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

Prof. Yann Thoma
Gregory Medwed
Fiorenzo Gamba

Laboratoire de Programmation Concurrente
semestre printemps 2020

Producteur-Consommateur pour calcul différé

Temps à disposition : 4 séances

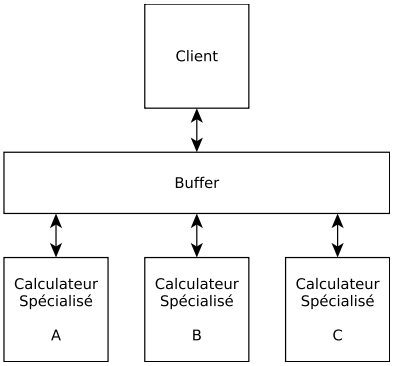
1 Objectifs pédagogiques

— Réaliser un système de type producteur-consommateur avec moniteur de Hoare

2 Cahier des charges

Dans ce laboratoire on s'intéresse à la mise en place d'un système permettant l'exécution concurrente et différée de calculs. Dans le cadre d'une application on aimerait pouvoir distribuer des calculs à réaliser de manière concurrente afin de ne pas devoir les faire dans le thread principal, les calculs seront donc envoyé à des calculateurs spécialisés, les résultats seront ensuite transmis en retour à l'application.

L'architecture globale est la suivante :



Le buffer sert au client à déposer des calculs et à ne pas devoir attendre si les calculateurs spécialisés sont occupés. Les calculateurs vont aller chercher ces calculs afin de les réaliser. Lorsque les calculs sont effectués les résultats sont stockés en retour dans le buffer afin que le client puisse les récupérer.

3 Spécificités du système

Le client ainsi que les calculateurs sont implémentés sous forme de threads de manière a permettre une exécution parallèle.

L'objet buffer sera implémenté sous forme de moniteur de Hoare afin de gérer la synchronisation des accès concurrents du client et des calculateurs ainsi que les attentes.

3.1 Client

Le client va transmettre des calculs à faire et en récupérer les résultats. Dans le cadre d'une application graphique un thread générera les calculs et un autre thread attendra les résultats pour les afficher.

L'interface du buffer côté client est composé des méthodes suivantes :

- `int requestComputation(Computation c);`
  Cette méthode permet de demander d'effectuer un calcul et retourne un id, donné par le buffer, correspondant au calcul.
- `void abortComputation(int id);`
  Cette méthode permet d'annuler un calcul en cours grâce à son id.
- `Result getNextResult();`
  Cette méthode permet de demander les résultats au buffer. Les résultats seront retournés dans le même ordre que l'ordre les demandes de calcul. Cette méthode ne doit pas retourner les résultats de calculs qui ont été annulés.

3.2 Calculateurs

Les calculateurs sont spécialisés et n'effectuent qu'un seul type de calcul, leur rôle est d'aller chercher des calculs à réaliser dans le buffer et de retourner le résultats du calcul.

L'interface du buffer pour les calculateurs est composé des méthodes suivantes :

- `Request getWork(ComputationType computationType);`
  Cette méthode permet au calculateur de demander du travail du type `computationType`, qu'il reçoit sous forme d'une requête de calcul.
- `bool continueWork(int id);`
  Cette méthode permet au calculateur de demander si il doit continuer à travailler sur le calcul avec l'identifiant donné.
- `void provideResult(Result result);`
  Cette méthode permet au calculateur de retourner le résultat du calcul.

Les calculateurs sont implémentés par un thread dont le fonctionnement est le suivant : Le thread commence par demander du travail, ensuite il lance le calcul avec un appel à `startComputation()`, tant que le calcul n'est pas terminé il appelle `advanceComputation()` qui permet d'effectuer une partie du calcul, jusqu'à ce que le calcul soit terminé suite à quoi il renvoie le résultat. Le calculateur contrôle aussi si le calcul n'est pas annulé entre les appels successifs de `advanceComputation()`, si le calcul est annulé le calculateur ne renvoie pas de résultat et demande à nouveau du travail.

Ceci permet aux calculateurs lancés sur un calcul de s'arrêter avant la fin du calcul complet si le calcul est annulé et les rends donc à nouveau disponible pour d'autres calculs.

Si il n'y a pas de calcul à effectuer l'appel de `getWork()` mettra le thread en attente sur une condition dans le moniteur (buffer) et il sera reveillé lorsqu'il y en aura.

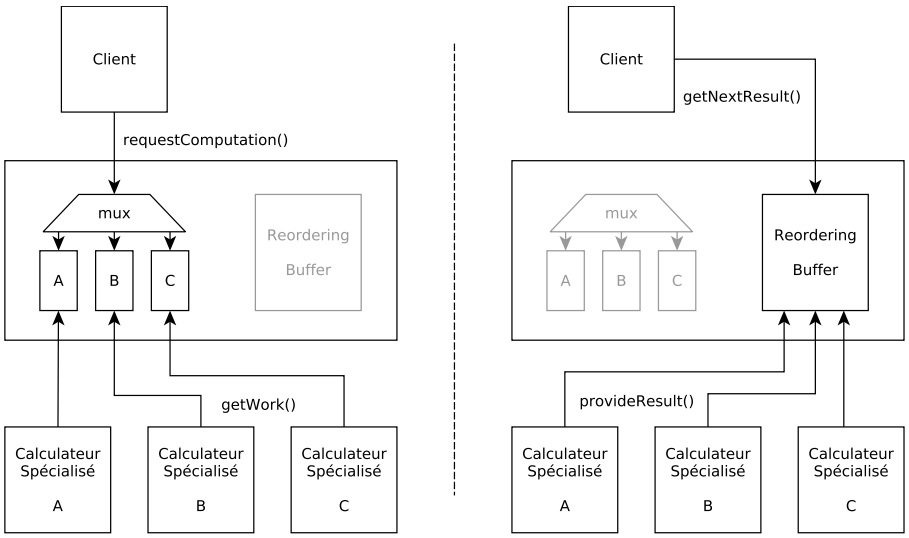
3.3 Buffer

Le buffer est l'objet principal de synchronisation entre threads dans ce laboratoire. Il sera implémenté sous forme de moniteur.

Le buffer doit pouvoir permettre l'accès à un nombre arbitraire de calculateurs spécialisés de chaque type (A,B,C). Par exemple sur une machine avec plus de ressources on lancerait peut-être 5 instances de chaque calculateur, alors que sur une machine plus modeste on lancerait juste une instance de chaque.

Afin de limiter le nombre de ressources utilisées par le buffer on limitera le nombre de requêtes de calcul en attente de chaque type.

Le schéma suivant modélise les fonctions du buffer appelées par le client et les calculateurs.



Les requêtes de calcul émises par le client avec `requestComputation()` sont conservées par type dans le buffer jusqu'à ce qu'un calculateur du type concerné appelle `getWork()`. Une fois qu'un calculateur a fini le calcul il utilise la méthode `provideResult()` du buffer afin de retourner le résultat. Etant donné que les calculateurs peuvent prendre plus ou moins de temps à effectuer un calcul il faut ensuite réordonnancer leur résultats afin que la méthode `getNextResult()` renvoie les résultats dans le même ordre que les requêtes ont été initialement effectuées.

Les flèches des appels de fonctions sont montrées en deux diagrammes séparés pour des raisons de clareté, les threads des calculateurs et le client vont appeler les méthodes comme décrit dans les sections ci-dessus.

Il faut aussi noter que le client peut annuler un calcul à n'importe quel moment et il s'en suit que le calcul ne devrait pas être effectué et le résultat pas retourné.

- Si la requête de calcul se trouve dans le buffer lors de l'annulation elle devra être éliminée.
- Si un calculateur travaille déjà sur la requête il devra arrêter le calcul au prochain appel de `continueWork(id)` et ne pas retourner le résultat.
- Si le résultat est déjà parvenu au buffer il devra être éliminé.
- Si le client annule un calcul dont le résultat a déjà été consommé par getNextResult() il ne doit rien se passer. De même si il demande d'annuler un identifiant inconnu.

Le buffer doit aussi permettre de libérer les threads client et calculateurs lors d'un arrêt du système. Le buffer a une méthode `stop()` qui permet d'annoncer qu'on va arrêter de l'utiliser. La méthode `stop()` doit pouvoir libérer tous les thread actuellement en attente sur les différentes conditions du moniteur.

Note : Ceci est similaire à la fin de service du laboratoire télécabine mais l'implémentation sous forme de moniteur devrait vous faciliter la tâche.

4 Travail à faire

Implémentez le buffer, qui se nomme `ComputationManager` et se trouve dans les fichiers `.h .cpp` correspondants, sous forme de moniteur de Hoare grâce aux méthodes `monitorIn()`, `monitorOut()`, `wait(Condition& cond)` et `signal(Condition& cond)` héritées depuis la classe `PcoHoareMonitor`.

Une application graphique vous est fournie afin d'effectuer les requêtes de calcul ainsi que de visualiser la progression des calculs dans les calculateurs.

4.1 Etape 1

Mettez en place la distribution des calculs grâce aux méthodes :

— `int requestComputation(Computation c);`

— `Request getWork(ComputationType computationType);`

Ceci devrait permettre aux calculateurs de commencer à travailler. Le nombre limite de requêtes de chaque type en attente dans le buffer est passé au constructeur du buffer `ComputationManager()`, par défaut cette valeur est 10. Les deux méthodes sont potentiellement bloquantes et les attentes se réalisent avec l'utilisation de conditions du moniteur. Lors d'une requête de calcul par le client vous pouvez créer la requête avec le constructeur `Request(Computation c, int id)` les identifiants doivent être attribués par le buffer et suivre l'ordre des appels à `requestComputation()`, cet identifiant est retourné lors de l'appel à cette méthode.

4.2 Etape 2

Mettez en place la gestion des résultats grâce aux méthodes :

— `Result getNextResult());`

— `void provideResult(Result result);`

Ceci permettra à l'application de récupérer les résultats retournés par les calculateurs, ceux-ci sont de type `Result` et portent le même identifiant que la requête associée. Le nombre de résultats dans le buffer n'est pas limité. L'appel à `getNextResult()` est potentiellement bloquant si le prochain résultat n'est pas encore disponible. Les résultats devront arriver dans l'ordre pour le client, il faut donc mettre en place une stratégie de réordonnancement des résultats.

4.3 Etape 3

Mettez en place la possibilité d'annuler des calculs demandés par le client grâce aux méthodes :

— `void abortComputation(int id);`

— `bool continueWork(int id);`

Ces méthodes ne sont pas bloquantes (pas d'appel au `wait()` du moniteur) cependant il sera peut-être nécessaire de réveiller des threads en attente (via un appel à la méthode `signal()` du moniteur). Par exemple si un thread client attend le prochain résultat et que celui-ci est annulé alors que le suivant est déjà présent il faudra alors réveiller le thread en attente. De même si un thread client attend pour déposer un calcul de type A car le buffer est plein pour les requêtes de type A, si une des requêtes dans le buffer est annulée, il y aura à nouveau de la place pour une nouvelle requête et il faudra réveiller le thread en attente du dépot.

4.4 Etape 4

Mettez en place la gestion de la terminaison grâce aux méthodes suivantes du buffer :

— `void stop();`

— `private: void throwStopException();`

la fonction `stop()` devra libérer tous les threads en attente sur le buffer et devra empêcher la mise en attente de tout thread lors d'un appel à une méthode du buffer après l'appel de `stop()`. Les appels à `continueWork()` devront maintenant toujours retourner `false`.

Afin de ne pas devoir générer de résultats ou requêtes factices lorsqu'un thread en attente sera réveillé par `stop()`, on signalera à la fonction appelante qu'on a été reveillé à cause de l'arrêt par exception, pour ce faire appelez la méthode privée `throwStopException()` après que le thread réveillé par `stop()` sorte du moniteur (`monitorOut()`). L'appel de `throwStopException()` lève

une exception, qui sera remontée jusqu'à un `catch()`, ceci permet d'éviter de continuer à exécuter du code inutilement après la levée de l'exception et permet aux appelants d'être interrompu proprement pour ensuite gérer la fin comme ils le désirent.

Pour ce faire une solution possible est de modifier les fonctions du buffer avec un test avant et après les appels de `wait()` qui contrôle si le système est arrêté, si oui, sortez du moniteur et appelez `throwStopException()`.

N'hésitez pas à regarder le fichier `src/computeengine.h` dans la méthode `run()` pour voir comment c'est géré depuis l'appelant.

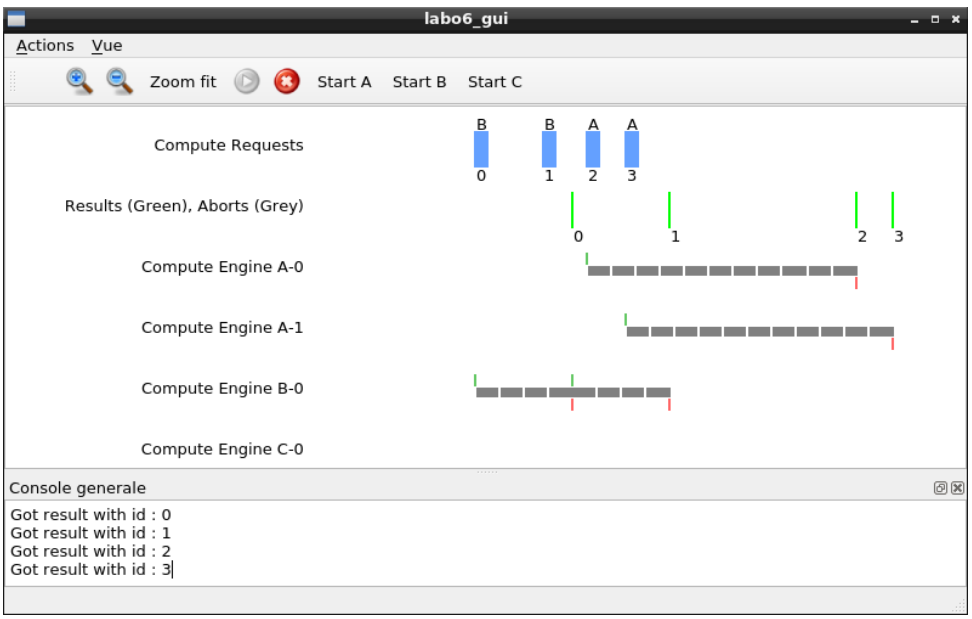
5 Informations sur le projet

Le projet fourni se décompose en deux sous projets, **labo6_gui** et **labo6_tests**, le premier vous propose une interface graphique pour lancer des calculs, voir l'exécution des calculateurs et récupérer les résultats. Le second effectue une série de tests afin de vous aider à trouver des erreurs dans vos implémentations des différentes étapes. Les sources spécifiques à ce laboratoire se trouvent dans `src` et sont partagées pour les deux projets.

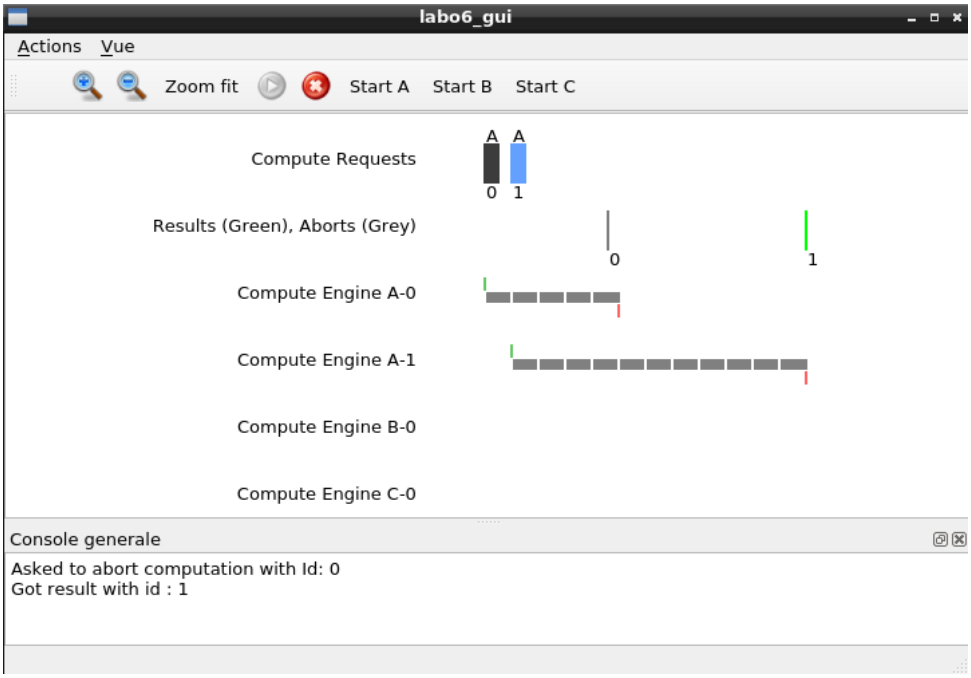
Les tests sont là pour vous aider mais ne sont malheureusement pas exhaustifs, le fait que tous les tests passent est un bon signe mais n'est pas une garantie que votre implémentation est totalement fonctionnelle. Au contraire si certains test ne passent pas, cela indique qu'il y a un problème avec votre solution.

5.1 GUI

Une application graphique vous est fournie, les boutons **Start A,B,C** vous permettent de démarrer des calculs. Ceux-ci sont affichés dans la première ligne avec le type et l'id. La seconde ligne vous affiche quand les résultats sont reçus avec leur id, ceux-ci sont aussi affichés dans la console pour rendre la lisibilié de l'ordre plus facile. Finalement les lignes suivantes vous montrent les calculateurs. Un indicateur vert est affiché au démarrage d'un calcul et un rouge à la fin d'un calcul, en gris le temps écoulé entre les appels à `continueWork()`.



Il est possible de demander l'annulation d'un calcul en cliquant sur la requête correspondante, celle-ci devient alors grisée et un indicateur est placé afin de montrer quand le calcul a été annulé. On voit ci-dessous que le calculateur A-0 s'est arrêté et qu'on a pu récupérer le résultat suivant lorsque celui-ci était disponible.



Si des calculs sont lancés mais que le buffer est plein ceux-ci apparaitront gris dans compute requests et n'auront pas encore d'id, au fur à mesure que les calculs seront acceptés par le buffer ils deviendront bleu et afficheront l'id comme dans les images ci-dessus. Le bouton stop (croix) permet d'appeler la fonction `stop()` du buffer.

6 Consignes

- Ne pas créer de nouveaux fichiers.
- Modifiez le buffer dans `computationManager.h` et `ComputationManager.cpp` afin de réaliser le comportement attendu.
- Remplissez l'en-tête de ces fichiers avec vos noms.
- Vous devez travailler en équipe de deux personnes.
- Rendez votre code selon la procédure décrite dans les consignes de laboratoire.

7 Barème de correction

Documentation : conception et tests	40%
Exécution et fonctionnement	40%
Codage (style, erreurs de base, etc.)	10%
Commentaires au niveau du code	10%

Vous pouvez remplir votre rapport sous forme de journal dans lequel vous documentez étape par étape votre conception et comment vous avez testé (ou retesté) vos implémentations. Lorsque vous avancez au fil des étapes vous allez potentiellement vous retrouver à modifier les fonctions implémentées lors des étapes précédentes.

Un rapport décomposé en 4 étapes avec chacune une section conception ainsi qu'une section pour les tests est attendu, pas besoin de répéter la donnée dans une introduction. Vous pouvez ajouter une conclusion si vous le souhaitez.