TP Noté - ASR7 Programmation Concurrente

1h30

20 octobre 2020

Consignes générales

Le programme qui vous est demandé doit être écrit en `C` ou `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.

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 `sujet.cpp` sur le serveur `tpnote.tpr.univ-lyon1.fr`**. Vous n'oublierez pas de mettre votre NOM, PRÉNOM et numéro étudiant en commentaire de la première 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. 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 `sujet.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. Il ne faut également ajouter aucune fonction mais uniquement compléter les fonctions fournies lorsque cela est indiqué dans le code.

Le fichier se compile avec la ligne de commande suivante :

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

Pour exécuter le programme dans sa version par défaut, il faut exécuter la commande : `./sujet`

Barème

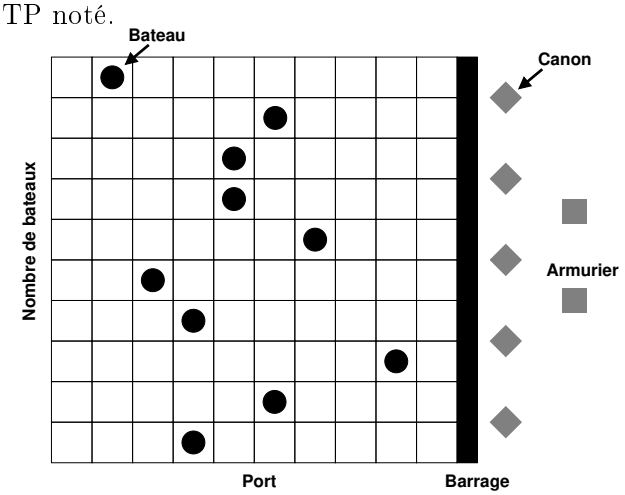
Voici enfin le barème prévu pour la notation. Il pourra être adapté en fonction de la réussite du sujet.

- **Clarté du code :** 2 points
- **À plusieurs, c'est plus efficace :** 3 points
- **À chacun son rôle :** 8 points
- **Attaque simultanée :** 7 points

- **Un code ne compilant pas ne pourra avoir au dessus de 10.**

Présentation du sujet

On souhaite simuler l'attaque d'un port par une armada ennemie. La figure ci-dessous illustre les forces en présence ainsi que le fonctionnement de la version initiale fournie dans le code de base que vous devrez faire évoluer au cours de ce TP noté.



Défense

Le port est défendu par `nb_canons` *canons* (configurable par `--nb-canons <int>`, 5 par défaut pour la version parallèle, 1 pour la version séquentielle). Pour défendre le port, chaque canon peut tirer un boulet, ce qui lui demande 0.25 seconde pour viser. Il a alors une chance sur cinq d'atteindre un des bateaux encore à flot. En cas de réussite, le cannonier incrémente le compteur global de bateaux coulés.

Avant de tirer, le canonnier doit récupérer un boulet à l'armurerie, ce qui prend 1 seconde (cette deuxième action pourra être déléguée à des *armuriers* si on donne au programme un nombre d'armuriers strictement positif via l'option `--nb-armuriers` et que vous avez écrit le code correspondant). Les canons tirent tant que le barrage porte tient bon et qu'il reste encore plus de 40% de bateaux dans le port.

Dans la version initiale, il n'y a qu'un seul canon qui effectue les deux actions. Pour activer la version parallèle (qu'il vous faudra écrire), il suffit d'ajouter `--par` à la ligne de commande. Pour déléguer l'approvisionnement en boulets, il faut aussi donner l'option `--nb-armuriers <int>`.

Attaque

Les différents *bateaux* de l'armada (configurable par `--nb-bateaux <int>`, 10 par défaut) démarrent chacun leur assaut depuis une distance différente du barrage. Ils progressent vers le barrage d'une case par seconde et s'y rejoignent pour déclencher l'attaque.

Lorsque 60% des bateaux sont arrivés au barrage, ils peuvent tirer une *salve* commune pour le détruire. S'il n'y a pas assez de bateaux, le barrage tiendra bon. Les bateaux déjà arrivés doivent alors attendre.

Dans la version initiale, chaque bateau attend que le précédent soit arrivé au barrage ou qu'il ait été coulé, avant de commencer sa propre progression (s'il est encore à flot). Il vous faudra écrire la version parallèle dans laquelle tous les bateaux démarrent simultanément puis tentent de détruire le barrage dès que possible, c'est-à-dire dès que 60% des bateaux sont arrivés.

Fin du jeu

La durée d'une partie est fixée à 10 secondes (configurable par `--timeout <int>`). Si l'armada arrive à tirer une salve et donc détruire le barrage avant la fin de partie, elle gagne. Dans le cas contraire, ou s'il ne reste plus assez de bateaux alors le port a été défendu et les canonniers remportent la partie.

I Défense du port

La défense la plus basique du port est assurée par un unique canonnier qui va lui-même chercher ses boulets... le port est donc très mal défendu. À vous de jouer pour améliorer cette défense !

À plusieurs, c'est plus efficace

Dans la fonction `defense_par()`, implémentez une première version parallèle de la défense du port. Vous devrez donc créer et détruire les threads correspondant aux canons. Dans cette version, chaque thread execute une fonction `canon_thread()` dont le code est similaire à celui de la version séquentielle. Chaque canonnier va lui-même chercher ses boulets, et met à jour le compteur global de bateaux coulés. `nb_canons` canons devraient suffire. *Une fois votre code écrit, testez le en ajoutant* `--par` *à la ligne de commande.*

À chacun son rôle

L'inconvénient majeur de cette première version, c'est que pendant qu'un canonnier va chercher unboulet, il n'est pas en train de tirer. Dans la fonction `defense_avec_armuriers()`, implémentez une seconde version de la défense du port. Dans celle-ci, les rôles sont bien identifiés :

— Des threads *canon_ v2* tirent et mettent à jour le compteur de bateaux coulés ;

— Des threads *armurier* apportent les boulets.

Une fois les différents threads lancés, vous devrez organiser l'interaction entre armuriers et canons. Vous devrez pour cela avoir défini **une** *pile* **commune, contenant au plus 4 boulets par canon**.

Quand il amène un boulet, un armurier le dépose sur la pile et va en chercher un autre. Le canonier récupére un boulet sur la pile pour tirer. Attention, si la pile est pleine, un armurier ne peut pas y déposer de nouveau boulet. De même, impossible pour un canon de tirer sans boulet. *Une fois votre code écrit, testez le en ajoutant* `--nb-armuriers 2` *à la ligne de commande.*

II L'armada veut forcer le barrage

Dans la version initiale, une stratégie maladroite est adoptée. Les bateaux avancent un par un et peuvent être coulés en attendant de démarrer. En exécutant plusieurs fois le programme (sans aucun argument), vous constaterez que ce type d'attaque a peu de chance de réussir (vous pouvez tester avec une durée plus longue, `--timeout 100` pour vous en convaincre).

Attaque simultanée

Dans la fonction `attaque_par()`, implémentez une version parallèle de l'attaque du port. Chaque bateau est un thread qui avance à son propre rythme. Le déclenchement de la salve simultanée agit comme une barrière de synchronisation entre les bateaux : les premiers arrivés doivent attendre qu'un nombre suffisant soit atteint (dans le code, ce nombre est défini à 60% du nombre initial de bateaux). Vous compléterez et utiliserez pour cela la classe `Salve`, dont le squelette est fourni. *Une fois votre code écrit, testez-le en ajoutant* `--par` *à la ligne de commande.*

NB : Attention à ne pas oublier que dès que la salve est tirée, l'attaque est un succès.

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 :
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 :
c.wait(m); // m de type mutex
c.notify_all(); // ou c.notify_one();
```