PROJET DE PROGRAMMATION SYSTÈME AVANCÉE

Ordonnanceur Par Work Stealing

DEDEOGLU Dilara
DIAMANT Alexandre
Promotion 2023/2024

Professeur Encadrant :
Juliusz Chroboczek

07/05/2024

Table des matières

1 Introduction

Dans le contexte actuel de l'informatique, où les tâches sont de plus en plus complexes et gourmandes en ressources, il devient crucial d'exploiter au maximum les capacités des processeurs multi-cœurs. Cependant, gérer efficacement plusieurs processus qui fonctionnent en parallèle présente des défis considérables. Une gestion optimale nécessite l'utilisation de techniques d'ordonnancement avancées pour répartir les charges de travail de manière efficace entre les différents cœurs.

Dans ce projet, nous avons exploré deux méthodes d'ordonnancement distinctes : le work stealing et le LIFO (Last In, First Out). Ces techniques visent à améliorer l'efficacité des programmes multithreads en optimisant l'utilisation des ressources disponibles. À travers une série de tests, nous avons évalué ces ordonnanceurs selon plusieurs critères clés qui impactent directement la performance des programmes. Cette analyse nous permet de déterminer quelle méthode est la plus efficace dans divers scénarios d'utilisation.

Ce rapport présente les résultats de notre étude, en décrivant les avantages et les limites de chaque méthode d'ordonnancement testée, dans le but d'identifier l'approche la plus performante pour gérer des tâches complexes en parallèle.

2 Présentation de l'Ordonnanceur par LIFO

L'ordonnanceur LIFO, qui utilise une pile pour gérer les tâches, est une méthode simple mais efficace dans certains contextes. Ce type d'ordonnanceur ajoute et retire les tâches au sommet de la pile, suivant le principe du dernier arrivé, premier servi. Cette approche présente des avantages significatifs en termes de simplicité de mise en œuvre et de performance dans des scénarios à faible concurrence.

2.1 Fonctionnement de la Pile LIFO

Dans l'ordonnanceur LIFO, chaque tâche ajoutée en dernier est la première à être traitée. Ce modèle est particulièrement adapté pour des tâches dont la gestion ne nécessite pas un ordonnancement complexe ou pour des applications où les tâches les plus récentes sont prioritaires.

2.2 Arrêt du Programme

L'ordonnanceur LIFO arrête le programme lorsque la pile des tâches est vide et que tous les threads attendent. Cela signifie qu'il n'y a plus de tâches à effectuer et que le programme peut se terminer proprement.

2.3 Limitation de l'Ordonnanceur LIFO

Accès à la Pile : Un gros problème avec la pile LIFO est que les tâches ne peuvent pas être retirées de la pile par plusieurs threads en même temps. Puisque l'accès à la pile doit être contrôlé pour éviter que deux threads ne prennent ou ajoutent des tâches en même temps, cela peut créer un ralentissement si beaucoup de threads essaient d'accéder à la pile simultanément.

3 Présentation de l'Ordonnanceur par Work Stealing

L'ordonnanceur par work stealing est une méthode avancée pour gérer les tâches dans un environnement multi-thread, c'est-à-dire quand plusieurs programmes ou parties d'un programme (threads) s'exécutent en même temps. Cette méthode est conçue pour maximiser l'utilisation des ressources du processeur et réduire le temps d'inactivité des threads. Comment fonctionne le Work Stealing?

3.1 Structure de Données : Deques

Chaque thread a sa propre deque (file à double extrémité), où il stocke les tâches qu'il doit exécuter. Une deque permet d'ajouter ou de retirer des tâches à chaque extrémité.

3.2 Exécution des Tâches

Normalement, un thread prend la tâche située à l'extrémité inférieure de sa deque et la traite. Si cette deque est vide, le thread ne reste pas inactif.

3.3 Processus de 'Vol' de Tâches

Lorsqu'un thread a fini toutes ses tâches et que sa deque est vide, il tente de 'voler' une tâche de l'extrémité supérieure de la deque d'un autre thread. Ce processus commence par choisir au hasard un autre thread et essayer de prendre une tâche de sa deque.

Chaque tâche est exécutée immédiatement après avoir été volée. Cela signifie qu'une fois qu'une tâche est retirée d'une deque et attribuée à un thread, elle ne se retrouve plus dans une position où elle pourrait être volée à nouveau. En conséquence, chaque tâche peut être volée au maximum une seule fois. Cela garantit que les tâches ne sont pas traitées plus d'une fois et évite les duplications de travail, ce qui pourrait réduire l'efficacité de l'ordonnancement.

3.4 Gestion de l'Échec du Vol

Si le premier essai de vol échoue parce que la deque de l'autre thread est également vide, le thread voleur continue à chercher en passant aux deques des autres threads jusqu'à ce qu'il trouve une tâche à exécuter ou que toutes les possibilités aient été épuisées.

4 Analyses des Résultats

4.1 Comparaison d'Exécution Sérielle et en Parallèle

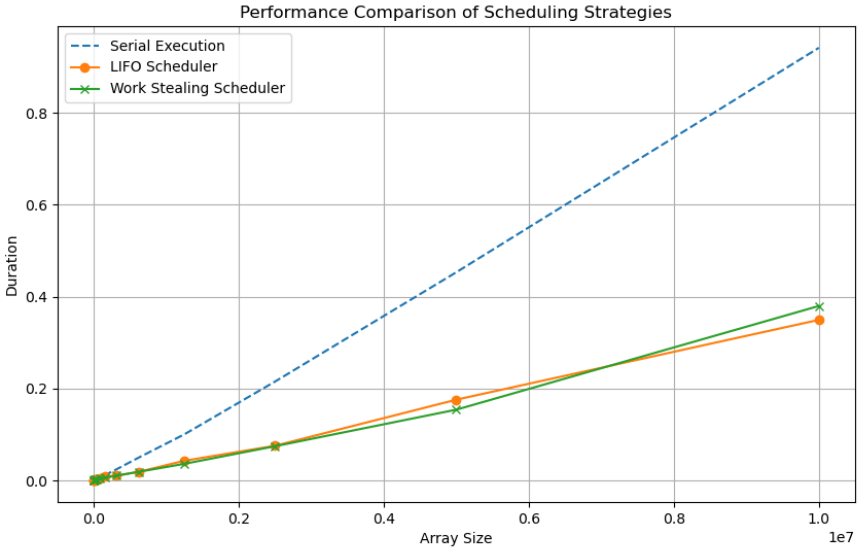

FIGURE 1 – Ce graphique présente une comparaison des durées d'exécution pour trois différentes stratégies d'ordonnancement : exécution sérielle, LIFO Scheduler, et Work Stealing Scheduler, en fonction de la taille d'un tableau traité.

Comme prévu, l'exécution sérielle montre une augmentation linéaire du temps en fonction de la taille du tableau. C'est la référence de base, avec la plus longue durée d'exécution, particulièrement visible quand la taille du tableau augmente significativement. Ce graphique démontre clairement l'avantage des ordonnanceurs parallèles, et en particulier du Work Stealing Scheduler, dans la gestion de charges de travail croissantes par rapport à l'exécution sérielle. Le Work Stealing Scheduler est particulièrement efficace pour traiter des volumes de données élevés, ce qui en fait une option préférable pour des applications nécessitant un haut degré de parallélisme et une répartition efficace des tâches. Cela confirme l'importance de choisir une stratégie d'ordonnancement adaptée aux besoins spécifiques en performance et en efficacité des systèmes modernes.

4.2 Comparaison de LIFO et Work Stealing

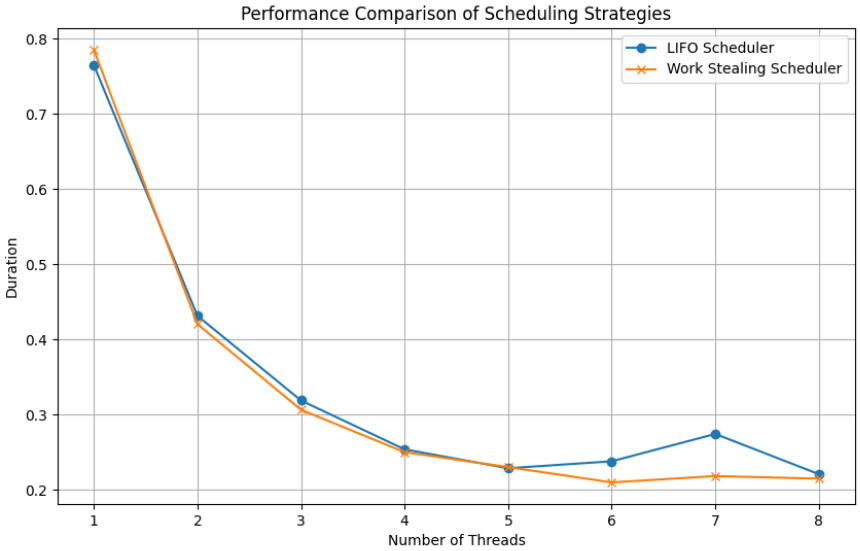

FIGURE 2 – Ce graphique montre la performance des ordonnanceurs LIFO et Work Stealing en fonction du nombre de threads utilisés.

On observe que les performances des deux ordonnanceurs s'améliorent significativement avec l'augmentation du nombre de threads. Le Work Stealing montre une légère supériorité sur le LIFO, particulièrement avec un nombre de threads supérieur, indiquant une meilleure gestion de la concurrence et un équilibrage de charge plus efficace.

4.3 Comparaison d'Exécution Normale et Optimisée

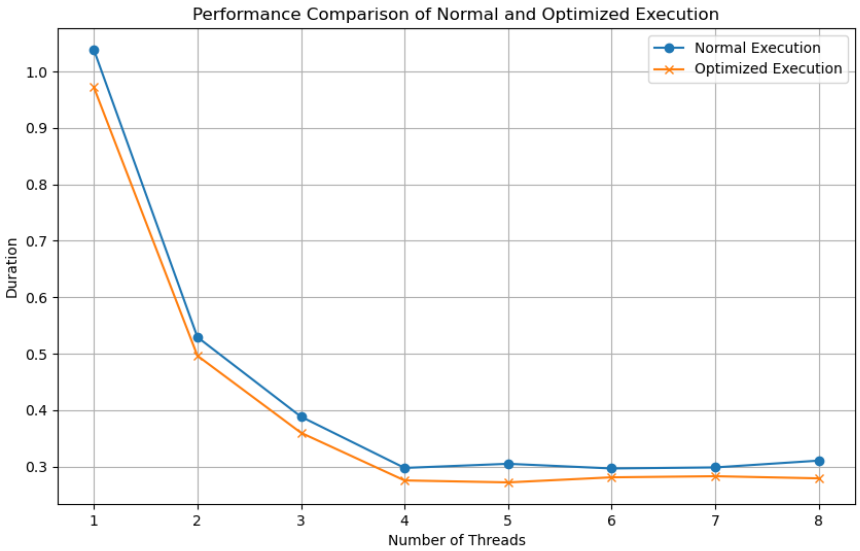

FIGURE 3 – Ce graphique compare les performances d'une exécution normale avec une exécution optimisée où un paramètre de seuil à partir duquel les sous-tâches passent d'une exécution en parallèle à une exécution séquentielle.

Lorsque le seuil est bas, les sous-tâches passent trop lentement en mode séquentiel, sous-utilisant les cœurs disponibles. À l'inverse, un seuil trop élevé peut entraîner une surcharge par le nombre excessif de threads actifs, augmentant ainsi la consommation de mémoire et les coûts de changement de contexte. Le passage d'un seuil de 128 à 1024 pour la décision de basculer en exécution séquentielle montre une réduction significative du temps d'exécution.

4.4 Comparaison du Work Stealing sur Différents Systèmes d'Exploitation

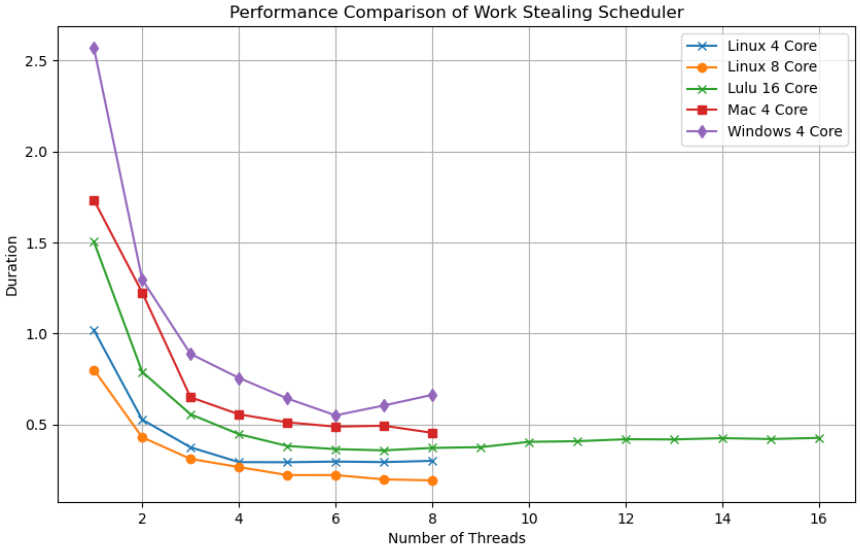

FIGURE 4 – Graphique comparant les performances du Work Stealing sur différents systèmes d'exploitation avec divers nombres de cœurs.

Les systèmes avec un plus grand nombre de cœurs (par exemple, Linux 8 Core) montrent de meilleures performances, illustrant l'efficacité du Work Stealing dans des environnements multi-cœurs. Les performances varient également significativement avec le type d'OS, ce qui peut refléter des différences dans la gestion des threads et des processus par les systèmes d'exploitation.

4.5 Comparaison du LIFO sur Différents Systèmes d'Exploitation

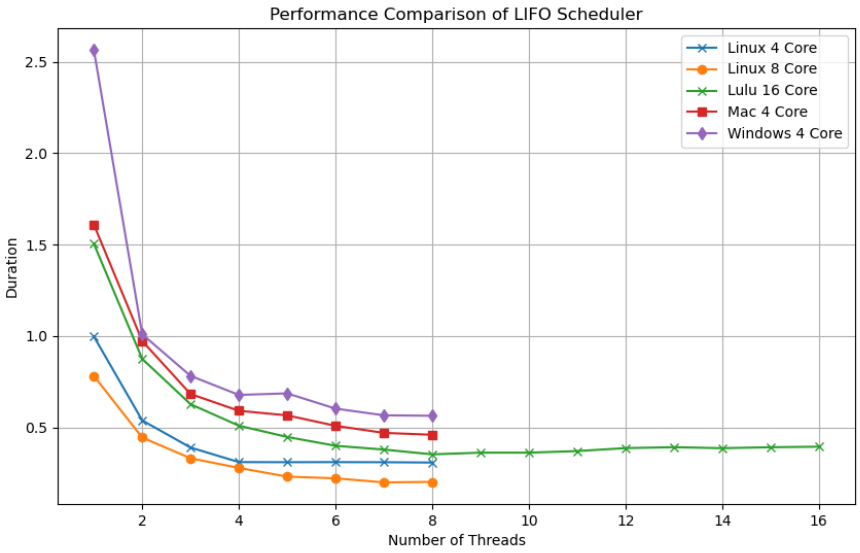

FIGURE 5 – Graphique comparant les performances du LIFO sur différents systèmes d'exploitation avec divers nombres de cœurs.

Tout comme avec le Work Stealing, le nombre de cœurs influence les performances du LIFO, mais les variations entre les systèmes d'exploitation sont moins prononcées. Cela suggère que bien que le LIFO soit moins efficace pour l'équilibrage de la charge, il est moins dépendant des spécificités de l'OS que le Work Stealing.

5 Optimisations et Expériences Futures

5.1 Impact de la Gestion de la Mémoire

Au cours de ce projet, nous avons observé que la méthode d'allocation de la mémoire avait un impact significatif sur les performances des ordonnanceurs, en particulier avec l'ordonnanceur par work stealing. Initialement, pour allouer les nœuds de données utilisés dans nos deques, nous avons employé mmap anonyme privé. En cours de projet, nous avons remplacé cette méthode par l'utilisation de malloc, ce qui a amélioré les performances en réduisant la surcharge associée à la gestion des espaces de mémoire virtuelle et en optimisant l'accès à la mémoire pour les opérations fréquentes d'enfilage et de défichage dans les deques. Cette modification a rendu l'accès aux nœuds plus rapide et moins coûteux en termes de cycles CPU, ce qui est crucial pour les opérations de vol de tâches qui exigent des réponses rapides et une synchronisation efficace entre les threads.

5.2 Implémentation de la Gestion Bloquante dans la File d'Ordonnancement

Pour améliorer l'efficacité de notre ordonnanceur et éviter le gaspillage des ressources du processeur avec une attente active, nous avons adopté un comportement bloquant pour les opérations sur la file d'attente. Cette approche est essentielle pour gérer efficacement les conditions de compétition entre les threads consommateurs et producteurs dans des scénarios de file vide ou pleine.

Nous avons utilisé des variables de condition pour bloquer les threads tentant de réaliser des opérations non valides :

— Pour les opérations de pop : Un thread consommateur qui tente de retirer une donnée d'une file vide est mis en attente jusqu'à ce qu'une donnée soit disponible.

— Pour les opérations de push : De manière symétrique, un thread producteur essayant d'ajouter une donnée à une file pleine est bloqué jusqu'à ce que de l'espace soit libéré.

Ces variables de condition sont utilisées pour suspendre les threads en attente et les réveiller uniquement lorsque l'état de la file change. Aussi, le choix judicieux entre signal et broadcast pour la gestion des threads en attente a également été déterminant, garantissant que les ressources sont utilisées de manière optimale sans surcharger le système.

5.3 Projets d'Optimisation Non Réalisés

En outre, nous avions envisagé de tester l'impact de l'utilisation des sémaphores en remplacement des mutex pour la synchronisation des threads dans les deux ordonnanceurs. Les mutex, bien qu'efficaces dans la gestion des accès exclusifs, peuvent devenir un goulot d'étranglement en présence de nombreux threads concurrents. Les sémaphores, offrant des mécanismes de signalisation plus flexibles, pourraient potentiellement améliorer la performance en réduisant les temps d'attente et en augmentant la concurrence. Cependant, en raison de contraintes de temps, nous n'avons pas pu mener cette expérience.

6 Conclusion

Cette étude a exploré et comparé en détail les performances et les caractéristiques de deux méthodes d'ordonnancement des tâches : le LIFO et le work stealing, en utilisant diverses configurations de threads et plusieurs systèmes d'exploitation. À travers une série de tests comparatifs, nous avons mis en évidence les forces et les faiblesses de chaque stratégie d'ordonnancement, soulignant leur impact significatif sur l'efficacité du traitement des tâches.

Nous avons constaté que bien que l'ordonnanceur LIFO soit simple à mettre en œuvre, il peut ne pas toujours offrir la meilleure performance en termes de gestion des ressources, surtout dans des scénarios hautement concurrentiels. En revanche, l'ordonnanceur par work stealing a démontré une capacité supérieure à adapter dynamiquement la charge de travail entre les threads, se révélant ainsi plus efficace dans des environnements où la distribution des tâches est inégale et la demande de ressources est variable.

Les optimisations apportées en termes de gestion de la mémoire et des opérations bloquantes sur les files d'ordonnancement ont également montré une amélioration notable des performances. Ces ajustements ont permis de réduire la surcharge de gestion de la mémoire et d'éviter l'inefficacité liée à l'attente active.

Finalement, les perspectives futures pour améliorer encore les ordonnanceurs comprennent l'exploration d'autres mécanismes de synchronisation, comme les sémaphores, qui pourraient offrir des avantages supplémentaires en termes de gestion de la concurrence. La poursuite de ces recherches contribuera à élaborer des solutions d'ordonnancement encore plus robustes et efficaces pour les systèmes informatiques modernes, où la performance et l'efficacité sont primordiales.

7 Références

1. Chroboczek, J., “Programmation système”, Mars 2024, <https ://www.irif.fr/ jch/enseignement/program
systeme.pdf >.
2. GeeksforGeeks. (n.d.). Introduction to Stack Data Structure and Algorithm Tutorials. <https ://www.geek
to-stack-data-structure-and-algorithm-tutorials/>.