Labo 4 : Télécabine

Auteurs: Müller Robin, Teixeira Carvalho Stéphane

Description des fonctionnalités du logiciel

Lors de ce laboratoire, il nous a été demandé d'implémenter de la synchronisation entre thread à l'aide de sémaphore dans une application de simulation de télécabine.

Il a fallu implémenter cela car le système qui nous a été donné est construit de telle sorte à avoir un ensemble de `N` thread qui seront les skieurs et d'un thread qui représentera la télécabine.

La cabine fera monter des skieurs sur la piste et ils descendront alors la piste chacun à leur rythme. Les threads n'arriveront donc pas forcément en même temps en fin de piste. La télécabine, une fois tous les skieurs déchargés, effectuera le trajet de retour à la station pour prendre plus de skieurs. Ce trajet s'effectue avec la cabine vide s'il n'y a aucun skieurs en attente.

La télécabine ne pourra pas supporter plus de `M` skieurs. Une fois toutes les places prises, les skieurs attendront la prochaine remontée. Ils devront donc attendre leur tour.

Si la télécabine termine son service et que des skieurs sont en attente ils ne pourront pas effectuer la montée, ils rentrent.

Lorsque la fin de service est annoncé la cabine si elle se préparait à monter charge les skieurs et effectue la montée puis elle décharge des skieurs et redescends puis s'arrête.
Sinon si elle était en descente elle s'arrête à la fin de la descente.

Il est possible de voir d'après les explications ci-dessus qu'il faut effectivement implémenter de la synchronisation entre les threads de par le fait que les skieurs ne peuvent pas descendre la montagne sans avoir pris la télécabine ou encore qu'il ne peuvent pas monter à plus de `M` personnes etc. . .

Choix d'implémentation

En prenant en compte les fonctionnalités que le problème expliqué ci-dessus comporte, voici comment nous avons implémenté la solution.

Dans un premier temps nous avons décidé du comportement de la télécabine et des skieurs. Voici 2 flowchart expliquant le fonctionnement.

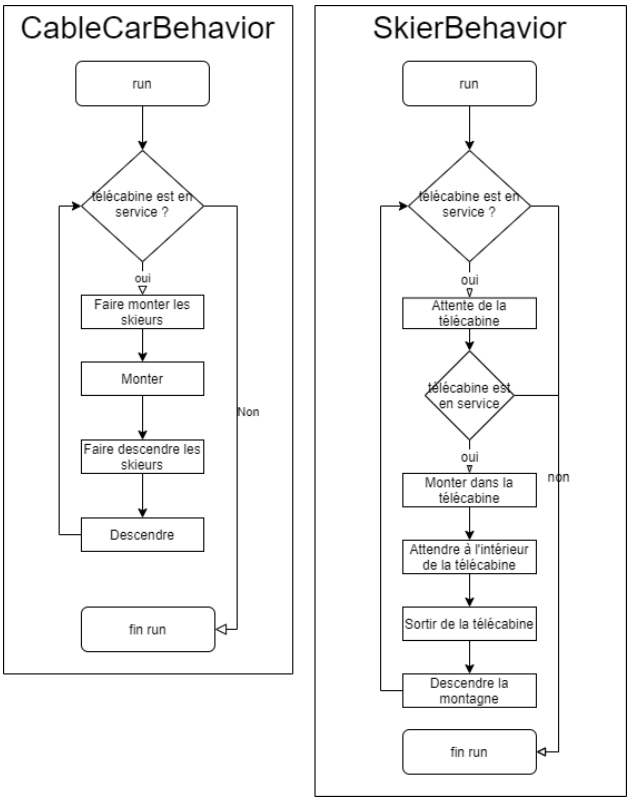

Figure 1: Comportement programme

Si nous commençons par la `télécabine` son comportement est très basique si elle est en service elle fait monter et descendre des skieurs. Même si aucun skieurs n'est présent lors de l'arrivée de la télécabine celle-ci montera quand même.

Pour que la `télécabine` effectue la remontée il faut que tout les skieurs qu'elle va monter aient fini de monter dans la télécabine. La télécabine ne peut pas partir avec un skieur en moins parce qu'elle est partie trop vite.

Une fois que la `télécabine` a fini sa montée, elle doit faire descendre tous ces skieurs avant de pouvoir redescendre. Une fois tous les skieurs hors de la télécabine celle-ci descend et le cycle recommence.

Si nous prenons maintenant les `skieurs`, on sait que si la cabine est en service, il doit attendre la télécabine car celle-ci pourrait être en haut de la montagne en train de écharge ou elle pourrait être en montée ou descente.
Une fois que la télécabine arrive le `skieur` doit à nouveau vérifier que la télécabine est en service car cela pouvait être sa dernière remontée et donc elle ne remontera plus. Dans ce cas le comportement du skieurs est stoppé.

Par contre, si la télécabine est en service, le skieur va alors monter dans celle-ci puis, attendre à l'intérieur le temps que les autres skieurs montent dans la télécabine et que la montée se termine. Une fois la montée terminée, il va sortir de la télécabine et descendre la montagne.

Nous n'avons bien sûr pas encore démontré comment nous avons implémenter la synchronisation car ici le

but est d'expliquer simplement le comportement que nous voulons des 2 classes néanmoins nous pouvons déjà voir que les threads de la `télécabine` et des `skieurs` auront besoin de signaux pour arrêter leur exécution ou la reprendre.

Une fois les comportements mis en place nous avons implémenté la classe `PcoCableCar.cpp`.

Pour l'implémentation des différents fonctionnements et de la synchronisation entre threads nous avons utilisés 4 sémaphores.

```
PcoSemaphore cableCarLoad;
PcoSemaphore cableCarUnload;
PcoSemaphore skieurInside;
PcoSemaphore skieurOutside;
PcoSemaphore mutex = PcoSemaphore(1);
PcoSemaphore mutexloadSkiersEndService = PcoSemaphore(1);
```

La première sémaphore `cableCarLoad` va permettre à un skieurs de faire une demande(`acquire`) pour monter dans la télécabine lorsque un skieur attendra la télécabine(fonction `waitForCableCar`). Cette sémaphore nous permet également de maintenir les demandes d'accès des skieurs dans l'ordre grâce à la FIFO mise en place dans la conception de sémaphore.
Les demandes seront ensuite accepté(`release`) lorsque que la télécabine fera monter les skieurs.

La deuxième sémaphore `cableCarUnload` permet à un skieur de faire une demande(`acquire`) pour descendre de la télécabine. Ainsi tant que la télécabine ne permet pas au skieur de descendre(`release`) celui-ci attends à l'intérieur de la télécabine(`waitInsideCableCar`).

La sémaphore `skieurInside` permet à la télécabine d'attendre que tous les skieurs soit entrés. La télécabine va indiquer au skieurs que celui-ci peut monter(`acquire`) et une fois que celui-ci a terminé de monter(`release`), la télécabine prend le prochain skieur autorisé à monter. Ainsi la télécabine ne peut pas partir tant que les skieurs ne sont pas tous complètement montés.

La sémaphore `skieurOutside` permet d'attendre que tous les skieurs soit sortis. Elle a le même fonctionnement que `skieurInside` mais sauf qu'elle permet d'éviter de faire descendre la télécabine tant que tous les skieurs ne sont pas tous descendus.

La sémaphore `mutex` comme son nom l'indique va nous permettre d'avoir un verrou dans notre programme. Nous devons mettre cela en place car nous aurons des zones critiques, principalement lié à la variable `nbSkiersWaiting`. Cette variable est utilisée par les threads skieurs lorsqu'ils attendent la télécabine (dans la fonction `waitForCableCar`) ainsi que par la télécabine lors du chargement des skieurs et la fin du service.

Une seconde variable, `nbSkiersInside`, est uniquement utilisée par l'unique thread télécabine. Elle n'a donc pas besoin d'être protégée. Elle est utilisée lors du chargement et déchargement des skieurs.

Nous avons égalemet défini un deuxième mutex se nommant `mutexLoadingEndService` ce verrou va nous permettre d'éviter d'avoir des problèmes de deadlock avec la fonction loadSkiers.
Voici comment le deadlock survenait :
Au début de la fonction nous effectuons une opération qui nous indique le nombre de skieur que nous allons mettre dans la cabine(`nbToLoad = std::min(nbSkiersWaiting, capacity)`) puis nous effectuons une première boucle des release des skieurs(cableCarLoad) en fonction du nombre de skieur à charger.
Puis, c'est au moment de la deuxième boucle pour attendre que les skieurs montent(skieurInside.acquire), qui se base également sur nbToLoad, que nous avons un deadlock si un endService est apparu.
Cela provient du fait que lors d'un endService nous allons relacher tout les skieurs pour qu'il arrête d'attendre et donc nous mettons nbSkiersWaiting à 0. Donc si nbToLoad est supérieur à 0 nous avons alors un problème car nous allons attendre qu'un skieur monte alors que tous ceux-ci ont déjà été relachés par endService et se sont arrêtés.

Cela est donc éviter en mettant le acquire du verrou `mutexLoadingEndService` en début de chaque fonction et un release en fin ainsi si un loadSkier ou un endService est en exécution la fonction qui voulait s'exécuter(endService ou loadSkier) doit attendre la fin de son exécution.

Pour avoir une idée du résultat voici le code de la fonction loadSkier :

```
void PcoCableCar::loadSkiers()
{
    // Permet d'attendre la fin d'execution d'un EndService si exécuté
    mutexLoadingEndService.acquire();
    qDebug() << "loadSkiers()";
    // Calcul du nombre de skieurs à charger
    mutex.acquire();
    unsigned nbToLoad = std::min(nbSkiersWaiting, capacity);
    mutex.release();

    // On laisse les skieurs entrer dans la télécabine
    for (unsigned i= 0; i < nbToLoad; ++i) {
            cableCarLoad.release();
            mutex.acquire();
            --nbSkiersWaiting;
            mutex.release();
    }
    // On attend que tous les skieurs entrent dans la télécabine
    for(unsigned i = 0; i < nbToLoad; ++i){
        skieurInside.acquire();
        ++nbSkiersInside;
    }
    mutexLoadingEndService.release();
}
```

Dans le code nous avons utiliser la variable `nbSkiersInside` en faisant des incrémentations et décrementations sans jamais la protéger d'un mutex. Nous avons fait cela car cette variable ne sera utilisée que par le thread cablecar et donc nous n'avons pas de problème de concurrence.

Grâce aux sémaphores expliquées ci-dessus il nous est maintenant possible de gérer la synchronisation entre les thread skieurs et le thread de la télécabine.

Tests effectués

Pour les tests nous avons decidés d'inclure 2 démonstrations de tests avec 1 puis 2 skieurs. La télécabine dans le premier test à une capacité de 5 places tandis que dans le deuxième elle a une capacité de 1.

Pour les tests plus conséquent(plus d'affichage) nous avons fait un tableau expliquant le test ainsi que son résultat.

Test avec un skieur

```
Démarrage de la simulation de skieurs et télécabine ...
Nombre de skieurs : 1
Capacité du télécabine : 5

[START] Thread du télécabine lancé
[START] Thread du skieur 1 lancé
Le télécabine monte
waitForCableCar( 1 )
Le télécabine atteint le sommet
Le télécabine descend
Le télécabine atteint le bas
goIn( 1 )
```

```
waitInsideCableCar( 1 )
Le télécabine monte
Le télécabine atteint le sommet
goOut( 1 )
Skieur 1 est en train de skier et descend de la montagne
Le télécabine descend
Le télécabine atteint le bas
Le télécabine monte
Le télécabine atteint le sommet
Le télécabine descend
Skieur 1 est arrivé en bas de la montagne
waitForCableCar( 1 )

Arret du service
Le télécabine atteint le bas
cablecar se stop
[STOP] Thread du télécabine a terminé correctement
[STOP] Thread du skieur 1 a terminé correctement
```

Dans le résultat ci-dessus on voit bien que le skieur attend la télécabine.
Puis, tant que la télécabine monte le skieur attend à l'intérieur.
Le comportement du skieur est donc bien celui attendu. On peut aussi constater que la télécabine attend que le skieur monte à l'intérieur avant de commencer la remontée et qu'elle attend qu'il descende avant de redescendre.

Grâce au test ci-dessus donc que les comportements des skieurs et celui de la télécabine sont bien ceux desirés.

Test avec deux skieurs

```
Démarrage de la simulation de skieurs et télécabine ...
Nombre de skieurs : 2
Capacité du télécabine : 1

[START] Thread du télécabine lancé
[START] Thread du skieur 2 lancé
[START] Thread du skieur 1 lancé
Le télécabine monte
waitForCableCar( 1 )
waitForCableCar( 2 )
Le télécabine atteint le sommet
Le télécabine descend
Le télécabine atteint le bas
goIn( 1 )
waitInsideCableCar( 1 )
Le télécabine monte
Le télécabine atteint le sommet
goOut( 1 )
Skieur 1 est en train de skier et descend de la montagne
Le télécabine descend
Le télécabine atteint le bas
goIn( 2 )
waitInsideCableCar( 2 )
Le télécabine monte
Le télécabine atteint le sommet
goOut( 2 )
```

```
Skieur 2 est en train de skier et descend de la montagne
Le télécabine descend
Skieur 2 est arrivé en bas de la montagne
waitForCableCar( 2 )
Le télécabine atteint le bas
goIn( 2 )
waitInsideCableCar( 2 )
Le télécabine monte
Skieur 1 est arrivé en bas de la montagne
waitForCableCar( 1 )
Le télécabine atteint le sommet
goOut( 2 )
Skieur 2 est en train de skier et descend de la montagne
Le télécabine descend

Arret du service
Skieur 2 est arrivé en bas de la montagne
Skieur  2  se stop
Le télécabine atteint le bas
cablecar se stop
[STOP] Thread du télécabine a terminé correctement
[STOP] Thread du skieur 2 a terminé correctement
[STOP] Thread du skieur 1 a terminé correctement
```

Grâce au test ci-dessus on peut voir que la télécabine prend les threads dans l'ordre et que le comportement des threads fonctionnent toujours comme nous le désirons.

La télécabine prend bien un seul skieur à la fois.

Tests en variant le nombre de skieurs et la capacité de la télécabine

Chaque test a été effectuée à 10 reprises, avec un démarrage des threads dans un ordre aléatoire sans délai ainsi qu'en commençant par les skieurs puis le télécabine avec un délai de 100ms entre chaque threads.

#	Nb skieurs	Capacité	Résultat
1	10	5	10/10
2	10	2	10/10
3	10	10	10/10
4	10	20	10/10
5	40	20	10/10
6	100	20	10/10

Tests en supprimant les délais des skieurs et de la télécabine

Les tests ont été effectués avec un démarrage des threads dans un ordre aléatoire sans délai. Ces tests ont également été effectué à 10 reprises.

#	Nb skieurs	Capacité	Résultat
1	10	5	10/10
2	10	2	10/10
3	10	10	10/10
4	10	20	10/10
5	40	20	10/10
6	100	20	10/10