TP Noté - ASR7 Programmation Concurrente

1h30

25 octobre 2018

I Consignes générales

Les programmes qui vous sont demandés doivent être écrits 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, vous avez à disposition les corrections de ce qui a été fait en TP et TD (voir plus loin), et le sujet dispose de quelques pages avec de la documentation sur des fonctions qui pourraient être utiles.

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 `battle.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. Vous ne devez pas éditer les fonctions déjà écrites. **Lorsque cela est indiqué dans `battle.cpp`**, vous devez répondre aux questions suivantes, avant de commencer l'écriture de code **en éditant les commentaires prévus à cet effet**.

Quelles sont les données qui peuvent être :
- **lues par plusieurs threads ?**
- **lues par un thread et modifiées par un autre ?**
- **modifiées par plusieurs threads ?**
- **en déduire le type de problème de synchronisation rencontré**

Voici ci-dessous le barême qui est 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
- **Prudence n'est pas mère de succès :** 2 points
- **Attaque simultanée :** 5 points

- **Bonus Simple :** 1 point
- **Super Bonus :** 20 points

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

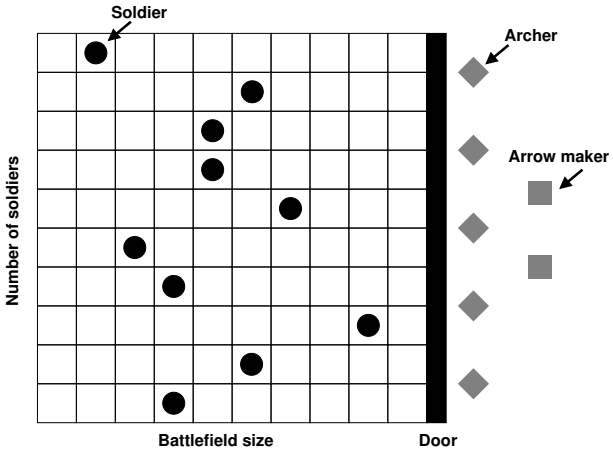
II Présentation du sujet

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

Prenez connaissance du fichier `battle.cpp`. Il est en particulier conseillé de lire (mais sans le modifier) le code des défenseurs et des attaquants.

Il ne faut absolument pas modifier la fonction `main()` qui traite les arguments donnés en ligne de commande, et ne fait que lancer les différents modes d'attaque et de défense considérés. Le fichier se compile avec la ligne de commande : `g++ -g -Wall -pthread -std=c++11 battle.cpp -o battle`

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



Défense

Le château est défendu par `number_of_archers` *archers* (configurable par `--nb-archers <int>`, 5 par défaut pour la version parallèle, 1 pour la version séquentielle). Pour défendre le château, chaque archer peut tirer une flèche, ce qui lui demande 0.25 seconde pour viser. Il a alors une chance sur cinq d'atteindre un des soldats encore vivants. En cas de réussite, l'archer incrémente le compteur global d'ennemis abattus.

Avant de tirer, l'archer doit construire la flèche, ce qui prend 1 seconde. Cette deuxième action peut être déléguée à des *fabriquants de flèches*. Les archers tirent tant que la porte du château tient bon et qu'il reste encore plus de 40% de soldats ennemis en vie.

Dans la version séquentielle activée par défaut, il n'y a qu'un seul archer 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 la fabrication de flèches, il faut aussi donner l'option `--arrow-makers <int>`.

Attaque

Les différents *soldats* de l'armée d'attaque (configurable par `--nb-soldiers <int>`, 10 par défaut) démarrent chacun leur assaut depuis une position différente. Ils progressent vers la porte d'une case par seconde.

Lorsque 60% des soldats sont arrivés à la porte du château, ils peuvent utiliser un *bélier* pour l'enfoncer. Tant que le nombre suffisant de soldats n'est pas atteint, cette action est impossible. Les soldats déjà arrivés doivent alors attendre.

Dans la version séquentielle activée par défaut, chaque soldat doit attendre que le précédent soit arrivé vivant à la porte du château, ou avoir été tué avant de commencer sa propre progression (s'il est lui-même encore en vie). Il vous faudra écrire la version parallèle dans laquelle tous les soldats attaquent simultanément.

Fin du jeu

Par défaut, la durée d'une partie (configurable par `--timeout <int>`) est fixée à 10 secondes. Si les soldats arrivent à utiliser le bélier et donc enfoncer la porte avant la fin de partie, les soldats gagnent. Dans le cas contraire, ou s'il ne reste plus assez de soldats pour actionner le bélier alors ce sont les archers qui sont parvenus à défendre le château et remportent la partie.

III Les archers défendent le château

La version la plus basique de la défense du château est assurée par un unique archer qui fabrique lui-même ses flèches... Non seulement, ce pauvre archer se sent bien seul face à l'ennemi, mais le château est aussi très mal défendu. À vous de jouer pour améliorer cette défense !

À plusieurs, c'est plus efficace

Dans la fonction `parallel_defense()`, implémentez une première version parallèle de la défense du château. Vous devrez donc créer et détruire les threads correspondant aux archers. Dans cette version, chaque thread execute une fonction `archer_thread()` dont le code est similaire à celui de la version séquentielle de l'attaque. Chaque archer construit donc lui-même les flèches qu'il tire, et met à jour le compteur global d'ennemis abattus. `number_of_archers` archers devraient suffire à affronter l'ennemi.

NB : 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 implémentation, c'est que pendant qu'un archer construit sa flèche, il n'est pas en train de tirer et laisse donc ses ennemis approcher.

Dans la fonction `parallel_defense_with_arrow_makers()`, implémentez une seconde version parallèle de la défense du château. Dans celle-ci, les rôles sont bien identifiés :

— Les threads *archer_ with_ maker* tirent des flèches et mettent à jour le compteur d'ennemis abattus ;

— Les threads *arrow_ maker* fabriquent les flèches.

Dans cette fonction, une fois les différents threads lancés, vous devrez organiser l'interaction entre les fabriquants de flèches et les archers. Vous devrez pour cela définir **un** *carquois* **commun, contenant au plus 4 flèches par archer**.

Dès qu'il a terminé une flèche, un fabriquant va la déposer dans le carquois. Un archer devra quant à lui récupérer une flèche dans le carquois pour tirer sur l'ennemi. Attention, si le carquois est plein, un fabriquant ne peut pas y déposer de nouvelle flèche. De même, impossible pour un archer de tirer si le carquois est vide.

NB : Une fois votre code écrit, testez le en ajoutant `--arrow-makers` *à la ligne de commande.*

IV Les soldats veulent enfoncer la porte du château

Dans la version initiale de l'attaque du chateau, une stratégie très prudente est adoptée. Les soldats avancent un par un, les autres restant cachés en attendant. En exécutant plusieurs fois le programme (sans aucun argument), vous constaterez que ce type d'attaque a peu de chance de réussir.

Prudence n'est pas mère de succès

Indiquez dans le commentaire précédent la fonction `sequential_attack()` :

— Pourquoi cette version ne peut pas mener à la victoire des attaquants ?

— Quelle devrait être selon vous la durée minimale d'une partie pour que les soldats aient une chance d'enfoncer la porte du château ?

NB : Pour tester cette version, ajoutez `--seq` *à la ligne de commande.*

Attaque simultanée

Dans la fonction `parallel_attack()`, implémentez une version parallèle de l'attaque-défense du château. Chaque soldat est un thread qui avance à son propre rythme. L'utilisation du bélier agit comme une barrière de synchronisation pour les soldats : les premiers arrivés doivent attendre qu'un nombre suffisant de soldats pour manipuler le bélier soit atteint (dans le code, ce nombre est défini à 60% du nombre initial de soldats). Vous compléterez et utiliserez pour cela la classe `BatteringRam` (« bélier » dans la langue de Shakespeare), dont le squelette est fourni.

NB : Une fois votre code écrit, testez-le en ajoutant `--par` *à la ligne de commande.*

NB : Attention à ne pas oublier que dès que le bélier est utilisé, l'attaque est un succès.

V Questions Bonus

Bonus simple : Quelle est la combinaison `number_of_archers / number_of_arrow_makers` qui permet aux archers de tirer en continu ? Indiquez votre réponse en commentaire à l'endroit indiqué dans le code.

Super bonus : Écrivez une interface graphique complète pour ce jeu.

```
std::condition_variable

void notify_all() noexcept;
(since C++11)


Unblocks all threads currently waiting for *this.
Parameters

(none)
Return value

(none)
Notes

The effects of notify_one()/notify_all() and each of the three atomic parts of wait()/wait_for()/

The notifying thread does not need to hold the lock on the same mutex as the one held by the wait
Example
Run this code

#include <iostream>
#include <condition_variable>
#include <thread>
#include <chrono>

std::condition_variable cv;
std::mutex cv_m; // This mutex is used for three purposes:
                 // 1) to synchronize accesses to i
                 // 2) to synchronize accesses to std::cerr
                 // 3) for the condition variable cv
int i = 0;

void waits()
{
    std::unique_lock<std::mutex> lk(cv_m);
    std::cerr << "Waiting... \n";
    cv.wait(lk, []{return i == 1;});
    std::cerr << "...finished waiting. i == 1\n";
}

void signals()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        std::lock_guard<std::mutex> lk(cv_m);
        std::cerr << "Notifying...\n";
    }
    cv.notify_all();

    std::this_thread::sleep_for(std::chrono::seconds(1));

    {
```

```
        std::lock_guard<std::mutex> lk(cv_m);
        i = 1;
        std::cerr << "Notifying again...\n";
    }
    cv.notify_all();
}

int main()
{
    std::thread t1(waits), t2(waits), t3(waits), t4(signals);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}

Possible output:

Waiting...
Waiting...
Waiting...
Notifying...
Notifying again...
...finished waiting. i == 1
...finished waiting. i == 1
...finished waiting. i == 1
```

Thread support library

std::condition_variable

void notify_one() noexcept;
(since C++11)

If any threads are waiting on *this, calling notify_one unblocks one of the waiting threads.
Parameters

(none)
Return value

(none)
Notes

The effects of notify_one()/notify_all() and each of the three atomic parts of wait()/wait_for()/

The notifying thread does not need to hold the lock on the same mutex as the one held by the wait

Notifying while under the lock may nevertheless be necessary when precise scheduling of events is
Example
Run this code

```
#include <iostream>
#include <condition_variable>
#include <thread>
#include <chrono>

std::condition_variable cv;
std::mutex cv_m;
int i = 0;
bool done = false;

void waits()
{
    std::unique_lock<std::mutex> lk(cv_m);
    std::cout << "Waiting... \n";
    cv.wait(lk, []{return i == 1;});
    std::cout << "...finished waiting. i == 1\n";
    done = true;
}

void signals()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Notifying falsely...\n";
    cv.notify_one(); // waiting thread is notified with i == 0.
                     // cv.wait wakes up, checks i, and goes back to waiting

    std::unique_lock<std::mutex> lk(cv_m);
```

```
    i = 1;
    while (!done)
    {
        std::cout << "Notifying true change...\n";
        lk.unlock();
        cv.notify_one(); // waiting thread is notified with i == 1, cv.wait returns
        std::this_thread::sleep_for(std::chrono::seconds(1));
        lk.lock();
    }
}

int main()
{
    std::thread t1(waits), t2(signals);
    t1.join();
    t2.join();
}
```

Possible output:

```
Waiting...
Notifying falsely...
Notifying true change...
...finished waiting. i == 1
```