Rapport du projet trains

Julie Rymer 11290993

Utilisation des programmes :

Les programmes ont été testé sur une machine virtuelle Ubuntu 16.04.
Les quatre se lancent de la même manière :

```
$gcc main.c header.h -Wall -Wextra -Werror -O3 -lpthread -lm -DDEBUG -DTEST -o train.out
$./train.out
```

Il faut néanmoins ajouter le tag « -lrt » pour le projet 4.
Lorsque le programme se lance, des informations sur le déplacement des trains s'affichent sur stdout. Pour ne plus les voir, on peut rediriger stdout ou retirer « -DDEBUG » des tags de compilation. S'affiche aussi sur stderr la progression des tests et leurs résultats à la fin. Pour ne plus les voir, on peut rediriger stderr ou retirer « -DTEST » des tags de compilation.
On peut également, dans chaque fichier *header.h* des différents projets, modifier les valeurs de N_TRAJETS (nombre de trajets complets effectués par chaque train) et MAX_SEED (nombre de seed random pour lesquels ces trajets sont mesurés), ce qui aura bien sûr une influence sur le temps d'exécution du programme. J'ai compté sur ma machine en moyenne 2 minutes d'exécution par seed pour dix trajets.

Structures utilisées :

Pour représenter le réseau ferré, j'ai utilisé deux structures. La première est la structure *liaison*, qui représente la liaison entre deux gares. Elle contient un booléen indiquant si cette liaison existe sur le réseau, un ou deux outils de synchronisation de thread (mutex, sémaphore, verrou ou file de message en fonction du projet) ainsi qu'un objet différent pour chaque projet en fonction des besoins : le nombre de train présent sur la liaison et/ou une file d'attente des trains qui peux aussi contenir des informations complémentaires dans une autre structure *trajet* toujours selon le projet (un booléen indiquant si le train est arrivé ou son temps de trajet sur la liaison). Toutes les liaisons possibles du réseau sont stockées dans une variable globale *liaisons* qui contient un tableau statique à deux dimensions, de taille du nombre de stations au carré, qui sera ensuite rempli en fonction des liaisons données par la consigne. J'en profite pour spécifier dès à présent que, toutes les tailles des données à stocker étant connues par avance, je ne me suis vraiment pas embêté avec la mémoire et je n'ai utilisé absolument partout que des tableaux statiques.

La deuxième structure du réseau est la structure train qui contient son nom, son trajet est la longueur de celui-ci, ainsi que, selon le projet, un ou deux outils de synchronisation de thread (sémaphore ou file de messages). Je garde une variable globale trains sur un tableau de trois trains, rempli avec les données en consigne.

Enfin, la dernière structure mise en place est simplement une structure thread_infos qui, comme son nom l'indique sert à transmettre des informations à un thread car c'est un objet de ce type qui sera passé en argument à la fonction du thread. Elle contient l'index du train dont doit s'occuper le thread ainsi que le seed random actuellement utilisé pour que le thread puisse remplir correctement le tableau des temps de trajet.

Différentes stratégies selon le projet :

Projet 1 : Mutex

Pour le premier projet, je me suis inspirée du modèle lecteur-écrivains avec uniquement différent groupes de lecteurs. Les différents groupes de lecteurs correspondent à des trains allant dans des directions opposées. La grosse différence avec ce modèle vient de la gestion du non-dépassement des trains entre eux, qui se fait à l'aide d'une file d'attente de train pour chaque liaison, appelée *train_fifo*. Il y a un mutex par liaison (initialisation dynamique car le tableau des liaisons est rempli par des fonctions, il n'y aura jamais d'initialisation statique pour une variable non unitaire) plus un mutex global (initialisation statique). Chaque thread correspond à un train, il en a donc 3. Aller d'une station à une autre est un acte de « lecture ». Le thread commence donc par verrouiller le mutex global (pour éviter un interblocage avec un train allant dans l'autre sens), verrouille le mutex de sa liaison pour éviter un accès concurrentiel à *train_fifo*, se rajoute dedans à sa bonne place et, si le thread constate être le premier à cet endroit grâce à sa place dans la file, verrouille le mutex de la liaison dans l'autre sens. Le thread déverrouille ensuite tous les autres mutex pour laisser passer les autres « lecteurs de son groupe », soit les trains empruntant la même liaison dans le même sens. Il va ensuite se mettre en pause pendant 1 à 3 seconde(s) avant de reverrouiller le mutex de sa liaison pour accéder à nouveau à *train_fifo*. S'il constate qu'il n'est pas le premier train de la file d'attente, il se met en pause et libère le mutex grâce à une variable conditionnelle globale pour tout le projet (initialisation statique). J'aurais pu faire une variable conditionnelle par liaison, mais vu qu'il n'y a que trois trains, cette situation ne peut se produire qu'à une seule liaison à la fois, c'est donc bien plus rentable en mémoire de n'avoir qu'une variable pour toutes les liaisons. Quand le thread en pause reçoit le signal pour s'en défaire et qu'il constate que c'est bien à son tour (contrairement à OSX, Linux ne permet pas de ne réveiller qu'un thread précis, on est donc obligé de les réveiller tous et qu'ils vérifient par eux même s'ils doivent se rendormir ou non), il peut alors se retirer de la file d'attente et, s'il est le dernier, libérer les éventuels trains dans l'autre sens, sinon et si d'autres trains attendent après lui, leur signifier de sortir de pause par l'envoi d'un signal, avant de débloquer le mutex de sa liaison.

Projet 2 : Sémaphore

Pour ce projet, le modèle producteur-consommateur m'a paru plus adapté. Il y a non seulement des threads pour chaque trains (producteurs), mais également des threads qui vont gérer l'arrivée en gare des trains (consommateurs), donc un pour chaque liaison, ce qui fait 3+11 soit 14 threads. Tous les sémaphores sont des sémaphores nommés, travaillant sous OSX, les sémaphores anonymes ne m'était pas accessibles. Il existe un sémaphore global (initialisation statique) et un sémaphore par liaisons (initialisation dynamique), *sem_engage*, qui jouent les mêmes rôles que les mutex du projet 1 : éviter l'interblocage, l'accès concurrentiel aux propriétés de la liaison et bloquer les threads des trains allant dans l'autre sens. Le début et la fin de la fonction des threads des trains vont donc être similaire à ceux du projet 1, à ceci près que pour savoir si le train est le premier/dernier, on utilisera un compteur et non plus la file d'attente. Ce qui change est dans le trajet du train à proprement parler. Le thread va récupérer le temps qu'il doit passer à voyager, puis, tout en prenant garde aux accès concurrentiels, va s'ajouter à la file d'attente en indiquant ce temps et faire un post sur un autre sémaphore, *sem_arrive* (initialisation dynamique), pour indiquer aux threads gérant cette liaison que la file d'attente est

à présent remplie (jouant donc le rôle de producteurs). Il va ensuite attendre que ce thread lui indique qu'il est arrivé à destination en faisant un wait sur un dernier sémaphore, qui appartient cette fois-ci au train lui-même, afin que seul le thread voulu reçoive le signal de continuer.

Les threads des liaisons sont donc ceux qui sont responsables de faire arriver les trains dans le bon ordre et avec le bon temps de trajet. Les fonctions de ces threads font une boucle infinie et attendent que la file d'attente soit remplie, ce qui leur ait indiqué par le sémaphore *sem_arrive*. Ils sont donc les consommateurs. Vu que ces fonctions ne se termine jamais, les liaisons ne pouvant pas savoir quand tous les trains ont fini de voyager, ces threads seront arrêtés en les annulant. Après avoir été notifié que leur file d'attente contient au moins un train, ces threads vont en faire la copie en vidant l'original (pour pouvoir travailler sans bloquer trop longtemps *sem_engage* qui évite, entre autres, l'accès concurrentiel à la file). Ils vont ensuite, pour chaque train présent sur la file, décrémenter au besoin *sem_arrive*, attendre le temps que doit attendre le train en cours, moins le temps déjà attendu si des trains sont déjà passé, avant de notifié le train en question qu'il a terminé son trajet. Ça permet donc de ne pas attendre de temps superflus pour des trains qui se suivent tout en garantissant l'ordre d'arrivée.

Projet 3 : RWLock

Pour le troisième projet, on retourne à nouveau dans un modèle lecteurs-lecteurs, avec des threads qui ne représentent que les trains. Il y a là encore un verrou global (initialisation statique) qui évite l'interblocage et un verrou pour accéder aux propriétés de la liaison, *rwlock_train_statut* (initialisation dynamique), qui sont utilisé tels les mutex du projet 1, ils ne sont donc utilisés qu'en écriture pour n'avoir toujours qu'un seul accès à la fois. Un autre verrou, *rwlock_train_deplace* (initialisation dynamique), est utilisé en lecture est en écriture pour gérer les déplacement d'un train. Au démarrage du thread, hormis les procédures habituelles, on va verrouiller en lecture le verrou de déplacement (ne bloquant donc pas les autre trains) et on va ajouter le train en file d'attente avec un booléen indiquant qu'il n'est pas encore arrivé. Le train va ensuite faire le sleep qui correspond à son temps de trajet. Puis, en s'occupant des accès concurrents, il va mettre à jour son statut dans la file d'attente pour indiquer qu'il est arrivé. Puis il va appeler une fonction qui met à jour la file d'attente en nettoyant tous les trains qui sont arrivés (c'est à dire dont le booléen est à vrai et qui ne sont derrière aucun train sauf si le ou les train(s) en question sont aussi arrivés. L'accès à la ligne dans l'autre sens sera bien sûr libéré s'il s'agissait du dernier train (fila d'attente à présent vide). Enfin, le train se retire des lecteurs du verrou de déplacement et, s'il n'a pas fait partie des trains qui ont été retiré de la file d'attente (qui sont arrivés donc) parce qu'il est bloqué par un train devant lui, il va demander un accès temporaire en écriture au verrou de déplacement, ceci afin de le bloquer tant que tous les autres trains ne soient arrivés et ne se soient donc retiré des lecteurs de ce verrou. Ce n'est pas idéal car ça impose que si trois trains sont sur la même liaison et sont arrivés dans l'ordre 1, 2, 3, mais que leur vitesse les fait arriver dans l'ordre 2, 1, 3, le train 2 sera bloqué jusqu'à ce que le train 1 mais aussi le train 3 ne se retirent des lecteurs, alors même qu'il n'aurait pas dû avoir besoin d'attendre le train 3. Mais c'est le seul cas de figure où cette stratégie n'est pas optimale et je n'ai pas trouver de meilleure solution pour garantir l'ordre d'arrivée avec les verrous.

Projet 4 : MQueue

Pour les files de messages, j'ai eu plus de mal car, travaillant sous OSX où elles n'existent pas, j'ai été obligée de coder à l'aveugle avant les tests sur machine virtuelle. J'ai à nouveau utilisé un modèle producteur-consommateur, qui paraissait de manière assez évidente être le plus

approprié à cette structure. Il y a donc dans ce projet un thread par train et un thread par liaison, avec beaucoup de similitude avec le projet 2. Je ne préciserais pas les méthodes d'initialisation des files de messages puisque la seule méthode existante est l'initialisation dynamique avec des files nommés. J'ai utilisé la plupart des files de messages de manière similaire à des sémaphores, en envoyant un message d'un caractère dont le contenu importe peu à l'initialisation quand le sémaphore doit démarrer à 1 et en remplaçant les wait et les post par des receive et des send. J'aurais pu faire un code bien plus proche de l'utilisation habituelle des files de messages en envoyant des messages pour demander à un autre thread de se charger de toutes les manipulations de données sans concurrence (puisque les effectuant message après message) en envoyant en même temps un identifiant pour qu'il puisse indiquer quand la tâche est finie par retour de message, voir même en enregistrant un notify plutôt que d'utiliser un thread qui boucle, mais ma méthode m'a paru la plus simple à mettre en place avec le code que j'avais déjà à disposition, même si elle n'est pas forcément optimale pour cette structure. On retrouve donc comme d'habitude une file globale pour éviter l'interblocage et une file *mqueue_status* pour les accès concurrentiels aux propriétés des liaisons. Il n'y a, dans cette implémentation, pas de file d'attente de train mais juste un compteur pour que le premier/dernier train bloque/débloque l'accès aux trains dans l'autre sens comme d'habitude.

Le non dépassement entre trains est ici géré en envoyant un message dans la file par liaison *mqueue_engage* contenant l'id du train lorsque celui-ci s'engage sur une voix (producteur). Le train va ensuite faire son sleep puis prévenir qu'il a fini en envoyant un message sur la file *mqueue_arrive_start* qui appartient au train et non pas à la liaison ce coup-ci. Enfin, il va attendre confirmation d'être réellement arrivé par retour de message sur la file *mqueue_arrive_ended*, toujours une propriété du train. Côté consommateur, les threads par liaison vont attendre de recevoir un id sur *mqueue_engage* puis vont attendre de savoir si le train correspondant à l'id reçu est arrivé par *mqueue_arrive_start*. Enfin, ils vont prévenir le train en question qu'il peut à présent continuer sa route par le biais de *mqueue_arrive_ended*. Puisque plus aucun message n'est lu sur *mqueue_engage* tant que le train déjà arrivé n'a pas fini d'être traité, aucun dépassement de train ne peux se produire.

Fonctions :

Communes à tous les projets

- **init_reseau** : Initialise le réseau ferré en créant les liaisons correctes entre les gares, en initialisant les mutex/sémaphore/verrous/file de messages et en assignant les trajets aux trains.
- **creer_trajet** : Permet de copier un trajet pour l'assigner à un train.
- **relier_stations** : Initialise une liaison.
- **clean_mutex/semaphore(s)/rwlock(s)/mqueue(s)** : Détruit et libère au besoin les différents objets alloués.
- **voyage** : Fait passer un train d'une station à une autre.
- **roule** : Fonction passée aux threads qui gèrent les trains, leur fait faire N_TRAJET fois leur trajet, et enregistre le temps pris par chaque trajet dans le tableau global *temps_trajet*.
- **affiche_test_result** : Calcule et affiche la moyenne et l'écart type des temps de trajet pour chaque train.

- **handle_sig** : Fonction enregistrée pour être appelée si l'utilisateur fait un ctrl-C pour nettoyer tout ce qui doit être nettoyé avant de quitter.
- **main** : Appelle les fonctions d'initialisation, créer les threads, affiche les résultats et nettoie tout avant de quitter.

Projet 1 : Mutex

- **push_fifo** : Met le train passé en argument dans la file d'attente des trains de la liaison passé en argument et renvoit un booléen indiquant si le train est le premier de la file
- **pop_fifo** : retire le premier train de dans la file d'attente des trains de la liaison passé en argument et renvoit un booléen indiquant si le train retiré était aussi le dernier.

Projet 2 : Sémaphore

- **push_fifo** : Met un train et son temps de trajet dans la file d'attente des trains de la liaison passé en argument
- **gere_arrive_gare** : Fonction passée aux thread qui gèrent les liaisons, fait une boucle infinie (la fonction sera stoppée par l'annulation des threads) en attendant de devoir gérer l'arrivée d'un ou plusieurs train.

Projet 3 : RWLock

- **push_fifo** : Met un train dans la file d'attente des trains de la liaison passé en argument avec le statut non arrivé.
- **update_statut_fifo** : Met le statut du train voulu dans la file d'attente comme étant arrivé.
- **gere_arrive_gare** : Fait arriver les trains qui doivent l'être dans la file d'attente (ceux qui ont le statut arrivé et qui ne sont pas derrière un train qui ne l'est pas) et renvoit un booléen indiquant si le train passé en argument fait partie de celui ou ceux qui sont arrivés.

Projet 4 : MQueue

- **gere_arrive_gare** : Sert à la même chose que dans le projet 2.

Tests :

Les tests ont été réalisés à l'aide de la variable globale *temps_trajet* qui est un tableau à trois dimensions de double qui stocke tous les temps de trajets pour chaque train pour chaque seed random testé. Les trains font N_TRAJETS trajets chacun (soient 10 dans mes tests) pour chaque seed random. La fonction *clock_gettime* avec le mode *CLOCK_MONOTONIC_RAW* a été utilisé pour les mesures car c'est apparemment ce qu'il y a de plus précis pour mesurer un écoulement de temps en C. Par cette fonction, le temps est donné dans une structure *timespec* en secondes et nanosecondes, je l'ai donc simplement converti en seconde et stocké dans un double pour pouvoir faire mes calculs de manière simple.

Je n'ai pu faire les tests pour les seeds allant de 1 à 1000 que pour le projet 4, pour les autres, la machine virtuelle a décidé de planter juste avant la fin (le projet 4 a juste été plus rapide. Les tests prenant une journée et demi à tourner, étant en déplacement je n'avais pas tout ce temps, je n'ai donc pu les relancer pour les trois autres projets que jusqu'à 500. J'ai calculé la moyenne et l'écart type en appliquant les formules usuelles sur tous les trajets en parcourant *temps_trajet*.

Voici les résultats :

- Projet 1 : Mutex :

```
Tests pour 10 trajets, pour des seeds allant de 1 à 500

Moyenne trajet train 1 = 92.524495 sec
Écart type trajet train 1 = 260.227569 sec

Moyenne trajet train 2 = 110.786240 sec
Écart type trajet train 2 = 311.570092 sec

Moyenne trajet train 3 = 107.176508 sec
Écart type trajet train 3 = 301.424844 sec
```

- Projet 2 : Sémaphore :

```
Tests pour 10 trajets, pour des seeds allant de 1 à 500

Moyenne trajet train 1 = 96.034594 sec
Écart type trajet train 1 = 270.112516 sec

Moyenne trajet train 2 = 118.009880 sec
Écart type trajet train 2 = 331.914112 sec

Moyenne trajet train 3 = 113.497496 sec
Écart type trajet train 3 = 319.212177 sec
```

- Projet 3 : RWLock :

```
Tests pour 10 trajets, pour des seeds allant de 1 à 500

Moyenne trajet train 1 = 94.274529 sec
Écart type trajet train 1 = 265.156488 sec

Moyenne trajet train 2 = 112.897179 sec
Écart type trajet train 2 = 317.515231 sec

Moyenne trajet train 3 = 107.755147 sec
Écart type trajet train 3 = 303.054824 sec
```

- Projet 4 : MQueue :

```
Tests pour 10 trajets, pour des seeds allant de 1 à 1000

Moyenne trajet train 1 = 79.323886 sec
Écart type trajet train 1 = 224.447943 sec

Moyenne trajet train 2 = 98.759698 sec
Écart type trajet train 2 = 279.428616 sec

Moyenne trajet train 3 = 98.636008 sec
Écart type trajet train 3 = 279.078442 sec
```

On remarque que la structure gagnante en termes de temps moyen comme en termes d'écart type est, assez étonnamment car j'avais l'impression que ce serait la plus lourde et celle que j'avais la moins optimisée, la file de message. Viennent ensuite le mutex et le verrou qui ont des résultats assez proches avec un léger avantage pour les mutex et, puisque leur fonctionnement n'est pas dépendant de l'implémentation choisi par l'OS, leur utilisation serait préférée même si l'avantage était inverse. Le dernier est le sémaphore alors que je le pensais assez performant, comme quoi les tests sont indispensables avant la moindre prise de décision.

Les écarts types restent quand même très importants dans les quatre projets, mais c'est tout à fait normal car les trains attendent, pour chaque liaison, un temps qui est toujours aléatoire, favorisant donc une grande dispersion des résultats. Il y a aussi le fait que, le train 1 finissant toujours avant les trains 2 et 3 (il a un trajet plus court), il leur permet d'aller plus vite une fois parti, même si l'impact de ceci est minime par rapport à la première raison. Je me rends compte à présent que pour avoir des mesures bien plus exploitables pour comparer les performances, j'aurais dû faire en sorte de supprimer le temps d'attente aléatoire du temps écoulé comptabilisé. Même si les moyennes sont probablement représentatives de par le grand nombre de tests, les écarts type ne sont pas exploitables en l'état. On peut aussi remarquer que les moyennes sont très éloignées du temps maximum théorique d'un trajet (4 ou 5 * 3 secondes suivant le train).