SCIENCES SORBONNE UNIVERSITÉ

Examen Session 2 PR
Master 1 Informatique Juin 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 souhaite instancier N threads qui vont travailler sur une donnée partagée ; on suppose une classe `SharedData`, munie d'un constructeur sans argument, et qui porte en attribut toutes les données utiles. Tous les threads exécutent la même fonction `worker`; cependant leur comportement dépend de leur identifiant, passé à la construction.

Question 1. (2 points)

a) Définir une classe ou struct `SharedData`, munie en attribut d'un vecteur C++ standard de double `tab` déclaré `public` pour simplifier la suite du code.

b) Donnez la signature de la fonction `worker` cohérente avec l'énoncé.

c) Ecrivez un programme C++ qui :

- lit dans son premier argument un entier N (on ne demande pas de traiter les erreurs de parse ou de pas assez d'arguments)
- crée N threads ouvriers concurrents, qui exécutent la fonction *worker* comme décrit ci-dessus
- attend leur terminaison et sort sans erreur

main.cpp

```
1 #include <cstdlib>
2 #include <vector>
```

```cpp
#include <thread>

using namespace std;

struct SharedData {
	vector<double> tab;
};

// version par ref
// void worker2 (int id, SharedData & data) { }
void worker (int id, SharedData * data) {

}

int main (int argc, const char * argv[]) {
	int N = atoi(argv[1]);
	SharedData data;
	vector<thread> threads ;
	for (int i=0; i < N ; i++) {
		threads.emplace_back(thread(worker,i,&data));
		// /!\ au std::ref sur la version par ref
		// threads.emplace_back(thread(worker2,i,ref(data)));
	}
	for (auto& t:threads) {
		t.join();
	}
	return 0;
}
```

Barème : TODO

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)

On s'intéresse à présent au corps de la fonction worker. Pour abstraire des détails du système, on va supposer trois fonctions, dont le corps ne sera pas détaillé dans cet énoncé :

- `void treatOther()` le thread qui invoque cette fonction va passer un temps non déterminé à exécuter d'autres tâches, sans rapport avec la partie du système considéré. De notre point de vue, c'est comme s'il dormait pour une durée aléatoire.
- `void analyze(const std::vector<double> & tab)` cette opération consiste en une analyse plus ou moins longue des données; il est essentiel que les données ne soient pas modifiés pendant l'analyse.
- `void normalize(std::vector<double> & tab)` cette opération "renormalise" le vecteur, une opération qui consiste à calculer le maximum du vecteur, puis en fonction du max à mettre à jour tous les éléments du vecteur.

a) Expliquez la signature de *analyze* et *normalize* ; décrivez ce que ces déclarations changent sur les manipulations possibles de *tab* au sein de ces fonctions.

b) Tous les thread vont boucler 100 fois; à chaque tour de boucle, ils doivent invoquer une fois

`treatOther`. Les worker d'indice pair sont des lecteurs ; à chaque tour de boucle ils lancent `analyze`. Les worker d'indice impair sont des écrivains; à chaque tour ils lancent `normalize`. Ecrivez le corps de la fonction `worker`.

c) Le comportement tel quel est incorrect; l'accès concurrent corrompt le travail des threads. Modifiez le corps de `worker` pour introduire une exclusion mutuelle complète : un seul thread à la fois ne doit pouvoir exécuter *analyze* ou *normalize*. Expliquez aussi comment modifier le reste du code en cohérence.

main.cpp

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

using namespace std;

struct SharedData {
      vector<double> tab;
      mutex mut;
};
void treatOther(){}
void analyze(const std::vector<double> &tab){}
void normalize(std::vector<double> &tab){}

// version par ref
void worker (int id, SharedData & data) {

      for (int i=0; i < 100; i++) {
            treatOther();
            data.mut.lock();
            if (i%2) {
                  normalize(data.tab);
            } else {
                  analyze(data.tab);
            }
            data.mut.unlock();
      }
}
```

Barème : TODO

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 3. (4 points)

Le système actuel est peu efficace ; l'exclusion mutuelle complète est trop restrictive. On propose d'utiliser un type de verrou plus flexible, dit "lecteur/écrivain". Le principe est que plusieurs lecteurs peuvent entrer simultanément en section critique, mais un seul écrivain à la fois, et les écrivains et lecteurs sont mutuellement exclusifs.

On propose l'interface suivante pour ce nouveau type de verrou ; les opérations `wlock/wunlock` correspondent à l'acquisition exclusive du verrou par un écrivain ; les versions `rlock/runlock` correspondent à des acuisitions par des lecteurs.

rwlock.h

```
#pragma once

class RWLock {
public :
      virtual void rlock()=0; // reader lock
      virtual void runlock()=0; // reader unlock
      virtual void wlock()=0; // writer lock
      virtual void wunlock()=0; // writer unlock
      virtual ~RWLock(){}
};
```

a) On propose la réalisation suivante de la classe, utilisant un semaphore POSIX. Critiquer cette approche ; décrivez précisément les problèmes qui peuvent se produire.

rwlock_sem.cpp

```
#include "rwlock.h"
#include <semaphore.h>

class RWLockSem : public RWLock {
      sem_t sem;
      size_t MAX;
public :
      RWLockSem(int MAX):MAX(MAX){
            sem_init (&sem, 0, MAX);
      }

      void rlock () {
            sem_wait(&sem);
      }
      void runlock () {
            sem_post(&sem);
      }
      void wlock () {
            for (int i=0; i <MAX ; i++)
                  sem_wait(&sem);
      }
      void wunlock () {
            for (int i=0; i <MAX ; i++)
                  sem_post(&sem);
      }
};
```

> Avec plus d'un écrivain on risque l'interblocage ; l'acquisition des MAX occurrences n'est pas atomique.

b) Proposez au contraire une implantation correcte de cette interface qui n'utilise que les mécanismes du standard C++ moderne. L'approche recommandée consiste à utiliser deux compteurs, comptabilisant respectivement le nombre de lecteurs et d'écrivains actuellement en section critique.

rwlock.cpp

```cpp
#include "rwlock.h"
#include <mutex>
#include <condition_variable>

using namespace std;

class RWLockStd : public RWLock {
      mutex mut;
      condition_variable cond;
      int currentR, currentW;
public :
      RWLockStd():currentR(0),currentW(0){
      }

      void rlock () {
            unique_lock<mutex> l(mut);
            while (currentW > 0) {
                  cond.wait(l);
            }
            currentR++;
      }
      void runlock () {
            unique_lock<mutex> l(mut);
            currentR--;
            if (currentR==0) {
                  cond.notify_all();
            }
      }
      void wlock () {
            unique_lock<mutex> l(mut);
            while (currentW > 0 || currentR > 0) {
                  cond.wait(l);
            }
            currentW++;
      }
      void wunlock () {
            unique_lock<mutex> l(mut);
            currentW--;
            cond.notify_all();
      }
};
```

c) Donnez les modifications à apporter au code de votre application pour utiliser cette nouvelle classe de verrou.

rwlock.cpp

```cpp

struct SharedData {
      vector<double> tab;
      RWLockStd mut;
};

// version par ref
void worker (int id, SharedData & data) {
```

```
9
10      for (int i=0; i < 100; i++) {
11            treatOther();
12            if (i%2) {
13                  data.mut.wlock();
14                  normalize(data.tab);
15                  data.mut.wunlock();
16            } else {
17                  data.mut.rlock();
18                  analyze(data.tab);
19                  data.mut.runlock();
20            }
21      }
22  }
```

2 Processus, Signaux (5 points)

Notions testées : Créations et fin de processus, signaux.

On considère un programme `ltime` (pour "Limit time") dont le rôle est de limiter le temps d'exécution d'une autre commande arbitraire. Par exemple la commande `ltime 30 analyse.exe data.csv` doit lancer `analyse.exe data.csv`, mais si au bout de 30 secondes cette analyse n'est pas terminée, l'analyse doit tout de même être interrompue.

Ecrivez en C++ un programme POSIX qui implémente ce comportement. Le programme doit lancer avec un nouveau processus pour exécuter la tâche (i.e. exécuter **argv** sans les deux premiers arguments). Il doit positionner une alarme, et se mettre en attente de la fin de la tâche ; s'il est au contraire interrompu par le signal d'alarme, il tue la tâche en simulant l'effet d'un "Controle-C".

On recommande l'utilisation de la primitive `alarm`.
`unsigned int alarm(unsigned int seconds);` : alarm() arranges for a SIGALRM signal to be delivered to the calling process in seconds seconds.

RAPPEL : 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`).

exemple :

```
[ythierry@oxygen src]$ ./ltime 2 /usr/bin/sleep 3
running /usr/bin/sleep
killed by timeout
Child dead
[ythierry@oxygen src]$ ./ltime 4 /usr/bin/sleep 3
running /usr/bin/sleep
Child dead
```

ltime.cpp

```
1  #include <cstdlib>
2  #include <unistd.h>
3  #include <csignal>
4  #include <cstdio>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7
```

```
using namespace std;

pid_t child;

void signaled (int sig) {
      printf("killed by timeout\n");
      kill (child, SIGINT);
}

int main (int argc, char * const * argv) {
      child = fork();
      if (child==0) {
            printf("running %s\n",argv[2]);
            execv(argv[2],argv + 2);
      } else {
            alarm (atoi(argv[1]));
            signal(SIGALRM, signaled);
            wait(nullptr);
            printf("Child dead\n");
      }
      return 0;
}
```

3 Manipulations Mémoire, Processus (4 points)

Notions testées : Communications IPC, mémoire, processus.

On considère le programme suivant, dans lequel un processus A doit construire une liste des 10 entiers de 0 à 9, et l'autre B doit les afficher.

shared.cpp

```
#include <unistd.h>
#include <iostream>
#include <sys/mman.h>
#include <sys/wait.h>

using namespace std;

class list {
      int data;
      list * next;
public :
      list (int data, list * next=nullptr) : data(data),next(next){}
      void add (int val) {
            if (next==nullptr)
                  next = new list(val);
            else
                  next->add(val);
      }
      void print (ostream & os) {
            os << data << "->" ;
            if (next != nullptr)
                  next->print(os);
      }
};

```

```
26 int main () {
27       list * head = (list *) mmap(0, sizeof(list) ,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|
             MAP_SHARED,-1,0);
28       if (fork()) {
29             // proc B : affiche
30             head->print(cout);
31       } else {
32             // proc A : construit
33             list * build = new list(0);
34             for (int i=1; i < 10 ; i++) {
35                   build->add(i);
36             }
37             *head = *build;
38       }
39       return 0;
40 }
```

Question 1. Ajoutez un mécanisme de synchronisation qui permette de garantir que la construction de la liste par A est finie avant que B ne tente l'affichage.

> un bête "wait(nullptr)" juste avant le `list->print(cout)` fait le job.

Question 2. Même avec cette synchronisation, le comportement du programme est étrange ; il affiche bien le premier "0->" de la liste mais le reste de la liste semble corrompu. Expliquez la nature du problème, et proposez une stratégie pour le résoudre. On ne demande pas le code de cette stratégie, simplement de l'expliquer en quelques phrases.

> On n'a que la tête de liste en mémoire partagée, les pointeurs "next" ne sont valides que dans l'espace d'adressage du processus A.
>
> La stratégie de résolution, c'est soit d'allouer la liste entière dans un segment partagé (pas très simple, il va falloir tout gérer soi même), soit plus simplement de *NE PAS UTILISER UNE LISTE* mais plutôt un bête tableau.

4 Sockets, Protocoles (4 points)

Notions testées : Communications Distantes.

Dans l'ensemble de cet exercice on considère un environnement Posix.

Question 1. On considère un serveur TCP constitué d'un seul processus, sans multi-threading. Comment supporter des connections de clients simultanées ?

> Avec un bon "select" bien senti, il faut attendre sur la socket d'accept et sur les sockets clients que read soit prêt.

Question 2. Qu'apporte l'utilisation d'un pool de thread dans le contexte d'un serveur TCP par rapport à l'instantiation d'un thread à chaque nouvelle connexion d'un client ? On identifiera au moins deux avantages.

> Performances accrues : le coût de création de thread n'est pas complétement négligeable.
>
> Résistance aux attaques : on va écrouler le serveur si on a un pic de connection et pas de pool.

Question 3. Que peut apporter le multi-threading pour un serveur UDP par rapport à un serveur sans concurrence ?

> Ben pas grand chose du point de vue du réseau, au mieux le traitement des requêtes sera plus rapide, mais il n'y a pas de session. Du coup pas besoin de "select" ou autre pour traiter le problème des attentes simultanées.

Question 4. On considère un serveur web type "google maps" qui offre un service permettant à partir d'une chaîne de texte (l'adresse) d'obtenir une carte de la zone.

a) Donnez une définition de la notion de "sérialisation". Instanciez votre définition en décrivant la sérialisation qui intervient dans cet exemple de "maps".

> Sérialiser = prendre un objet en mémoire et le mapper vers un champ contigü de bits le décrivant entièrement.
>
> Ici, le client va bien devoir sérialiser sa requête (adresse, préférences etc), et réciproquement la carte est un objet complexe qu'il faut sérialiser.

b) Quel est l'intérêt d'introduire un nouveau langage comme celui des fichiers `.proto` de Protobuf ?

> C'est un langage de description pure, et donc NEUTRE vis à vis des langages d'implém. On génère à partir de ces descriptions "pures" des primitives de (dé)sérialisation pour n'importe quel langage => interoperabilité.