TP Noté - LIFPCA Programmation Concurrente et Administration Système

1h30 (2h tiers temps)

22 Février 2024

Consignes générales

Le programme rendu doit être écrit en C++. Vous n'êtes pas autorisés à échanger d'information (ni par le réseau, ni via vos téléphones, ni par aucun autre moyen de communication). Vous n'êtes autorisés qu'à utiliser le système C5, dont la fenêtre doit rester en plein écran pendant toute la durée de l'épreuve. Un rappel de C++ est fourni avec ce sujet (les rappels s'afficheront après la première exécution du code).

Le fichier rendu à la fin du TP doit impérativement compiler et s'exécuter sans lever d'erreur dans la plupart des cas. Un code ne compilant pas aura un malus de 10 points. 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. Pensez à bien sauvegarder votre code régulièrement en cliquant sur l'enveloppe en haut.

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

Présentation du sujet

On s'intéresse à la modélisation d'un tunnel unidirectionnel qui peut être emprunté par plusieurs types de véhicules (piétons, vélos, voitures).

On modélise ce système avec un nombre quelconque de threads `vehicule` qui peuvent appeler les fonctions suivantes :

— `entreeTunnel(int type_v)` : quand un véhicule de type `type_v` veut entrer dans le tunnel.
— `sortieTunnel(int type_v)` : quand un véhicule de type `type_v` sort du tunnel.

Étant donné que les différents véhicules n'ont pas la même vitesse, certains règles doivent être respectées pour éviter les accidents. Un véhicule plus rapide (VOITURE > VELO > PIETON) ne peut pas entrer si des véhicules plus lents sont déjà présents dans le tunnel.

La fonction `vehicule(..., int type)` correspond à un véhicule d'un certain type qui essaie de traverser `NB_TOURS` fois le tunnel pendant son trajet. Le véhicule réussira donc à entrer au maximum `nb_entrees=NB_TOURS` fois.

Prise en main du code et lancement des threads

Pour l'instant, le programme ne fait rien et n'utilise même pas les threads.

Modifiez la fonction `lancement_multithread()` pour qu'elle lance `nb_vehicules` threads exécutant la fonction `vehicule()` avec les bons paramètres (numéro du thread, type de véhicule). La fonction `lancement_multithread()` doit faire en sorte que tous les threads s'exécutent en parallèle et attendre qu'ils terminent. Bien sûr, vous devez supprimer l'appel à `A_FAIRE`.

Relancez votre programme. Lorsqu'il vous pose la question « Entrez la version du code », répondez « `0` ». Pour l'instant, vous devriez obtenir un message, c'est normal, car pour l'instant les appels à `gestion_plateforme->...()` utilisent la classe `Tunnel_Sur`, qui n'est pas encore implémentée.

Une première solution naïve sans attente

Implémentez la classe `Tunnel_Sur`. Cette classe doit agir comme un moniteur de Hoare. Elle doit s'assurer qu'un véhicule ne rentre dans le tunnel que s'il y est autorisé. Un piéton peut tout le temps rentrer (il ne rattrapera personne), un vélo ne peut rentrer que s'il n'y a pas de piéton (il risquerait de les rattraper et de causer un accident) et une voiture ne peut rentrer que s'il n'y a que des voitures (ou tunnel vide). Dans cette approche naïve, un véhicule qui ne peut pas entrer immédiatement ne rentrera jamais (pour le tour courant).

Les appels aux fonctions `entreeEffective()` et `sortieEffective()` s'occupent de vérifier que les choses sont faites correctement. Il est interdit de supprimer ces appels de fonctions. Vous pouvez voir que l'appel à `entreeEffective()` ne se fait que que lorsque le véhicule peut entrer, c'est-à-dire quand la fonction `entreeTunnel()` renvoie `true`.

Calcul du nombre total d'entrées

À la fin du programme, on souhaite connaître le nombre total d'entrées dans le tunnel, c'est-à-dire la somme des entrées pour chaque thread. Mettez en place un mécanisme *efficace* pour calculer `nb_total_entrees`.

Affichage

Vous devriez observer des problèmes d'affichage : d'où viennent-ils ? Répondez à cette question dans le commentaire prévu en fin du code source.

Mettez ensuite en place un mécanisme pour gérer ce problème dans votre code.

Version avec attente simple

Dans l'approche naïve, un véhicule qui ne peut pas entrer immédiatement ne rentrera jamais (pour son tour courant). De ce fait, dans la fonction `vehicule()` le nombre final d'entrées ne sera pas toujours égal à `NB_TOURS()` (et à la fin `nb_total_entrees` ne sera pas toujours égal à `nb_vehicules * NB_TOURS`).

On veut maintenant mettre en place un mécanisme pour que `entreeTunnel()` attende qu'il soit possible d'emprunter le tunnel.

Réexécutez maintenant votre programme avec la version `1` (avec attente simple), qui va utiliser la classe `Tunnel_Avec_Attente_Simple` pour vos tests. Implémentez la classe `Tunnel_Avec_Attente_Simple`. Elle doit

s'assurer qu'un véhicule attende tant qu'il ne peut pas entrer.

Version avec attente avancée

Dans la version précédente, vous n'avez peut-être pas fait attention à la manière de faire les `notify()` et sûrement reveillé des threads qui ne pourront toujours pas passer et devront se rendormir. Expliquez dans le commentaire prévu à la fin du code source en quoi cette approche est sous-optimale.

Réexécutez maintenant votre programme avec la version `2` (avec attente avancée), qui va utiliser la classe `Tunnel_Avec_Attente_Avancee` pour vos tests. Implémentez la classe `Tunnel_Avec_Attente_Avancee` pour limiter au maximum le réveil inutile de threads.

Utiliser la variable `attentes` pour mesurer le nombre de fois où un thread fait une attente passive dans les deux classes `Tunnel_Avec_Attente_Simple` et `Tunnel_Avec_Attente_Avancee`. L'objectif est de réduire ce nombre dans la version avancée (tout en conservant un fonctionnement correct, bien entendu).

Version sans famine

Repartez de la version avec attente "simple". Expliquez dans le commentaire prévu à la fin du code source comment un problème de famine peut se produire dans le cas de la version attente "simple".

En repartant de cette version avec "attente simple", mettez ensuite en place un mécanisme pour supprimer les cas de famine dans la classe `Tunnel_Sans_Famine` instanciée dans la version `3` du programme. Vous pouvez ajouter des variables globales et des fonctions auxiliaires pour gérer ce cas.

Barème

Voici enfin un barème indicatif pour la notation.
— Clarté du code (code clair et commenté) : 1 point
— Création des threads : 2 points
— Classe `Tunnel_Sur` : 2 points
— Calcul du nombre total d'entrées : 2 point
— Explication et correction du problème d'affichage : 2 point
— Classe `Tunnel_Avec_Attente_Simple` : 4 points
— Explication réduire nombre appels wait() : 2 points
— Classe `Tunnel_Avec_Attente_Avancee` : 2 points
— Classe `Tunnel_Sans_Famine` : 3 points
— Un code ne compilant pas aura un malus de 10 points.

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();
```