3259.1 Paradigmes de program. avancés II– Rapport technique – ISC3il-b

|  |
| --- |
| **Laboratoire 2** |

|  |
| --- |
| Étudiants participant à ce travail :  **Nicolas Aubert, ISC3il-b**  **Théo Vuilliomenet, ISC3il-b**  Présenté à :  **Aïcha Rizzotti**  **Julien Senn**  Restitution du rapport : **12.05.2023**  Période : **2023**  École : **HE-Arc, Neuchâtel** |

**Mesure de performances d’un modèle producteur / consommateur en Java**

Table des matières

[1 - Glossaire 2](#_Toc134178177)

[2 - Introduction 3](#_Toc134178178)

[2.1 - Contexte 3](#_Toc134178179)

[3 - Présentation du framework 4](#_Toc134178180)

[3.1 - Classe Buffer 4](#_Toc134178181)

[3.2 - Classe Producer 4](#_Toc134178182)

[3.3 - Classe Consumer 4](#_Toc134178183)

[4 - Résultats 5](#_Toc134178184)

[4.1 - Acquisition de données 5](#_Toc134178185)

[4.2 - 500 producteurs, 500 consommateurs 6](#_Toc134178186)

[4.3 - 200 producteurs, 800 consommateurs 7](#_Toc134178187)

[4.4 - 800 producteurs, 200 consommateurs 8](#_Toc134178188)

[4.5 - Interprétation des résultats 9](#_Toc134178189)

[4.5.1 - Répartition égale entre producteurs et consommateurs 9](#_Toc134178190)

[4.5.2 - Répartition disparate entre producteurs et consommateurs 9](#_Toc134178191)

[5 - Limitations et perspectives 10](#_Toc134178192)

[6 - Conclusion 11](#_Toc134178193)

[7 - Annexes I](#_Toc134178194)

[7.1 - Bibliographies et références I](#_Toc134178195)

[7.1.1 - Sites Web I](#_Toc134178196)

1. Glossaire

|  |  |
| --- | --- |
| **Buffer** | Zone de stockage temporaire pour les données entre un producteur et un consommateur. |
| **Concurrence** | Situation où plusieurs processus ou threads sont en compétition pour l'accès aux ressources d'un système. |
| **Débit** | Nombre d'actions exécutées, de résultats d'actions exécutées ou de résultats produits par unité de temps. |
| **Latence** | Temps nécessaire pour effectuer une action ou produire un résultat. |
| **Log** | Enregistrement des actions qui se produisent dans le programme. |
| **Modèle producteur / consommateur** | Modèle de communication entre deux parties où l'une produit des données et l'autre les consomme. |
| **Opération atomique** | Une opération est atomique si elle est exécutée entièrement ou pas du tout, sans être interrompue par d'autres opérations en cours d'exécution sur d'autres threads. Une opération atomique garantit l'intégrité des données, même en présence de threads concurrents. |
| **Performances** | Mesure de la quantité de travail qu'une application peut traiter dans une période donnée et de la vitesse à laquelle elle peut traiter une unité de travail. |
| **Thread** | Unité d'exécution dans un programme qui permet une exécution parallèle de plusieurs tâches. |
| **Thread safe** | Une classe ou une méthode est thread safe si elle peut être utilisée simultanément par plusieurs threads sans causer de problèmes de concurrence ou d'incohérence de données. |

1. Introduction
   1. Contexte

L'objectif de ce projet est de réaliser un programme de monitoring de la concurrence en utilisant le modèle producteur-consommateur. Nous allons nous intéresser à la mise en place d'un framework qui nous fournira des informations utiles sur les performances d'un programme.

Pour ce faire, nous implémenterons un exemple simple du modèle producteurs-consommateurs, en Java, en prenant en compte des paramètres importants, tels que :

* La taille du buffer,
* Le nombre de producteurs,
* & le nombre de consommateurs.

1. Présentation du framework
   1. Classe Buffer

La classe Buffer est une implémentation du modèle producteur-consommateur ; elle est utilisée pour stocker des données produites par un producteur jusqu'à ce qu'elles soient prêtes à être consommées par un consommateur. Le fonctionnement de cette classe repose sur l'utilisation d'une blocking queue, qui est une structure de données qui permet aux threads de se bloquer lorsqu'ils essaient d'ajouter ou de retirer des éléments à partir de la queue vide ou pleine. Une blockingqueue est également thread safe, cela signifie qu’une ressource peut être uniquement modifiée de façon atomique.

Le constructeur de la classe prend en entrée la taille du buffer, qui est utilisée pour initialiser la blocking queue. Si aucune taille n'est spécifiée, la taille par défaut est de 100.

La méthode push permet d'ajouter des données au buffer. Cette méthode utilise la méthode put de la blocking queue, qui bloque le thread si la queue est pleine jusqu'à ce qu'il y ait de la place pour ajouter la donnée.

La méthode take permet de retirer des données du buffer. Cette méthode utilise la méthode take de la blocking queue, qui bloque le thread si la queue est vide jusqu'à ce qu'une donnée soit disponible pour être retirée.

La méthode getSize permet d'obtenir la taille du buffer.

* 1. Classe Producer

La classe Producer est au cœur de notre programme de monitoring de la concurrence. Elle représente un producteur de données qui a pour rôle de générer des ressources et de les stocker dans le buffer. La classe possède deux attributs principaux : le buffer dans lequel seront stockées les ressources générées et le nombre d'accès que le producteur doit effectuer.

La classe implémente l'interface Callable<Integer> plutôt que Runnable, car elle doit renvoyer le nombre d'accès effectués. Pour cela, elle définit une méthode call() qui contient une boucle infinie. À chaque itération de la boucle, le producteur génère une ressource et tente de la stocker dans le buffer à l'aide de la méthode push(). Si le buffer est plein, la méthode push bloquera le producteur jusqu'à ce qu'une place se libère dans le buffer. Le producteur incrémente également le compteur d'accès à chaque itération de la boucle.

* 1. Classe Consumer

La classe Consumer est l'une des deux classes principales de notre projet, représentant un consommateur. Son rôle est de récupérer les données stockées dans le buffer et de les traiter. Cette classe fonctionne exactement de la même manière que la classe Producer, sauf qu'elle lit une donnée au lieu d'en écrire dans la méthode call(). Elle implémente également l'interface Callable<Integer> afin de permettre au programme de récupérer le nombre d'accès effectués et ainsi des statistiques par la suite.

1. Résultats
   1. Acquisition de données

Afin d’obtenir des données et de pouvoir visualiser l’impact de la modification de la taille du buffer sur les performances, nous avons utilisé notre framework et démarré une batterie de simulation avec des tailles de buffer différentes.

Il est important de préciser que nous avons commencé par effectuer plusieurs simulations avec les mêmes paramètres afin d’obtenir une stabilité dans les résultats que nous allions produire par la suite. On peut comparer cette étape à une étape de calibration qui minimise l’impact de l’aléatoire dans les résultats obtenu. Le résultat de la calibration nous indique qu’il faut un très grand nombre d’accès à la ressource partagée pour qu’il n’y ait plus d’aléatoire dans les résultats. Dans notre cas, nous avons décidé de fixer le nombre d’accès à 500'000 pour les consommateurs et les producteurs. Les résultats souffrent toujours d’une légère part d’aléatoire, mais au-delà de 500'000 accès la simulation devient très lente à réaliser.

Une fois les simulations terminées, nous avons copié toutes les données dans des tableaux et aussi réalisé des graphes par le biais d’Excel.

Cette opération a été réalisée trois fois afin de visualiser l’impact de la répartition entre producteurs, consommateurs.

* 1. 500 producteurs, 500 consommateurs

Une image contenant table

Description générée automatiquement

* 1. 200 producteurs, 800 consommateurs

Une image contenant table

Description générée automatiquement

* 1. 800 producteurs, 200 consommateurs

Une image contenant table

Description générée automatiquement

* 1. Interprétation des résultats
     1. Répartition égale entre producteurs et consommateurs

On constate que la progression du débit et de la latence est asymptotique en fonction de la taille du buffer. Ça semble tout à fait cohérent étant donné que l’apport d’une place dans la taille du buffer a un impact énorme lorsque tous les producteurs/consommateurs doivent se partager un nombre limité de places pour réaliser leurs actions.

De plus, lorsque la taille du buffer est déjà conséquente et suffisante pour que la production et consommation aient un bon rendement, il est presque inutile de rajouter de la place.

* + 1. Répartition disparate entre producteurs et consommateurs

Étant donné qu’il n’y a pas de notion de priorité entre les producteurs et les consommateurs, les résultats des chapitres 4.3 et 4.4 sont similaires. En effet, ils sont les deux bloquants et doivent effectuer une action sur la ressource partagée, ce qui modifie son état.

Il aurait été intéressant d’avoir un modèle lecteurs-rédacteurs et d’y ajouter la notion de priorité entre la lecture et la rédaction de ressources partagées. À ce moment-là, nous aurions vu une différence conséquente lors d’une répartition inégale entre lecteurs et rédacteurs.

1. Limitations et perspectives

Le projet présenté dans ce rapport comporte certaines limites qui peuvent être adressées pour améliorer sa pertinence et son efficacité. Tout d'abord, l'absence d'interface graphique peut rendre l'utilisation de l'application difficile pour les néophytes. Il pourrait être envisagé de développer une interface graphique plus conviviale en utilisant des outils tels que Swing, afin de faciliter l'interaction avec l'application.

Une autre limitation est que l'application actuelle ne prend pas en charge les variantes du modèle producteur / consommateur telles que la priorité aux producteurs ou la priorité aux consommateurs. L'ajout de ces fonctionnalités permettrait de rendre l'application plus polyvalente et plus adaptée aux différents scénarios de producteurs / consommateurs.

Enfin, il est important de noter que JConsole n'a pas été utilisé dans le cadre de ce projet, mais il pourrait être intéressant de l'intégrer à l'avenir. En effet, JConsole permet de collecter des informations détaillées sur les performances de la JVM, telles que la consommation de mémoire, l'utilisation du processeur et bien plus encore. En utilisant JConsole, il serait possible de surveiller l'application de manière plus précise et d'optimiser les performances de manière plus fine. Cette amélioration pourrait être envisagée dans le cadre d'une perspective future pour le projet.

1. Conclusion

En conclusion, le framework développé est parfaitement fonctionnel et permet de mesurer les performances d'un modèle producteur/consommateur en Java. Les résultats obtenus grâce à ce benchmark peuvent permettre de déterminer le nombre optimal de producteurs et de consommateurs ainsi que la taille de buffer la plus efficace pour un système donné.

L'application offre également la possibilité de définir plusieurs paramètres au lancement du programme, tels que le nombre de producteurs et de consommateurs, la taille du buffer et le nombre d'actions à effectuer par les producteurs et les consommateurs. Ces paramètres permettent une personnalisation fine des tests de performance, offrant ainsi une grande flexibilité.

Cependant, il est important de noter que cette application est destinée à être utilisée comme un outil de benchmarking uniquement, sans aucune interaction avec l'utilisateur. Pour des projets nécessitant une interaction utilisateur, une interface plus conviviale devrait être développée.

1. Annexes
   1. Bibliographies et références
      1. Sites Web

Using JConsole - Java SE Monitoring and Management

Guide. (2006, October 1). <https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html>

Baeldung. (2021, December 21). *Guide to java.util.concurrent.BlockingQueue*. Baeldung. <https://www.baeldung.com/java-blocking-queue>

*Executor (Java Platform SE 8 )*. (2023, April 5). <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html>

Baeldung. (2022, November 24). *A Guide to the Java ExecutorService*. Baeldung. <https://www.baeldung.com/java-executor-service-tutorial>