TP Noté - ASR7 Programmation Concurrente

1h30

10 Mars 2020

Consignes générales

Le programme qui vous est demandé doit être écrit en `C++` sur les machines de la salle. Vous n'êtes pas autorisés à échanger d'information (notamment par le réseau), et vous n'avez pas accès à internet. Vous disposez normalement de suffisamment de connaissance avec ce qui a été fait en TP et TD (un rappel de C++ est fourni en fin de sujet).

Les machines de la salle vous permettent de vous connecter localement avec le login `moi` et le mot de passe `moi`. Vous sauvegarderez votre travail sur ce compte, et **vous le déposerez à la fin sous la forme d'un** *unique* **fichier** `mirepoix.cpp` **sur le serveur** `http://tpnote.tpr.univ-lyon1.fr`. Vous n'oublierez pas de mettre votre **NOM, PRÉNOM et numéro étudiant** en commentaire sur les premières ligne du fichier rendu.
Sans cela, votre travail ne pourra pas être considéré.

Le fichier rendu à la fin du TP doit impérativement compiler et s'exécuter sans lever d'erreur dans la plupart des cas. Le code écrit doit être clair et commenté. Il est préférable de traiter convenablement peu de questions plutôt que de fournir des codes non fonctionnels pour chaque partie.

Lisez bien tout le sujet avant de commencer. Les différentes parties du sujet peuvent être traitées indépendemment les unes des autres, sauf la partie I qui doit être traitée en premier. Vous pouvez donc commencer par la question avec laquelle vous vous sentez le plus à l'aise. Notez toutefois que leur difficulté est également croissante.

Prenez également le temps de lire le code du fichier `mirepoix.cpp`. Vous ne devez en aucun cas modifier le code des fonctions auxiliaires, ni celui de la fonction `main()` qui traite les arguments donnés en ligne de commande, et initialise les différents scénarios considérés. Toute modification de ces fonctions sera sanctionnée.

Le fichier se compile avec la ligne de commande suivante :

```
g++ -g -Wall -pthread -std=c++11 mirepoix.cpp -o mirepoix
```

Pour exécuter les différentes version du code, vous utiliserez :
— `./mirepoix` pour la version séquentielle du code
— `./mirepoix --para_V0` pour la première version parallèle du code
— `./mirepoix --para_V1` pour la seconde version parallèle du code
— `./mirepoix --famine` pour la version avec la gestion des famines
Vous avez également la possibilité d'avoir des messages de débug en ajoutant `--debug` à la ligne de commande.

Pour finir, vous pouvez contrôler le nombre de voitures en ajoutant `--nb-cars arg` à la ligne de commande. Si l'option n'est pas utilisée, le programme exécutera un test en dur avec 5 voitures.

Présentation du sujet – Pont de Mirepoix-sur-Tarn

A Mirepoix-sur-Tarn, au nord-est de Toulouse, un pont suspendu enjambe la rivière Tarn. En novembre 2019, un poids lourd dont le tonnage de 50 tonnes, ce qui est supérieur à la limite autorisée de 19 tonnes,

s'est engagé sur le pont, provoquant son effondrement. L'objectif de ce TP est de mettre en place un système de vérification pour ne pas dépasser le tonnage autorisé afin d'éviter l'effondrement du pont.

Le pont peut supporter au maximum un tonnage **cumulé**, c'est à dire la somme du tonnage des véhicules empruntant le pont, de 19 tonnes. Pour simplifier le problème, nous supposerons que les véhicules peuvent passer dans les deux sens sans que cela n'influence la résistance du pont. Sauf précision contraire, les voitures arrivent et s'engagent sur le pont dans n'importe quel ordre. Par exemple, il est possible qu'un camion de 10 tonnes soit en attente s'il y a déjà 10 voitures d'une tonne sur le pont, mais dans cette configuration une voiture d'une tonne peut s'engager même si elle est arrivée après le camion.

Comme dans le TD sur le pont de Miralonde, un mécanisme de barrière est mis en place pour contrôler les entrées et sorties sur le pont. A chaque fois qu'un véhicule souhaite s'engager, il fait appel à la fonction `entree()`, et fait appel à la fonction `sortie()` lorsqu'il a terminé de traverser le pont. Un compteur encapsulé dans l'objet `verif`, permet de comptabiliser le tonnage courant sur le pont. Dans le code fourni, les lignes `verif.entree_effective(masse, i);` et `verif.sortie_effective(masse, i);` permettent de vérifier que le tonnage ne dépasse pas le tonnage maximal autorisé afin que le pont ne cède pas (vous aurez un message d'erreur si cela arrive). L'appel à `entree_effective` doit être la dernière chose faite par `entree`, et celui à `sortie_effective` la première de `sortie`.

Le programme principal fourni permet de faire passer `nb_cars` voitures sur le pont. Chaque voiture à une masse, un temps d'attente avant d'entrer sur le pont et une durée de traversée du pont.

I V0 : Création des threads

Commencez par créer les threads qui correspondent aux différents véhicules voulant s'engager sur le pont. Chacun des véhicules attend un certain quota de temps avant de vouloir s'engager sur le pont. La durée de traversée du pont est également donnée. Complétez la fonction `loop_parallel` pour créer `nb_cars` threads qui exécutent chacun la fonction `car_thread`.

Testez votre programme avec l'option `--para_V0`. Qu'observez-vous par rapport au programme séquentiel ? Quel est le problème de l'approche ? Répondez en commentaire dans l'emplacement prévu à côté de la classe `GestionPont_V0`.

II V0 : Traversée du pont par un seul véhicule

Dans cette première version simplifiée, vous allez mettre en place un système afin d'éviter les accès concurrents au pont. Un véhicule sera alors assuré d'être le seul à emprunter le pont s'il entre en section critique, et sera autorisé à passer si son tonnage n'excède pas 19 tonnes (on suppose que c'est le cas pour tous les véhicules). Pour cette partie, complétez la classe `GestionPont_V0`.

Testez votre programme, toujours avec l'option `--para_V0`. Vous devriez maintenant avoir un temps total équivalent au temps séquentiel, deux véhicules ne pouvant s'engager simultanément sur le pont.

III V1 : Traversée du pont par plusieurs véhicules

Pour cette question, nous allons utiliser un moniteur de Hoare, implémenté par la classe `GestionPont_V1`.

Une voiture de masse `masse_voiture` ne pourra alors s'engager que lorsque que la condition `masse_voiture + masse_courante < 19` est respectée. Vous devrez utiliser un compteur `masse_courante` pour compter la masse totale des voitures actuellement engagées sur le pont. Ce compteur sera mis à jour dans les fonctions `entree()` et `sortie()` du moniteur.

Afin d'empêcher les véhicules de passer, vous mettrez en place un système à base de variables de condition pour que les processus associés s'endorment lorsque le pont est déjà trop chargé et se réveillent au bon moment. Complétez la classe `GestionPont_V1` pour la gestion des voitures.

IV Problème de famine

La solution précédemment proposée peut poser un problème de famine. Indiquez à l'emplacement prévu dans le fichier un exemple de famine. Pour vous aider, l'exemple pourra faire intervenir un véhicule lourd (15 tonnes) et des véhicules plus légers (5 tonnes). Donnez un exemple précis (masse des voitures, ordre d'arrivée, ..., et raison de la famine).

Proposez maintenant un mécanisme pour éviter le problème de famine. Vous pouvez par exemple mettre en place un ordonnancement de type FIFO (First In, First Out), où on impose aux véhicules en attente de passer dans l'ordre dans lequel ils sont arrivés. On peut réaliser ceci en maintenant à jour une `std::queue` contenant les numéros de voitures en attente.

Pour vous aider, voici un exemple d'utilisation de `std::queue` :

```
std::queue<int> f;
f.push(42); f.push(12);
cout << f.front(); // 42
cout << f.front(); // toujours 42
f.pop(); // Retire la première valeur
cout << f.front(); // 12
```

Pour cette partie, vous modifierez la classe `GestionPont_Famine`.

Barème

Voici enfin le barème prévu pour la notation. Il pourra être adapté en fonction de la réussite du sujet.

- **Clarté du code :** 2 points
- **Création des threads :** 2 points
- **Traversée par un seul véhicule :** 3 points
- **Traversée par plusieurs véhicules :** 7 points
- **Résolution du problème de famine :** 6 points
- **Un code ne compilant pas ne pourra avoir au dessus de 10.**

Rappels sur C++11 et les threads

Pour vous aider, voici un rappel de la syntaxe C++11 pour les threads :

```
// Création et attente de terminaison d'un thread :
int main () {
  // ...
  std::thread t(f, 42, std::ref(x));
  // ...
  t.join();
}

// Déclaration d'un mutex
std::mutex m;

// Verrouillage/déverrouillage d'un mutex :
m.lock();
// ...
m.unlock();

// Instantiation d'un verrou :
{
  std::unique_lock<std::mutex> l(m);
  // ...
}

// Opérations sur une variable de condition :
std::condition_variable c;
c.wait(l);  // l de type verrou (std::unique_lock par exemple)
c.notify_one(); // ou c.notify_all();

// Opérations sur une variable de condition :
std::condition_variable_any c;
c.wait(m); // m de type mutex
c.notify_all(); // ou c.notify_all();
```