Labo 4
Simulateur de train

Durée du travail :	**Du 08.10.2023 au 28.11.2023**
Auteurs :	***Kilian Demont*** **Calvin Graf**
Enseignant :	**Florian Vaussard**
Cours :	**PCO**
Lieu de travail :	**HEIG-VD, Yverdon-les-Bains**

Table des matières

1. Introduction 2
2. Développement 3
2.1 Chemin réalisé 3
2.2 Architecture du projet 4
2.3 Choix 5
2.4 Gestion de la concurrence 5
2.5 Tests 6
2.6 Amélioration 6
3. Conclusion 7

1. Introduction

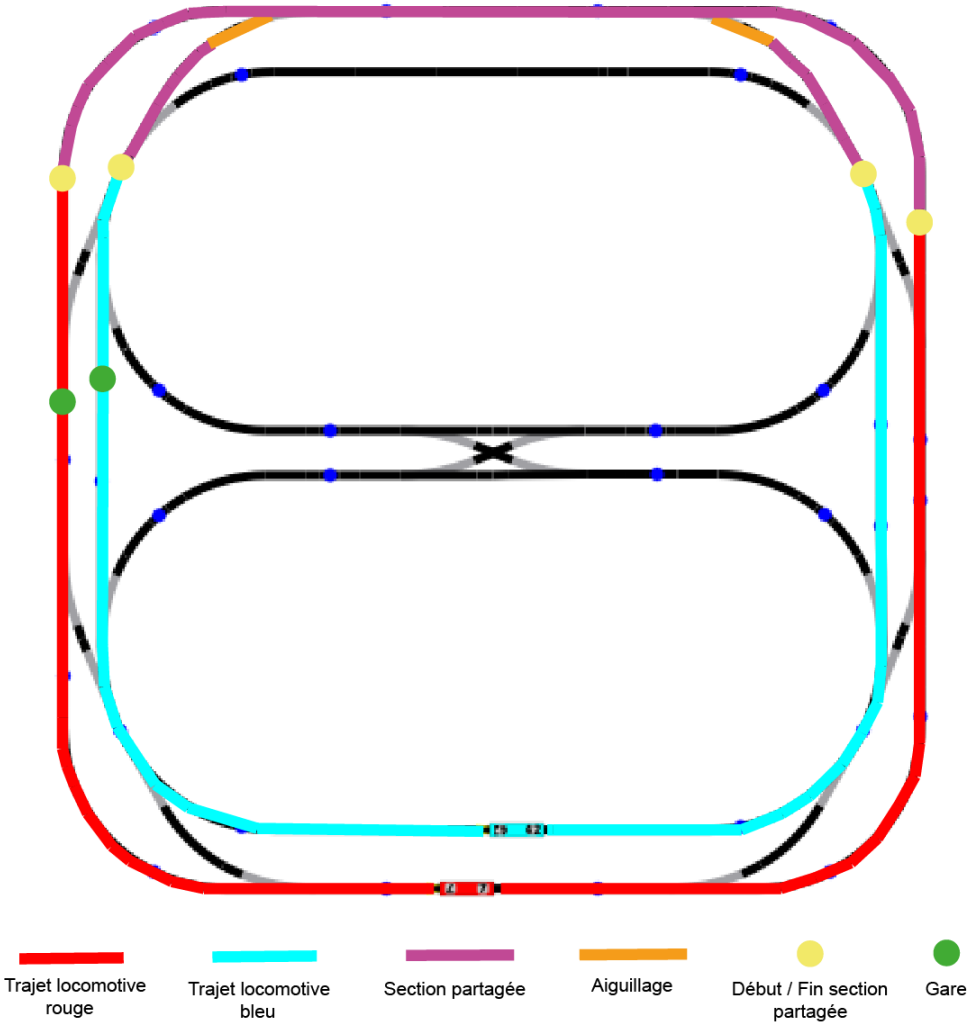
Pour ce laboratoire, nous devons implémenter en C++ un simulateur de locomotives qui suivra un chemin que nous avons préalablement choisi. Les deux chemins devront avoir une section partagée à laquelle ils accèderont à tour de rôle afin de ne pas provoquer d'accident. Le chemin doit être cyclique et une fois un cycle réalisé, le tour d'accès à la section partagée sera inversé. Les deux trains doivent s'attendre à la gare puis on ajoute un délai de 5 secondes afin de laisser les passagers monter et descendre du train.

L'objectif de cette simulation est d'approfondir notre compréhension pratique des concepts de la programmation concurrente tels que les mutex, sémaphore, threads, etc. Cela nous confrontera également au besoin d'avoir une stratégie pour que les trains ne rentrent pas en collision et respectent le tour de chacun. L'UI est déjà implémenté et nous ne devons pas y toucher.

2. Développement

2.1 Chemin réalisé

Nous avons réalisé un schéma du parcours que vont suivre nos locomotives. L'image est légendée afin de mieux la comprendre. Bien sûr, ce chemin peut être modifiée plutôt facilement tant que le nouveau chemin est cohérant et ne provoque pas d'accident.



2.2 Architecture du projet

Le projet est structuré en plusieurs classes, nous allons détailler uniquement les fichiers que nous avons modifié qui sont les suivants :

locomotivebehavior.h : Nous avons modifié le constructeur pour lui ajouter plusieurs variables liées à la locomotive.

- (unsigned int) nbStation : Le numéro de la gare où la locomotive doit s'arrêter
- (unsigned int) nbStartSharedSection : Le numéro qui indique que la locomotive est arrivée au début de la section partagée
- (unsigned int) nbEndSharedSection : Le numéro qui indique que la locomotive est arrivée à la fin de la section partagée
- (bool) switchSharedSection: Indique si l'aiguillage doit être modifiée ou non
- (unsigned int) trainSwitch[2][2] : Tableau 2D qui contient les aiguillages qui devront être modifiée ainsi que leurs états actuels. Nous avons fait un tableau de 2 car notre chemin contient 2 aiguillages.

locomotivebehavior.cpp : Permet de définir le comportement des locomotives durant le trajet grâce à la méthode « run() ». Les étapes sont les suivantes :

1. Arrête la locomotive lorsqu'elle arrive en gare
   1.1. Attendre que les deux locomotives arrivent en gare + délai de 5 secondes puis redémarre automatiquement
   1.2. Donne la priorité à la locomotive arrivée en dernier
2. Attend le contact avec le point du début de la section partagée
   2.1. Si c'est son tour, accéder à la section partagée
   2.2. Sinon, attendre que l'autre locomotive ait passée la section partagée
3. Redirige l'aiguillage si nécessaire
4. Attend le contact avec le point de la fin de la section partagée
   4.1. Donne la priorité à l'autre locomotive
5. Redirige l'aiguillage si nécessaire

synchro.h : Ce fichier est nécessaire pour gérer les accès concurrents aux ressources. Il implémente la classe `Synchro`, qui représente la section partagée du système ferroviaire. Cette classe utilise des sémaphores pour synchroniser l'accès aux ressources partagées. Une description plus détaillée de cette classe se trouve au point 2.4.

cppmain.cpp : Ce fichier crée deux locomotives, initialise les aiguillages, positionne les locomotives, et lance les threads de comportement des locomotives pour simuler leur déplacement synchronisé sur une maquette de chemin de fer. Nous avons ajouté la gestion de l'arrêt d'urgence, la fonction emergency_stop() permet ainsi de forcer l'arrêt des locomotives et de les empêcher de redémarrer.

2.3 Choix

Nous passons un tableau en deux dimensions nommé « trainSwitch » pour gérer le comportement des aiguillages de la section partagée. Ce dernier peut facilement être modifié dans le cas où l'on souhaiterait changer de parcours et ainsi avoir plus ou moins de 2 aiguillages. Dans ce cas, il faudra également modifier la 1ère boucle "for" du constructeur pour mettre le nombre d'aiguillages que l'on souhaite ainsi que les deux boucles dans la fonction run().

Le paramètre temps_alim utilisé pour diriger les aiguillages n'est pas utiliser (mis systématiquement à zéro) puisque nous utilisons uniquement la simulation (sur laquelle ce paramètre n'a aucun effet) et non la maquette physique.

2.4 Gestion de la concurrence

Les sémaphores sont utilisés pour gérer la concurrence et la synchronisation entre les threads représentant les locomotives. Ils permettent de créer des sections critiques dans le code où l'accès à certaines ressources partagées (gare, section partagée) doit être contrôlé. Cela garantit que les locomotives se synchronisent correctement, évitant les problèmes de concurrences et assurant un accès exclusif aux ressources partagées. Voici, en détail, comment ils sont utilisés :

- waitingStation : C'est un sémaphore qui contrôle que les deux locomotives soient bien arrivées à la gare avant de redémarrer. Lorsqu'une locomotive arrive, elle indique à l'aide du booléen "trainAtStation" qu'elle attend en gare et acquire le sémaphore pour bloquer le thread jusqu'à ce que l'autre locomotive arrive et le release afin de la débloquer.

  Lorsque l'autre locomotive arrive, elle remet le booléen à false, donne la priorité à la locomotive arrivée en dernier pour le tour suivant, la débloque et lance le délai de 5 secondes avant de redémarrer.

- waitingSharedSection : Ce sémaphore est utilisé pour contrôler l'accès à la section partagée entre les deux locomotives. Lorsqu'une locomotive veut accéder à la section partagée elle vérifie si elle a la priorité ou non. Si oui, elle passe et continue son chemin. Si non, elle acquire le sémaphore pour la bloquer jusqu'à ce que l'autre locomotive ait fini de traverser la section partagée et libère la locomotive qui attend avec un release. Elle pourra alors à son tour traverser la section partagée.

- mutex : Il s'agit d'un sémaphore binaire (sémaphore initialisé à 1) utilisé comme mutex pour protéger l'accès à certaines sections critiques du code où des variables partagées sont modifiées. Par exemple, il est utilisé pour protéger l'accès aux variables trainAtStation, trainAtSharedSection, et priorityTurn.

- trainAtStation et trainAtSharedSection : Ces variables booléennes indiquent si une locomotive attend actuellement à la gare ou au début de la section partagée, respectivement. Leur utilisation est protégée par le sémaphore "mutex".
- priorityTurn : C'est une variable entière qui indique le numéro de la locomotive qui a la priorité pour accéder à la section partagée. Elle est également protégée par le sémaphore "mutex". À chaque cycle effectué, la priorité va s'inverser.

2.5 Tests

Nous avons effectué une batterie de tests afin de vérifier le bon fonctionnement de notre programme. Voici toutes les vérifications que nous avons réalisées :

Système

- Les locomotives suivent le parcours prédéfini : OK
- L'arrêt d'urgence force les locomotives à s'arrêter et à ne jamais redémarrer même si on leur demande : OK

Gare

- Les locomotives s'y arrêtent : OK
- Les locomotives s'attendent : OK
- Les locomotives attendent 5s avant de repartir lorsque la seconde locomotive est arrivée : OK

Section partagée

- Les locomotives n'y accèdent pas simultanément : OK
- L'ordre de priorité est respecté : OK
- La priorité va à la dernière locomotive arriver en gare : OK

2.6 Amélioration

Le laboratoire est basé sur le travail de bachelor d'un étudiant de la HEIG-VS d'il y a quelques années. La structure du code fournie pourrait être divisée autrement afin de rendre le tout plus clair. Il est difficile à l'heure actuelle de bien comprendre certains aspects de ce qui a été implémenter sans se confronter suffisamment longuement au code.

Il serait intéressant de rendre ce code plus générique. Pouvoir ajouter plusieurs chemins avec plusieurs sections partagées facilement. Ce n'est pas si compliqué à réaliser et ajoute un réel plus. Avoir la possibilité d'ajouter plusieurs locomotives seraient également envisageable mais nécessiterait de revoir toute la stratégie actuelle puisque celle-ci a été conçu pour 2 locomotives.

3. Conclusion

En conclusion ce laboratoire nous a permis d'exercer l'utilisation des sémaphores dans une simulation explicitant bien l'importance d'une bonne synchronisation. Si les trains ne sont pas correctement synchronisés alors cela pourrait entraîner des conséquences terribles comme la collision entre deux trains. Les tests ont mis en évidence le bon fonctionnement du système et de la synchronisation.

Travailler avec une interface graphique est également un réel plus pour notre formation. Bien que nous ne devions pas modifier le code la concernant, cela nous permet de nous faire découvrir de nouvelles fonctionnalités.