HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Yann Thoma
Florian Vaussard
Alberto Dassatti

Laboratoire de Programmation Concurrente
semestre automne 2025

Gestion de ressources partagées

Temps à disposition: 6 périodes (trois séances de laboratoire)

Récupération du laboratoire : `retrieve_lab pco25 lab04`

1 Objectifs

- Gérer des situations de concurrence, gestion de ressources partagées et compétitions à l'aide de **sémaphores** (PcoSemaphore).

2 Enoncé

Grâce au simulateur qui vous est fourni, implémentez un programme en C++ qui réalise la gestion et le contrôle de deux locomotives. Chaque locomotive est exécutée par un thread et part depuis un point prédéterminé. Vous êtes libres de choisir le tracé des locomotives, tant qu'elles suivent un parcours cyclique pour revenir à leur point de départ. De plus, les deux tracés devront comporter au moins un tronçon commun. Ce tronçon commun ne peut être emprunté que par une seule locomotive à la fois. Si une locomotive est déjà présente, la deuxième doit s'arrêter et attendre que le tronçon soit libre. Attention, une locomotive ne doit pas s'arrêter pour redémarrer immédiatement, afin d'éviter de donner des hauts le cœur aux voyageurs. Il faudra également prévoir que les locomotives puissent changer de sens. A vous de décider d'un endroit où elles le feront, de temps en temps, et en dehors du tronçon partagé.

Vous pouvez choisir entre deux maquettes différentes (nommées A et B) au moyen de la fonction `selection_maquette()` dans le fichier `cppmain.c`. Choisissez une maquette et déterminez le parcours que chaque locomotive devra emprunter en vous inspirant du code existant.

Afin de faire rentrer ou sortir les locomotives du tronçon commun (section partagée) il vous faudra commander les aiguillages de manière appropriée.

Sur chaque maquette, des points de contact vous permettent de situer les locomotives sur leur parcours. La fonction `attendre_contact{ContactID}` devrait vous être utile. Celle-ci bloque le code de l'appelant tant qu'une locomotive n'est pas passée sur le point de contact spécifié.

L'accès au tronçon commun est défini de la manière suivante:

- Si le tronçon est libre, la locomotive pourra y accéder ;
- Si le tronçon est occupé par l'autre locomotive, alors la nouvelle arrivante doit s'arrêter ;
- Lorsqu'une locomotive quitte le tronçon, si l'autre est en attente, nous avons deux cas:
  1. Si les deux locomotives vont dans une direction opposée, alors celle en attente doit pouvoir immédiatement reprendre son chemin.

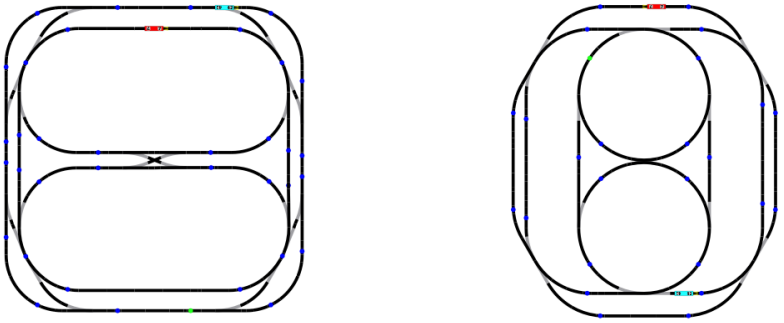

Figure 1: Maquettes A et B

2. Si les deux locomotives vont dans le même sens, alors celle qui quitte le tronçon doit avancer un peu avant de laisser la deuxième accéder au tronçon. Ceci peut être fait en attendant un contact suivant, puis en appelant la méthode `release()`.

3 Travail à faire

Le code qui vous est fourni est décomposé de manière à avoir une classe qui fournit la synchronisation. Celle-ci représente le tronçon commun et devra dériver de l'interface suivante:

```
class SharedSectionInterface
{
public:
    enum class Direction {D1, D2};

    virtual void access(Locomotive& loco, Direction d) = 0;
    virtual void leave(Locomotive& loco, Direction d) = 0;
    virtual void release(Locomotive& loco) = 0;
    virtual void stopAll() = 0;

    [[nodiscard]]
    virtual int nbErrors() = 0;
};
```

Les méthodes à implémenter dans `sharedsection.h` sont:

- `access()`: Appelée lorsqu'une locomotive souhaite accéder à la section partagée. Bloque si la section n'est pas libre.
- `leave()`: Appelée lorsque la locomotive a quitté physiquement la section.
- `release()`: Appelée pour libérer la section après un `leave()`, autorisant éventuellement une autre locomotive à entrer.
- `stopAll()`: Stoppe toutes les locomotives qui attendent d'accéder à la section partagée (utilisé pour un arrêt d'urgence, voir ci-dessous).
- `nbErrors()`: Retourne le nombre d'erreurs détectées (voir ci-dessous).

En théorie nous partons du principe que les appels seront faits dans un ordre pertinent, c'est-à-dire qu'on ne devrait pas appeler `leave()` s'il n'y a pas eu un `access()` appelé avant, et notamment avec la même locomotive et la même direction. De même `release()` ne devrait être appelé qu'au moment opportun (donc pas sans que la locomotive n'ait fait un passage dans le tronçon et que la condition pour le release ne soit bonne).

Il est impératif de respecter cette interface, car les tests du code l'utiliseront tel quel.

Évidemment, pour l'implémentation dans le simulateur nous pouvons partir du principe que les appels seront pertinents, mais afin de rendre le code le plus robuste possible il vous sera nécessaire

de détecter les erreurs d'appels potentiels. La détection devrait simplement incrémenter un compteur d'erreurs, qui est ensuite renvoyé par `nbErrors()`. Pour vérifier ceci nous vous fournissons quelques tests que vous pouvez lancer. Attention, il se pourrait que nous ayons quelques tests gardés au chaud pour une validation finale, soyez donc inventifs sur les erreurs à détecter. Aussi, il vous est fortement suggéré d'ajouter des tests que vous jugeriez pertinents.

Votre programme devra avoir un arrêt d'urgence actionné par l'interface graphique. Implémentez la fonction `emergency_stop()` de `cppmain.cpp` pour arrêter toutes les locomotives immédiatement, et ce SANS appeler `mettre_maquette_hors_service()`. Il s'agira également de garantir qu'une locomotive ne redémarre pas de manière intempestive. L'interface de synchronisation offre une méthode `stopAll()` au cas où vous voudriez y faire quelque chose en cas d'arrêt d'urgence.

4 Simulateur

Le code ainsi que la documentation du simulateur de maquettes sont fournis. Voici quelques indications indispensables sur le simulateur:

- Lors de votre première compilation, il faut aller dans le dossier de build et lancer la commande `make install` avant de pouvoir lancer l'application. Ceci peut aussi être fait depuis QtCreator (cf. documentation de QTrainSim).
- Il est possible de modifier certains paramètres dans le fichier `cppmain.cpp`, tels que:
  - La position et la vitesse initiales de chaque locomotive
  - La maquette utilisée (A ou B)
  - La position de départ des aiguillages
- Le contrôle du trajet emprunté par chaque locomotive s'effectue au moyen de points de contacts et d'aiguillages. On peut afficher ou masquer le numéro des points de contact et des aiguillages depuis le menu *View* de l'interface du simulateur.
- Il est possible de mettre en "pause" chaque locomotive depuis l'interface du simulateur, ce qui permet de tester certains scénarios plus facilement. De même, il est possible de changer un aiguillage en cliquant dessus.

5 Remarques

Quelques dernières remarques:

- Ne pas modifier les fichiers `locomotive.h/cpp`, `launchable.h` et `synchrointerface.h`. Vous pouvez créer de nouveaux fichiers ou classes pour gérer vos parcours au besoin.
- La description de l'implémentation, ses différentes étapes, la manière dont vous avez vérifié son fonctionnement et toute autre information pertinente doivent figurer dans un petit rapport rendu avec le code. Celui-ci se nommera `rapport.pdf` et se trouvera au même niveau que le script `pco_rendu.sh`, ce dernier servant à générer l'archive à rendre sur Cyberlearn.
- Documentez également les tests que vous auriez écrits en expliquant dans le rapport ce que vous avez mis en place.
- Inspirez-vous du barème de correction pour savoir là où il faut mettre votre effort.
- Ce travail est à réaliser en groupe de 2 personnes.

6 Barème de correction

Conception et conformité à l'énoncé	40%
Tests	15%
Documentation (rapport, etc.)	40%
Qualité du code (codage, commentaires, etc.)	5%