UCLouvain

École polytechnique de Louvain

Delacroix Maxime - Louis Hugolin

LINFO1252 : Systèmes informatiques - Projet 1

7 décembre 2022

Table des matières

1 Introduction

Dans le cadre de ce premier projet, l'objectif était d'implémenter en multi-thread différents problèmes informatiques bien connus à l'aide de la librairie *POSIX*. Dans la seconde partie de ce projet, l'objectif était d'implémenter un équivalent à la librairie *POSIX* à l'aide de l'opération atomique *xchg* et d'observer les conséquences de son utilisation.

2 Etude des resultats

2.1 Tâche 1

Introduction

L'objectif de la tâche 1 consistait à implémenter 3 problèmes bien connus. Les problèmes étant celui des philosophes, le producteur-consommateur et le problème des lecteurs-écrivains. Ces trois problèmes comportent des enjeux différents, tels que la gestion de sections critiques, la synchronisation dans un "buffer" et le partage de ressources.

Etude Philosophe

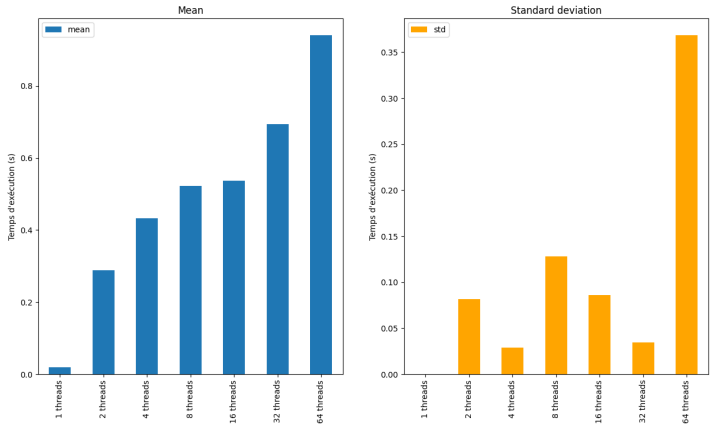

FIGURE 2.1 – Moyenne et variance du temps d'exécution du problème des philosophes en fonction du nombre de threads

Sur le graphique on peut voir que le temps d'exécution de philosophe augmente considérablement avec le nombre de thread. Cela est du au fait que les philosophes bloquent l'accès à deux mutex dans sa section critique (ici presque immédiate), ce qui a pour effet de bloquer tous les threads qui veulent accéder à ces deux variables. Ainsi plus le nombre de thread augmente, plus la probabilité que le plusieurs threads veulent accéder à une même variable augmente. On peut voir ce phénomène sur le graphique, lorsqu'il y a 64 threads le temps d'exécution explose. Inversement lorsque le nombre de threads est égal à 1, le philosophe n'a aucun conflit et l'exécution est donc presque instantanée. L'écart type évolue significativement avec le nombre de threads, ce qui vient du faite l'augmentation du nombre de threads augmente la différence entre le chemin le plus long et le chemin le plus court. Ainsi il est possible d'avoir de grande disparité entre les mesures.

Etude producteurs-consommateurs

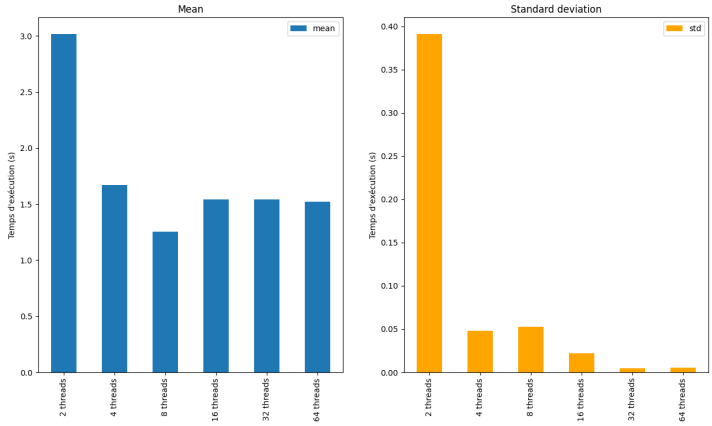

FIGURE 2.2 – Moyenne et variance du temps d'exécution du problème producteurs-consommateurs en fonction du nombre de threads

Contrairement aux philosophes, le temps d'exécution de l'algorithme producteurs-consommateurs diminue avec le nombre de threads alloués. Pour rappel l'algorithme de producteurs-consommateurs consiste à synchroniser des threads producteurs qui produisent des données dans une structure de données, et des threads consommateurs qui consomment les données produites par les producteurs.

On peut voir que le temps d'exécution diminue avec le nombre de threads. Cela est dû au fait que le nombre de threads travaillant simultanément augmente, et donc diminue le charge de travail individuelle. Dans notre cas de figure nous avons un buffer de 8 places, ce qui veut dire 8 producteurs et 8 consommateurs en même temps au maximum. Cette limitation de notre programme explique le plateau obtenue une fois que nous arrivons à 16 threads, en effet, au delà de 16 threads, les threads supplémentaires n'accélère plus le processus puisque la sémantique leur interdit de fonctionner tous simultanément. L'écart-type n'évolue pas significativement avec le nombre de threads, et il est assez petit pour être négligeable.

Etude lecteurs-écrivains

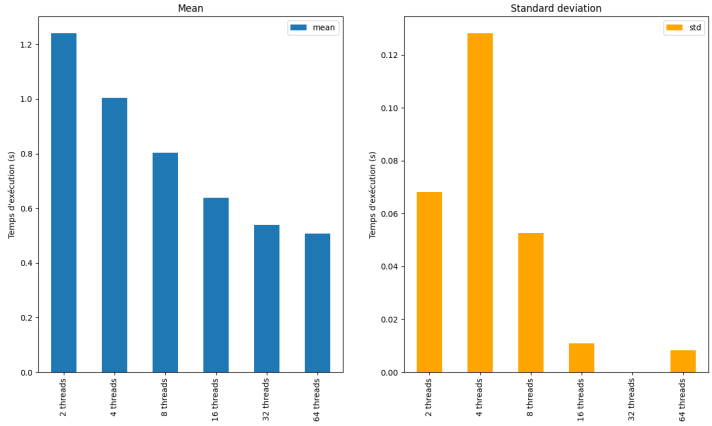

FIGURE 2.3 – Moyenne et variance du temps d'exécution du problème des lecteurs-écrivains en fonction du nombre de threads

L'algorithme lecteurs-écrivains permet la synchronisation de threads qui lisent et écrivent dans une base de données. Le nombre de lecteurs simultané n'as pas de limite mais lorsqu'un écrivain veut accéder à la section critique, plus aucun lecteur n'est autorisé et les lecteurs occupés terminent leur opération. L'écrivain peut ainsi utiliser la sections critique seul jusqu'à ce qu'il ait finis.

On voit que plus le nombre de threads augmente plus le temps d'exécution est cours, ce qui est du au fait que le travail est exécuté simultanément (pour les lecteurs). L'écart type n'évolue pas significativement avec le nombre de threads, et il est assez petit pour être négligeable.

2.2 Tâche 2

Introduction

La première partie de cette deuxième tâche était d'implémenter la synchronisation par attente active en utilisant des opérations atomiques. Lors de la deuxième partie, la tâche était d'analyser et de comparer la performance de l'attente active avec celle des primitives de synchronisation POSIX pour les programmes réalisés en première partie.

Partie 1 : Comparaison TAS [1] et TATAS [2]

Pour rappel, la différence entre l'algorithme *TAS* et *TATAS* se situe lors de l'attente active. L'opération atomique *xchg* est coûteuse car elle arrête le bus CPU à chacune des ses exécutions, l'idée de *TATAS* est d'effectuer l'opération atomique seulement lorsqu'elle a pu lire que le verrou était "ouvert". Cela a pour conséquence de réduire le nombre d'appel à *xchg* tout en gardant la même finalité mais avec moins de consommation.

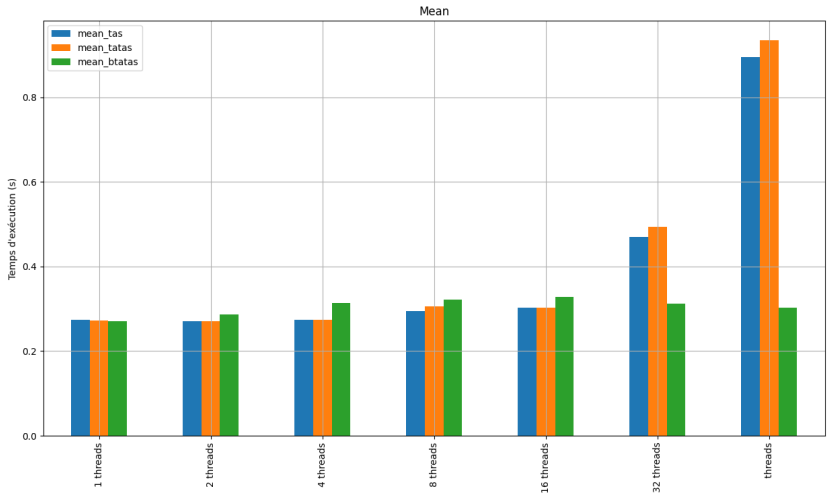

FIGURE 2.4 – Temps d'exécution de *TAS*, *TATAS* et *BTATAS* en fonction du nombre de threads

On observe sur le graphique qu'avec peu de thread l'avantage de *TATAS* est peu significatif, cela s'explique par le fait qu'il y a peu de thread en attente active qui utilise en permanence l'opération *xchg* qui ralentit l'exécution. Lorsqu'il y a un grand nombre de thread utilisé, il y a plus de thread qui pourrait être en attente active donc ils effectueraient tous l'opération coûteuse *xchg* ce qui explique la différence de temps d'exécution. De plus, pour l'algorithme *TATAS*, tous les threads en attente active sont dans une boucle "while" où ils lisent la valeur du verrou. En conclusion, la lecture de la valeur du verrou ralenti moins que l'opération atomique *xchg* qui bloque le bus du CPU à chaque exécution, et donc bloque tous les autres threads.

1. L'acronyme TAS est utilisé pour désigner l'algorithme *test-and-set*
2. L'acronyme TATAS est utilisé pour désigner l'algorithme *test-and-test-and-set*

On remarque que l'implémentation d'un "cooldown" entre chaque lecture du verrou permet diminuer les ressources consommées au fil du temps, en admettant qu'un verrou fermé depuis longtemps est susceptible de le rester encore longtemps. On comprend que cette approximation est proche de la réalité étant donné le gain de temps. Lors du test, la meilleure borne trouvée est de 1 millisecondes pour l'attente maximum

Partie 2 : Comparaison des performances

Comparaisons du problème des philosophes

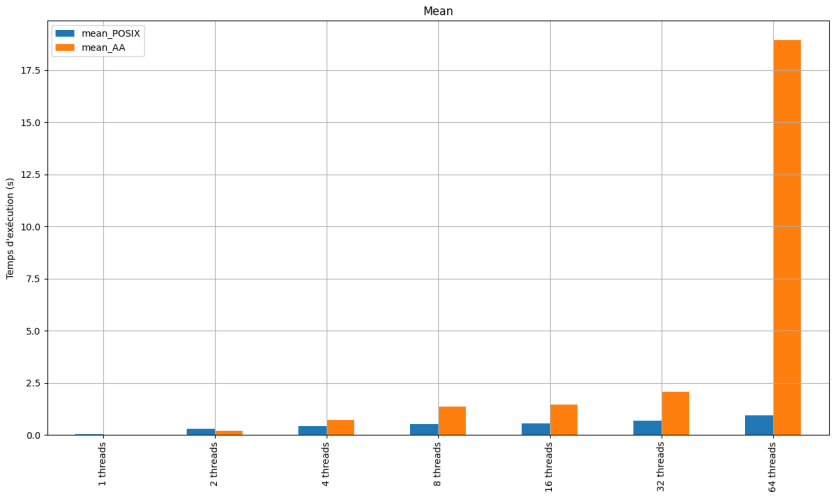

FIGURE 2.5 – Moyenne et variation du temps d'exécution du problème des philosophes avec *POSIX* et avec opération atomique en fonction du nombre de threads

On peut observer une grande différence de temps d'exécution entre les deux méthodes, cela s'explique par la nature du problème. En effet, le problème des philosophes utilise de nombreuses fois l'opération *lock* et *unlock* étant donné que chaque baguette est protégée par un verrou. L'utilisation excessive de l'opération *xchg* qui est très coûteuse en temps augmente le temps d'exécution de manière significative.

Comparaisons des problèmes producteurs-consommateurs et lecteurs-écrivains

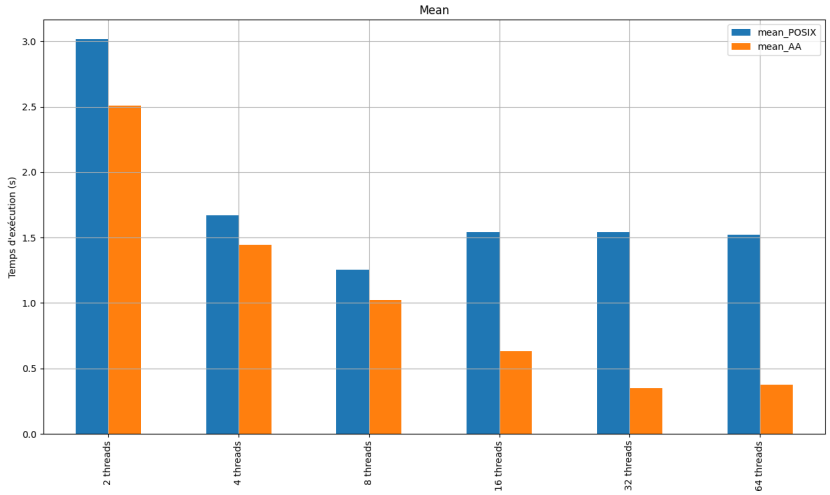

FIGURE 2.6 – Moyenne et variance du temps d'exécution du problème producteurs-consommateurs avec *POSIX* et avec opération atomique en fonction du nombre de threads

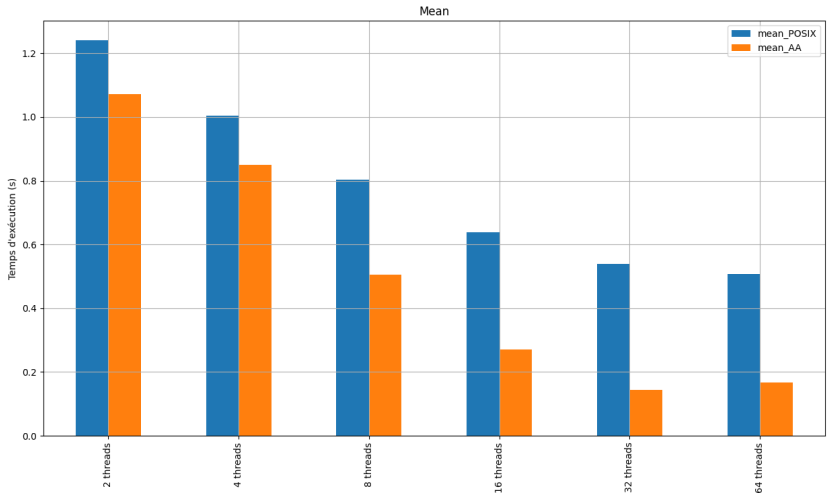

FIGURE 2.7 – Moyenne et variance du temps d'exécution du problème lecteurs-écrivains avec *POSIX* et avec opération atomique en fonction du nombre de threads

Pour les 2 derniers problèmes, on peut tirer les mêmes conclusions. L'utilisation de l'attente active permet de réduire considérablement le temps d'execution, en effet, ces problèmes sont bien adaptés à cette solution puisque que les threads travaillent sur une structure de données partagée qui est le buffer. De plus, l'attente à la section critique est bien plus longue que la durée de la section critique donc les opérations atomiques sont un bon moyens d'accélérer l'exécution.

3 Conclusion

Au travers de ce projet nous avons pu identifier quels types de problèmes étaient plus efficaces avec la libraire *POSIX* ou avec le mécanisme d'attente active. L'attente active est efficace lorsqu'il s'agit de redémarrer un thread bloqué à une section critique. En effet, l'attente active est utile lorsque les threads travaillent sur une structure partagée ou lorsque la latence à la section critique est plus important que la durée d'exécution de celle-ci. La librairie *POSIX* quand à elle est plus efficace lorsqu'il faut mettre en oeuvre des synchronisations avec équité. On peut identifier les 3 problèmes de par leur structure aux deux méthodes disponibles, *POSIX* pour les philosophes et l'attente active pour les deux derniers. Il

convient donc de choisir la bonne méthode de "multi-threading" face à un problème sinon une dégradation notable de la performance possible surviendra.