UCLouvain

École polytechnique de Louvain

LINFO1252 - Système Informatiques

# Projet 1:
# Analyse de performance en Multithreading

Groupe 4


***Auteurs**:*
Delsart Mathis
Guerrero Gurriaran Anthony

***Enseignant**:*
Rivière Etienne

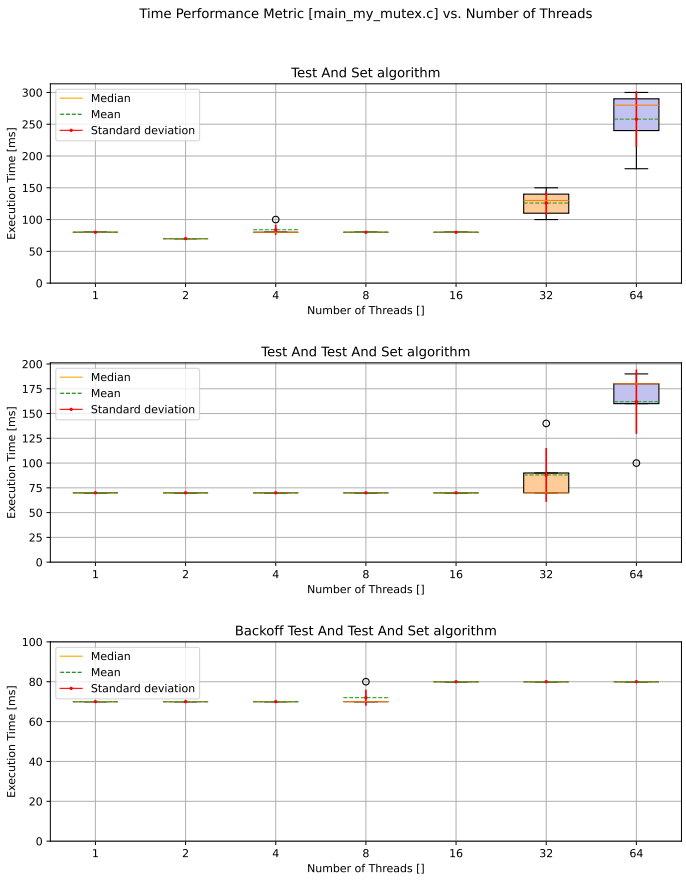
***Assistants**:*
Buchet Aurélien
Piraux Maxime
Rousseaux Tom
de Keersmacker François


2023-2024

# 1 Notations utilisées:

- TS = Test And Set
- TTS = Test And Test And Set
- BTTS = Backoff Test And Test And Set
- POSIX = Librairie C utilisée pour générer des threads POSIX, souvent appelés pthreads

# 2 Test de nos primitives d'attente active




Graphique illustrant le surcoût de synchronisation des threads en fonction du nombre de threads, en utilisant différents algorithmes d'attente active

***Fonctionnement du programme :***

Le programme executé est un test de performance visant à évaluer le surcoût de synchronisation associé à différentes implémentations de verrous (lock/unlock) d'attente active utilisées dans un environnement multithreadé. Chaque thread effectue 6400/N itérations d'une section critique.

***Type de courbe attendue :***

Bien que le nombre total d'itérations soit constant (6400), la présence de temps de synchronisation entre les threads rend improbable une courbe de temps d'exécution constante. On s'attend plutôt à observer une tendance croissante.

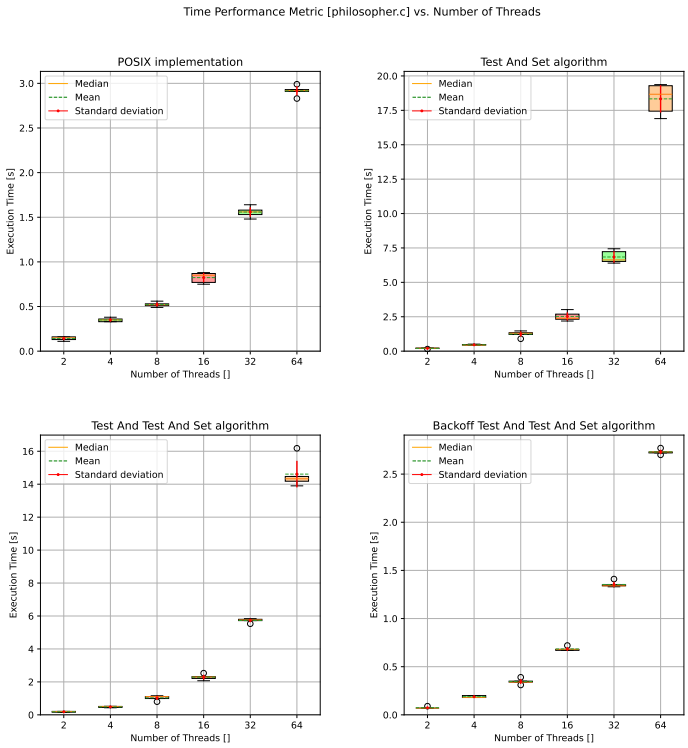
***Observations :***

L'analyse des résultats révèle des variations du temps d'exécution en fonction du nombre de threads et de l'algorithme de verrouillage utilisé. Pour un faible nombre de threads, les trois algorithmes semblent équivalent à quelques millisecondes près. Cependant, pour un nombre élevé de threads, les temps de synchronisation entre les threads pour TS et TTS deviennent significatifs, avec TTS montrant une amélioration d'environ 100 ms. BTTS maintient un surcoût de synchronisation constant, le positionnant comme le choix optimal.

***Interpretation des résultats :***

L'algorithme TS s'avère être le moins performant en raison de son recours coûteux à une opération atomique (xchg) à chaque itération, engendrant un surcoût significatif dû à l'arrêt du bus CPU lors de l'exécution de xchg. En revanche, l'algorithme TTS démontre une meilleure efficacité grâce à une approche intelligente réduisant la fréquence des opérations atomiques. Il commence par vérifier la disponibilité du verrou à l'aide d'une opération non atomique (movl), avant de tenter d'acquérir le verrou avec une opération atomique d'échange si celui-ci est libre. Quant à l'algorithme BTTS, il se distingue davantage avec une efficacité constante, illustrant l'impact positif de la stratégie de backoff. La stratégie backoff est implémenté en utilisant l'opération usleep(positif de la stratégie de backoff implémentée avec l'opération usleep(). Ces observations soulignent l'importance de choisir l'algorithme de verrouillage en fonction des exigences spécifiques de la charge de travail.

# 3 Problème des philosophes




Graphique illustrant le temps d'exécution du problème des philosophes en fonction du nombre de threads, en utilisant différents algorithmes d'attente active et l'implémentation de la librairie POSIX

***Fonctionnement du programme :***

Ce programme modélise la situation où plusieurs philosophes alternent entre la méditation et les repas autour d'une table, partageant des ressources communes pour manger. Chaque philosophe réalise un million de cycles penser/manger.

***Type de courbe attendue :***

Le nombre d'itération qu'effectue chaque thread étant constant (1 million), le temps d'exécution va augmenter avec le nombre de threads.
On peut anticiper deux phases :

- Une linéarité initiale avec un faible nombre de threads, où la performance augmente de manière linéaire. Chaque thread peut accéder aux ressources partagées sans trop de concurrence, ce qui permet une utilisation efficace des ressources.

- Une saturation à mesure que le nombre de threads augmente, entraînant potentiellement une croissance non linéaire (quadratique ou exponentielle) dû aux conflits et aux temps d'attente, car les threads se disputent les ressources nécessaires à l'exécution de leurs tâches.

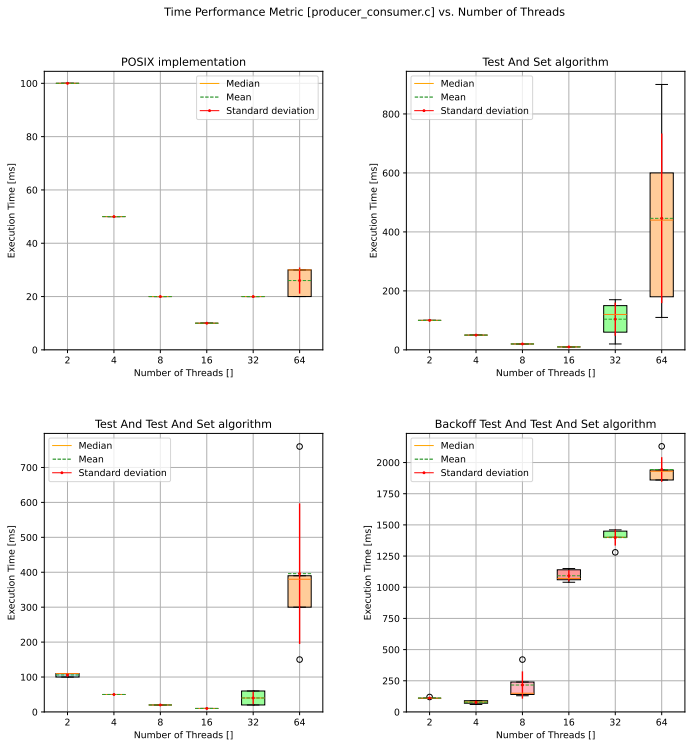
***Observations :***

Les courbes démontrent une tendance linéaire initiale pour un faible nombre de threads, conforme aux attentes. Cependant, une croissance non linéaire se manifeste avec une augmentation du nombre de threads, mettant en évidence les défis croissants de synchronisation. Pour un nombre de threads inférieur ou égal à 16, les algorithmes TS et TTS montrent des performances relativement équivalentes. Cependant, au-delà de cette limite, les avantages de TTS deviennent plus apparents, dépassant les performances de TS. Tandis que BTTS excelle, dépassant également POSIX. L'implémentation POSIX, bien que presque équivalente à BTTS, présente une légère baisse de performances. Cette observation suggère que bien que l'approche POSIX soit robuste, le mécanisme de backoff de BTTS apporte un avantage supplémentaire dans des conditions de concurrence élevée.

***Interpretation des résultats :***

Dans des scénarios de charge élevée comme le problème des philosophes, la stratégie de backoff confère un avantage significatif. Ces résultats soulignent l'importance de choisir soigneusement l'algorithme de verrouillage en fonction des exigences spécifiques de la charge de travail, avec BTTS émergeant comme une solution particulièrement efficace dans ce contexte du problème des philosophes.

# 4 Problème des producteurs/consommateurs




Graphique illustrant le temps d'exécution du problème des producteurs/consommateurs en fonction du nombre de threads, en utilisant différents algorithmes d'attente active et l'implémentation de la librairie POSIX

***Fonctionnement du programme :***

Ce programme modélise la dynamique du problème Producteur/Consommateur, où plusieurs threads interagissent pour produire et consommer des éléments partagés. Chaque thread participe à produire (resp. consommer) 8192 éléments.

***Type de courbe attendue :***

Lorsque le nombre de threads augmente, la charge de travail partagée entre eux peut conduire à une réduction du temps d'exécution, suggérant une tendance décroissante. Cette distribution équitable des tâches entre les threads est généralement bénéfique pour optimiser les performances. Cependant, à mesure que la synchronisation entre les threads devient plus prédominante avec une augmentation continue du nombre de threads, il peut y avoir un point critique. À ce point critique, le surcoût de synchronisation peut contrebalancer les avantages du travail partagé, entraînant un pic de temps d'exécution minimale suivi d'une remontée. Cela peut se produire lorsque la concurrence pour les ressources partagées devient intense, générant des conflits et des temps d'attente significatifs entre les threads.
On s'attend donc à une courbe adoptant une configuration en U (parabole positive non nécessairement symétrique).

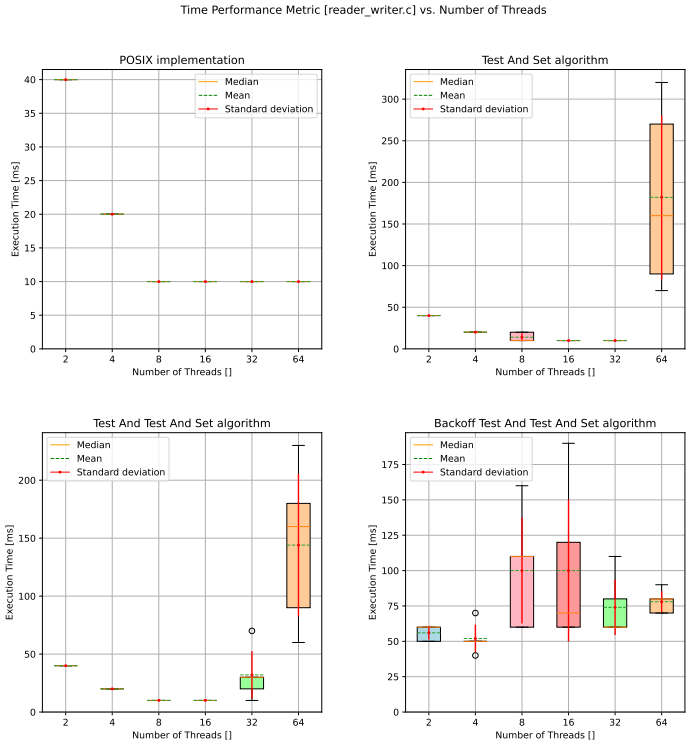
***Observations :***

Des observations révèlent une similitude de comportement entre TS, TTS et POSIX lorsque le nombre de threads est faible. Cependant, à partir de 32 threads, les temps d'exécution pour TS et TTS commencent à devenir significatifs. Un point critique émerge à 32 threads, avec un temps d'exécution minimal de 10 ms. Pour BTTS, le comportement est particulier, car son point critique apparaît à 4 threads, ce qui le rend peu performant pour ce type de problème.

***Interpretation des résultats :***

Pour un nombre faible de threads, le choix de l'algorithme n'est pas crucial car les comportements sont presque similaire. Cependant, l'utilisation de BTTS ici est à déconseiller au vu des mauvaises performances. Il est possible que les paramètres de BTTS n'aient pas été correctement adaptés, ce qui pourrait expliquer les résultats décevants par rapport aux attentes de meilleures performances.

# 5 Problème des lecteurs/écrivains




Graphique illustrant le temps d'exécution du problème des lecteurs/écrivains en fonction du nombre de threads, en utilisant différents algorithmes d'attente active et l'implémentation de la librairie POSIX

***Fonctionnement du programme :***

Ce programme simule la dynamique du problème Lecteur/Écrivain, où un écrivain (resp. lecteur) simule un accès en écriture (resp. lecture) à la base de données. Il n'y a pas de limite au nombre de lecteurs qui peuvent accéder simultanément à la base de données. Cependant, si un écrivain accède à la base de données, aucune autre opération, que ce soit en lecture ou en écriture, ne peut être effectuée en même temps que lui. Le programme effectue un total de 640 écritures et 2560 lectures.

***Type de courbe attendue :***

La forme de la courbe devrait adopter une configuration en U (parabole positive non nécessairement symétrique), similaire à celle observée pour le problème précédent des producteurs/consommateurs, en raison de similitudes dans les spécificités des deux problèmes. Cette anticipation est basée sur les caractéristiques partagées des deux problèmes, suggérant une tendance générale de diminution du temps d'exécution avec un nombre croissant de threads, suivie éventuellement d'une stabilisation ou d'une légère augmentation due aux contraintes de synchronisation à des niveaux élevés de concurrence.

***Observations :***

Des observations révèlent une similitude de comportement entre TS, TTS et POSIX lorsque le nombre de threads est faible. Cependant, à partir de 16 (resp. 32) threads, le temps d'exécution pour TTS (resp. TS) commence à devenir significatif. Un point critique émerge donc à 16 (resp. 32) threads, avec un temps d'exécution minimal de l'ordre de 10 ms. Pour BTTS, le comportement est très particulier, car le temps d'exécution augmente avec le nombre de threads et ce dès le début. On peut aussi voir qu'il y a enormément de variance pour TS, TTS et BTTS.

***Interpretation des résultats :***

Les performances des différents algorithmes dans le contexte du problème Lecteur/Écrivain suggèrent des nuances importantes. L'algorithme POSIX, avec sa rapidité conforme aux attentes, souligne la robustesse de son approche. D'autre part, TTS, bien qu'initialement performant jusqu'à 16 threads, révèle une vulnérabilité à des charges de travail plus lourdes, témoignant de la nécessité de considérer attentivement son utilisation en fonction des scénarios. TS affiche une constance relative, avec une augmentation significative seulement à 64 threads, mettant en lumière sa résilience à des charges modérées. Enfin, BTTS présente une variabilité dans ses résultats, indiquant une augmentation du temps d'exécution avec l'augmentation du nombre de threads, soulignant sa sensibilité aux conditions de charge. Comme indiqUé à la page précédente, il est possible que les param'etres de BTTS n'aient pas été correctement adaptés, ce qui pourrait expliquer les résultats décevants par rapport aux attentes de meilleures performances.
Ces observations soulignent l'importance cruciale de choisir l'algorithme de verrouillage en fonction des caractéristiques spécifiques de la charge de travail, avec POSIX émergeant comme une option efficace dans des contextes multithread complexes.

# 6 Conclusion

Ce projet avait pour objectif d'explorer et de comparer différents algorithmes d'attente active dans des contextes multithread complexes en mettant en œuvre des solutions pour des problèmes classiques tels que les philosophes, les producteurs/consommateurs et les lecteurs/écrivains. Notre approche a impliqué l'utilisation d'algorithmes tels que Test And Set (TS), Test And Test And Set (TTS), et Backoff Test And Test And Set (BTTS), ainsi que l'implémentation de la librairie POSIX pour évaluer les performances.

Les observations révèlent des comportements distincts en fonction du nombre de threads et de l'algorithme de verrouillage choisi. En général, pour des charges de travail légères à modérées, l'approche POSIX se distingue par sa rapidité et sa constance mais l'attente active est relativement similaire en temps d'exécution. En revanche, pour des charges plus lourdes, les algorithmes d'attente active, excepté BTTS (voir problème des philosophes), montrent des performances médiocres comparé à l'approche POSIX. BTTS se comporte de manière inhabituelle (sauf pour le problème des philosophes), soulignant l'importance cruciale du réglage précis des paramètres pour optimiser ses performances.

En termes de choix, l'attente active se révèle pertinente lorsque la concurrence est faible ou modérée, offrant une alternative efficace à POSIX. Cependant, pour des charges de travail plus élevées, POSIX émerge comme un choix plus stable et performant, soulignant la nécessité de sélectionner l'algorithme en fonction des exigences spécifiques de la charge de travail.

De plus, ce projet nous a permit d'acquérir des compétences pratiques, notamment la mise en œuvre de l'assembleur inline et la compréhension approfondie du coût de certaines opérations telles que l'instruction atomique d'échange (xchg). Ces connaissances ont enrichi notre compréhension des performances des algorithmes dans des environnements multithreadés.