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

Laboratoire n°06

Producteur-Consommateur pour calcul différé (Hoare)

Département : TIC

Unité d'enseignement PCO

Auteur: **Alexandre Iorio & Colin Jaques**

Professeur: **Thoma Yann**

Assistant : **Da rocha Carvalho Bruno**

Salle de labo : **A01-b**

Date : **14.01.2024**

1. Introduction

1.1 Contexte

Ce rapport documente notre travail sur le laboratoire de Programmation concurrente consacré à la mise en place d'un système producteur-consommateur avec un moniteur de Hoare. L'objectif était de développer un buffer partagé pour gérer des calculs concurrents et différés, offrant ainsi une compréhension approfondie des moniteurs de Hoare et de la synchronisation des threads.

Le défi principal résidait dans la coordination efficace entre le thread producteur (client soumettant des calculs) et consommateurs (calculateurs exécutant les calculs), tout en garantissant l'intégrité des données et la gestion correcte des conditions d'attente et d'arrêt.

2. Implémentation détaillée

Afin de simplifier la compréhension de notre implémentation, nous allons détailler notre implémentation pour chaque étape du laboratoire.

2.1. Étape 1 : distribution des calculs

1. Structure générale : Le but de cette partie est d'implémenter la méthode `requestComputation(Computation)` qui permet de soumettre un calcul ainsi que la méthode `getWork(ComputationType)` qui permet de récupérer un calcul à effectuer. L'idée était donc d'implémenter le buffer partagé.

2. Structure du buffer Le buffer partagé doit permettre de stocker les requêtes de calculs. Les requêtes peuvent être de type différent, dans notre cas, `A`, `B` ou `C`. Les requêtes doivent ensuite être exécutées dans l'ordre d'arrivée. Afin de pouvoir représenter le problème correctement, nous avons décidé, dans un premier temps, d'utiliser un `std::array<std::queue<Request>, 3>` pour représenter le buffer. Chaque queue représente un type de calcul.

3. Gestion des requêtes Une fois notre structure de donnée définie, il a fallu réaliser le code concurrent permettant de gérer les requêtes. Pour cela, il a fallu créer deux conditions par type de calcul. L'une permettant d'indiquer que le buffer est plein et l'autre permettant d'indiquer un buffer vide . Sachant qu'il y a 3 types de calculs, il y a donc 6 conditions au total. Les conditions sont stockées dans deux `std::array<Condition, 3>`, `fulls` et `empties`.

Ensuite la logique est la même que pour une implémentation simple du producteur/consommateur. Lors de l'appel à `requestComputation`, le client va vérifier si le buffer est plein. Si c'est le cas, il va attendre sur la condition `fulls[type]`. Si le buffer n'est pas plein, il va ajouter la requête dans le buffer et notifier les calculateurs en attente sur la condition `empties[type]`.

Lors de l'appel à `getWork()`, le calculateur va vérifier si le buffer est vide. Si c'est le cas, il va attendre sur la condition `empties[type]`. Si le buffer n'est pas vide, il va récupérer la requête et notifier les clients en attente sur la condition `fulls[type]`.

2.2. Étape 2 : gestion des résultats

1. Structure générale : Le but de cette partie est d'implémenter la méthode `getNextResult` qui permet de récupérer le résultat d'un calcul ainsi que la méthode `provideResult(Result)` qui permet de fournir un résultat. Les résultats doivent arriver dans l'ordre pour le client. Il faut donc réfléchir à une solution de réordonnancement des résultats.

2. Structure de données : En parallèle, nous avons crée une `list<int> computationsId` qui permet de stocker les ids des calculs pour lesquels aucun résultat n'a encore été récupéré. Cela nous permet de savoir si le premier résultat de la liste correspond bien au premier calcul pour lequel aucun résultat n'a encore été récupéré.

Nous avons décidé de stocker tous les résultats dans la même structure de données. Pour cela nous avons dans un premier temps utilisé une `std::queue<Result>`. Mais nous nous sommes ensuite rendu compte que cela n'était pas idéal. En effet, sachant que les résultats doivent arriver dans l'ordre de lancement de la requête et pas simplement dans leur ordre de fin d'exécution, il faut pouvoir ordonner les résultats dans le bon ordre. Pour cela, nous avons remplacé notre `std::queue<Result>` par une `std::list<result>`. Cela nous permet, lors de l'insertion des résultats, de les insérer dans le bon ordre.

3. Gestion de la concurrence : Pour les résultats, nous avons utilisé une `Conditon resultAvailable` qui permet de notifier le client en attente de résultats. Lors de l'appel à `getNextResult`, le client va vérifier si le buffer est vide et si l'id du premier résultat de la liste correspond bien à l'id de la première requête pour laquelle aucun résultat n'a encore été récupéré. Si le buffer est vide ou si le premier résultat ne correspond pas, il va attendre sur la condition `resultAvailable`. Si le buffer n'est pas vide,il va récupérer le résultat et le retirer de la list des résultats. Lorsque les calculateurs ont terminé un calcul,ils vont appeler la méthode `provideResul(Result)`. Cette méthode va ajouter le résultat au bon endroit dans la liste des résultats. Si le résultat inséré correspond à l'id du premier élément de `computatuionsId`, il va notifier le client en attente sur la condition`resultAvailable`.

2.3 Étape 3 : gestion des annulations

1. Structure générale : Le but de cette partie est d'implémenter la méthode `abortComputation(int id)` qui permet d'annuler un calcul en attente ou déjà en cours d'exécution. L'implémentation de la méthode `bool continueWork(int id)` permettant de savoir si un calcul doit être annulé ou non est également nécessaire.

2. Structure de données : L'implémentation de ces fonctionnalités n'a pas nécessité d'ajout de structure de données. Néanmoins, afin de pouvoir supprimer des requêtes du buffer, nous avons décidé de remplacer notre `std::array<std::queue<Request>, 3>` par un `std::array<std::list<Request>, 3>`. Cela nous permet de facilement supprimer des éléments au milieu de la liste.

3. implémentation : Lors de l'appel à `abortComputation(int id)`, des éléments faisant référence à la requête `id` peuvent se trouver à trois potentiels endroits. Dans la liste des requêtes à effectuer `buffers`, dans la liste des computation ids `computationsId` ainsi que dans la liste des résultats `results`.

Si la requête `id` se trouve dans la liste des requêtes à effectuer, nous la supprimons du buffer puis nous faisons un appel à signal sur la condition `fulls[type]` afin de notifier un potentiel client en attente sur cette condition.

Si la requête `id` se trouve dans la liste des computation ids, nous la supprimons de la liste.

Si la requête `id` se trouve dans la liste des résultats, nous la supprimons de la liste. Puis si le résultat `id` est le prochain résultat attendu, nous faisons un appel à signal sur la condition `resultAvailable` afin de notifier

un potentiel client en attente sur cette condition.

Lors de l'appel à `continueWork(int id)`, nous vérifions si la requête `id` se trouve dans la liste des computation ids. Si c'est le cas, nous retournons `true`, sinon nous retournons `false`.

2.3 Étape 4 : gestion de la terminaison

1. Structure générale : Le but de cette partie est d'implémenter la méthode `stop()` qui permet d'arrêter le système. L'implémentation de cette méthode doit permettre de terminer tous les threads en cours d'exécution.

2. implémentation : Lors de l'appel à `stop()`, nous mettons un booléen `isStopRequested` à `true`. Nous faisons un appel à signal sur toutes les conditions `fulls` et `empties` afin de notifier tous les threads en attente sur ces conditions. Nous faisons un appel à signal sur la condition `resultAvailable` afin de notifier le potentiel client en attente sur cette condition.

Afin de pouvoir gérer les cas d'arrête, il a fallu modifier nos appels au wait afin d'y contrôler si un arrêt de l'application est demandé. Nous avons remplacé tous nos appels à des `wait()` par des `waitAndManageStop(Condition&)`.

La méthode `waitAndManageStop(Condition&)` est une méthode qui permet d'effectuer un `wait()` sur une condition tout en gérant les cas d'arrêt. Lors de l'appel à cette méthode, nous commençons par vérifier si un arrêt de l'application est demandé. Si c'est le cas, nous sortons du moniteur puis nous faisons un appel à `throwStopException`. Sinon, nous faisons un appel à wait sur la condition passée en paramètre. Après le wait, lorsqu'un signal à eu lieu, nous revérifions la condition d'arrêt. Si un arrêt est demandé, nous faisons un signal sur la condition afin de réveiller les threads en cascade, nous sortons du moniteur puis nous faisons un appel à `throwStopException`. Sinon, nous continuons l'exécution de la méthode.

4. Tests et Résultats

En plus de la série de tests unitaires fournie, des tests approfondis ont été menés pour valider notre implémentation. Les cas de test incluaient différents scénarios de soumission, de calculs, d'annulation de calculs et de gestion de la terminaison du buffer. Les résultats ont montré que :

Le buffer gère correctement les requêtes de calcul et les résultats dans l'ordre attendu. Les annulations et les arrêts du système sont traités efficacement, sans laisser de threads bloqués. Les calculateurs répondent correctement aux requêtes de travail et aux demandes d'arrêt. Ces tests ont validé la robustesse de notre système dans diverses conditions de concurrence et de synchronisation.

5. Conclusion

Ce laboratoire nous a permis de développer une compréhension approfondie des moniteurs de Hoare et de la synchronisation en programmation concurrente. La mise en œuvre d'un système producteur-consommateur pour le calcul différé a souligné l'importance d'une conception et d'une implémentation soignées pour gérer efficacement la concurrence et l'ordre des opérations dans un environnement multithread.