Question 2. (2,5 points) Ecrivez une classe `ValueServer` qui sera hebergée sur la machine serveur et stocke une référence vers une instance de `IValue` (le sujet). Elle est munie d'un constructeur dont vous préciserez la signature, et d'une opération `void treatClient()`. A la construction, la classe doit mettre en place les mécanismes réseau utiles pour pouvoir recevoir des demandes de clients. La méthode `treatClient` traite une demande d'un client : établissement de la connection, reconnaitre si le client veut invoquer "set" ou "get", traiter sa demande **par délégation sur le sujet** et enfin mettre fin à la connection.

main_serveur.cpp

```cpp
#include "IValue.h"
#include <sys/socket.h>
#include <netinet/in.h>
#include <string>
#include <cstring>
#include <unistd.h>
#include <iostream>

using namespace std;

class ValueServer {
	int sacc;
	IValue * subject;
public :
	ValueServer(int port, IValue * subject) : subject(subject){
		// create socket
		int fd = socket(AF_INET,SOCK_STREAM,0);
		// bind
		struct sockaddr_in sin;
		sin.sin_family = AF_INET;
		sin.sin_addr.s_addr = htonl(INADDR_ANY);
		sin.sin_port = htons(port);

		bind(fd, (struct sockaddr *) &sin, sizeof(sin));
		// listen
		listen(fd, 50);

		sacc = fd;
	}
	void treatClient() {
		int scon = ::accept(sacc, nullptr, nullptr);
		int val=0;
		read(scon, &val, sizeof (int));
		if (val == 0) {
			// get
			val = subject->get();
			write (scon, &val, sizeof (int));
		} else if (val == 1) {
			// set
			read(scon, &val, sizeof (int));
			subject->set(val);
		}
		close (scon);
	}
	~ValueServer() {
		close (sacc);
	}
};

int main () {
	Value realval;
	// port serveur
	ValueServer server (1664, & realval);
	for (int i = 0; i < 10 ; i++) {
		server.treatClient();
	}
	return 0;
```

```
}
```

10 % le constructeur prend une IValue (par réf ou pointeur) et un numéro de port.
20 % le constructeur enchaîne les appels à socket, bind, listen avec des paramètres raisonnables.
10 % treatclient démarre par un accept/récupère la sock de communication et s'en sert.
20 % on commence par lire un code qui identifie l'action à faire/présence de switch ou équivalent
15 % dans les deux cas, on voit une délégation sur le sujet
15 % dans les deux cas, on voit l'interaction client raisonnable (read/write)
10 % close de la socket *de communication* en fin de treatclient

Question 3. (2,5 points) Ecrivez une classe `ValueProxy` qui sera hebergée sur la machine client. Elle est munie d'un constructeur dont vous préciserez la signature, et réalise l'interface `IValue` en appui sur une connection réseau vers un `ValueServer`.

main_client.cpp

```
#include "IValue.h"
#include <sys/socket.h>
#include <netinet/in.h>
#include <string>
#include <cstring>
#include <netdb.h>
#include <unistd.h>
#include <iostream>

using namespace std;

in_addr_t getIP (const string & host) {
      struct addrinfo * addrq;
      getaddrinfo(host.c_str(), /* service*/ nullptr, /* hints*/ nullptr, &addrq);
      auto toret =((struct sockaddr_in *) addrq->ai_addr)->sin_addr.s_addr;
      freeaddrinfo(addrq);
      return toret;
}

class ValueClient : public IValue {
      sockaddr_in addr;
public :
      ValueClient(const string & ipstr, int port) {
            ::memset (& addr, 0, sizeof addr);
            addr.sin_family = AF_INET;
            addr.sin_port = htons(port);
            addr.sin_addr.s_addr = getIP(ipstr);
      }
      int get () const {
            int sock = socket(AF_INET,SOCK_STREAM,0);
            connect(sock,(struct sockaddr *)&addr,sizeof addr);
            int val = 0;
            write (sock, &val, sizeof (int));
            read(sock, &val, sizeof (int));
            close(sock);
            return val;
      }
      void set (int newval) {
            int sock = socket(AF_INET,SOCK_STREAM,0);
```

```
40              connect(sock,(struct sockaddr *)&addr,sizeof addr);
41              int val = 1;
42              write (sock, &val, sizeof(int));
43              write (sock, &newval, sizeof(int));
44              close(sock);
45      }
46  };
47
48  int main () {
49      // IP+port serveur
50      IValue * val = new ValueClient("127.0.0.1",1664);
51      cout << "Current val : "<< val->get() << std::endl;
52      cout << "Setting to 64" << std::endl;
53      val->set(64);
54      cout << "Current val : "<< val->get() << std::endl;
55      delete val;
56      return 0;
57  }
```

10 % la classe réalise IValue.

20 % le constructeur prend une IP et un port

30 % on voit une séquence composée de socket (10%), connect (10%), (read ou write), close (10%). On en donne que 10 % si les invocations sont mal placées, e.g. ouverture de la socket dans le constructeur.

20 % le get fait write (code) puis read du résultat et le rend

20 % le set fait write (code) puis write de la valeur argument

Question 4. (2 points) Ecrivez un main client et un main serveur qui utilisent ces classes.

cf corrigé précedent.

20 % le client utilise un couple IP+port

20 % le client se sert de l'instance "normalement" pour faire au moins un get et un set, c'est un proxy.

20 % le serveur instancie le vrai sujet et le passe au ValueServer

20 % le serveur se choisit un numéro de port

20 % le serveur invoque treatClient.

3 Questions de Cours (3 points)

Question 1. (1 point) Expliquez dans quel contexte la primitive `select` peut être utile.

Attente simultanée sur plusieurs descripteurs de fichier.

50 % on cite un contexte d'utilisation particulier (e.g. quand le thread attend sur la socket...)

50 % on explique correctement ce que ça fait en général.

Question 2. (1 point) On considère un programme où deux threads se synchronisent à l'aide de signaux (primitives `kill` et `sigsuspend`). Le comportement obtenu ne correspond pas aux attentes du programmeur. Expliquez pourquoi.

On ne peut pas cibler un thread particulier au sein du processus.

Barème : Binaire. En particulier tout ce qui est hors sujet vaut zéro.

Question 3. (1 point) Dans quelle(s) situation(s) doit-on utiliser un segment de mémoire partagée **nommé** (au lieu d'un segment anonyme) ?

> Si on doit s'en servir entre processus qui ne sont pas issus de la même famille de processus.
> 40 % l'idée d'arborescence de processus est présente (vague)
> 60 % explication claire et pertinente