HE VD HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Chargés de cours : Florian Vaussard, Rick Wertenbroek

Laboratoire de Programmation Concurrente

semestre automne 2022

Gestion de ressources partagées

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

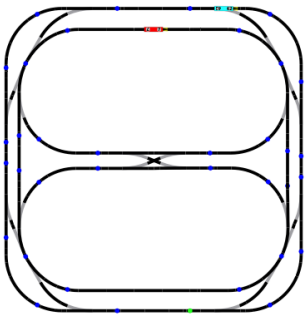
Récupération du laboratoire : `retrieve_lab pco22b lab4`

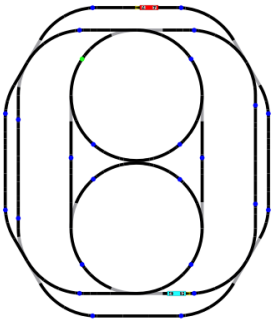
1 Objectifs

— Gérer des situations de concurrence, gestion de ressources partagées et compétitions à l'aide de sémpahores.

2 Enoncé

Dans un simulateur de maquettes de trains Märklin (maquettes en B08, malheureusement en panne cette année), implémentez un programme en C++ avec utilisation de sémaphores qui réalise la gestion et le contrôle de deux locomotives. Les locomotives suivent des tracés circulaires. Vous êtes libres de choisir le tracé des locomotives, toutefois il y a une contrainte à respecter : il doit y avoir au moins un tronçon commun aux deux parcours (une section partagée).





Deux programmes vous sont demandés, ils seront testés les deux, mais seul le dernier sera corrigé.

2.1 Programme 1

Les locomotives partent d'un point particulier de leur tracé. Chaque fois que les locomotives réalisent N tours complets (N compris entre 1 et 10), elles inversent leur direction et repartent pour N tours dans le sens contraire, et ceci de manière infinie. N est une valeur propre à chaque locomotive. Les locomotives formulent une requête pour obtenir un tronçon partagé et, si celui-ci n'est pas disponible, la locomotive s'arrête avant le tronçon demandé. Si au contraire le tronçon demandé est immédiatement disponible, les locomotives ne devront pas s'arrêter (pas même une micro-seconde).

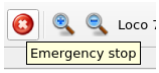
Le code qui vous est fourni est décomposé de manière à avoir une classe qui fournit la synchronisation. Celle-ci représente le tronçon commun et devra dériver de l'interface suivante :

```
class SharedSectionInterface
{
public:
```

```
    virtual void access(Locomotive& loco) = 0;
    virtual void leave(Locomotive& loco) = 0;
};
```

De plus, les abonnements de trains coûtent cher et nous souhaitons que les passagers ne s'ennuient pas... Vous devrez donc leur proposer la possibilité de changer de train pour qu'ils puissent expérimenter une toute autre sensation ! Pour cela, vous implémenterez une attente en gare. Une gare est un point que vous choisirez pour usne locomotive donnée (celui-ci peut très bien correspondre au point de départ choisi pour votre locomotive), la gare ne doit pas se trouver dans le tronçon partagé (section critique). Les gares choisies n'ont pas besoin d'être proches physiquement. Un train après avoir effectué son nombre N de tours définis devra attentre à sa gare que l'autre train termine son nombre de tours et arrive à sa gare, une fois les deux trains arrivés ils attendront 2 secondes, que les passagers puissent monter et descendre avant de repartir.

Pour vérifier correctement votre système et tester au mieux les fonctionnalités, nous vous conseillons d'avoir des valeurs différentes pour le nombre de tours que doit effectuer chaque locomotive et pour le nombre de tours nécessaires avant l'attente en gare.

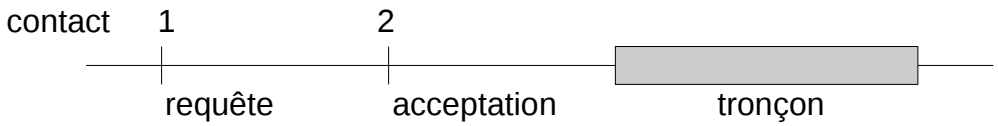


Votre programme devra avoir un arrêt d'urgence actionné par l'interface graphique. Cette fonctionnalité permettra d'arrêter toutes les locomotives immédiatement.

2.2 Programme 2

Reprenez votre code du programme 1 et introdusez le dans le projet *prog2* afin de le modifier et y ajouter une gestion de la priorité pour les accès au tronçon commun. Dans ce programme, la locomotive devra lors d'un accès au tronçon commun d'abord signifier sa demande (après le passage sur le contact *requête*), puis vérifier si l'accès lui est autorisé directement ou non (après le passage du second contact *acceptation*). La demande du tronçon commun se réalise donc en 2 étapes délimitées par des contacts qui précèdent le tronçon (seront différents selon le parcours ainsi que le sens d'arrivé au tronçon) :

contact 1 : formulation de la requête avec l'identifiant de la locomotive en paramètre.
contact 2 : arrêt de la locomotive ou passage de celle-ci en cas d'acceptation.



Au moment du passage sur le contact *d'acceptation*, si une locomotive de priorité supérieure a demandé le tronçon, alors la locomotive courante doit s'arrêter. Elle doit évidemment aussi le faire si le tronçon est déjà occupé.

Les priorités sont définies de manière dynamique selon les règles suivantes :

1. Chaque locomotive possède une priorité qui lui est propre. Les priorités sont comprises entre 0 et 10.
2. Si deux locomotives ont la même valeur de priorité, c'est la première qui arrive dans la section critique qui est prioritaire.
3. Lorsque le programme est initialisé, la locomotive avec la priorité la plus haute obtiendra l'accès à la section critique.
4. À chaque attente des trains en gare, la priorité de chaque locomotive doit être modifiée de manière aléatoire.
5. Le système de priorité change de *priorité à la valeur la plus grande* à *priorité à la valeur la plus petite* à chaque attente en gare. Donc, lorsque les deux trains repartent après une attente en gare, la gestion de la priorité est inversée. (État représenté par `HIGH_PRIORITY` et `LOW_PRIORITY` définis ci-dessous).

3 Remarques

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

```
class SharedSectionInterface
{
public:

    enum class PriorityMode {
        HIGH_PRIORITY,
        LOW_PRIORITY,
    };

    virtual void request(Locomotive& loco, int priority) = 0;
    virtual void access(Locomotive& loco, int priority) = 0;
    virtual void leave(Locomotive& loco) = 0;
    virtual void togglePriorityMode() = 0;
};
```

Il est impératif de respecter cette interface, car les tests du code l'utiliseront tel quel. Pour le programme 1, seules les fonctions `access()` et `leave()` sont à implémenter et PriorityMode n'est pas présent. Pour le programme 2, deux nouvelles fonctions `request()` et `togglePriorityMode()` sont ajoutées, et quelques adaptations sont à faire dans ces deux fonctions pour que la gestion de priorité s'intègre correctement.

- Le code du simulateur de maquettes ainsi que la documentation du simulateur et des maquettes est fourni.
- Le projet à ouvrir dans QtCreator est `QtrainSimStudent.pro`. Il s'agit d'un projet décomposé en 2 sous-projets : prog1 et prog2. Vous devez développer le programme 1 dans prog1 et le programme 2 dans prog2. Le script de génération de l'archive récupérera les deux codes.
- 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).
- Afin de faire rentrer ou sortir les locomotives du tronçon commun (section partagée) il vous faudra commander les aiguillages de manière appropriée.
- Ne pas modifier les fichiers `locomotive.h/cpp`, `launchable.h` et `sharedsectioninterface.h`. Vous pouvez créer de nouveaux fichiers ou classes pour gérer vos parcours au besoin.
- L'arrêt d'urgence est une nécessité pour tout système critique où un arrêt peut être réalisé. Dans le présent laboratoire, une manière simple et triviale de réaliser cet arrêt consiste à faire un appel à la procédure `mettre_maquette_hors_service`. Ceci équivaut à couper toute l'alimentation du réseau ferroviaire. Dans le contexte de ce laboratoire, il est demandé d'arrêter les deux locomotives sans utiliser cette fonction.
- Vous devrez nous rendre les deux programmes complets, selon les consignes données en annexe. Seul le code du programme 2 sera relu, mais les deux programmes seront évalués en exécution.
- 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 toruvera au même niveau que le script `pco_rendu.sh`, ce dernier servant à générer l'archive à rendre sur Cyberlearn.
- 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.

4 Barème de correction

Conception et conformité à l'énoncé	40%
Exécution et fonctionnement	20%
Codage	5%
Documentation	25%
Commentaires au niveau du code et en-têtes des fonctions	10%