groupe 8 (vendredi) - 2023-24

LINFO1252 – SYSTÈMES INFORMATIQUES

Systèmes Informatiques : Projet 1

Auteurs :
Arthur DE NEYER
Axel SNEESSENS

1 Le problème des philosophes

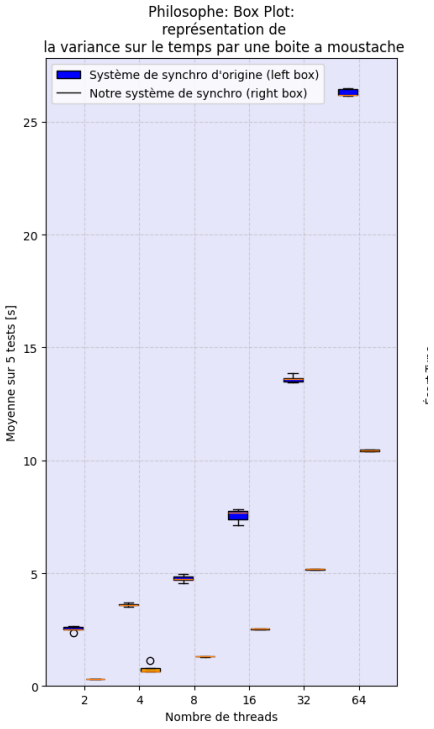
1.1 Explication

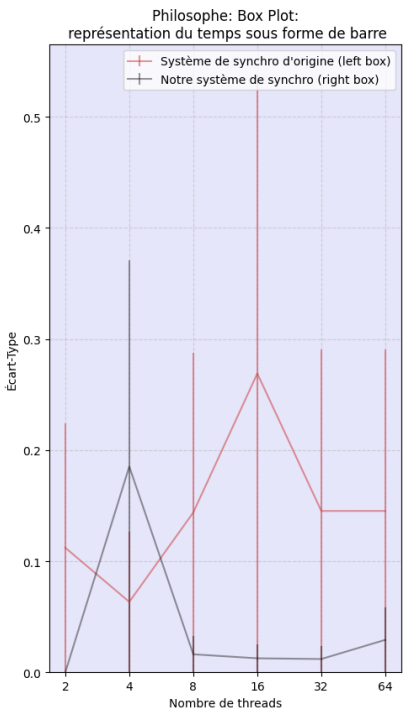
Le problème des philosophes, introduit par Dijkstra dans les années 1960, illustre les enjeux de l'exclusion mutuelle en programmation, particulièrement pour l'accès aux ressources partagées. Chaque philosophe possède deux baguettes (une à droite et une à gauche). Il se visualise par des philosophes alternant les actions "penser" et "manger", manger nécessitant les deux baguettes. Une approche simple mais défectueuse serait de saisir d'abord la baguette de gauche, créant un risque de blocage si tous attendent simultanément la baguette à leur droite.

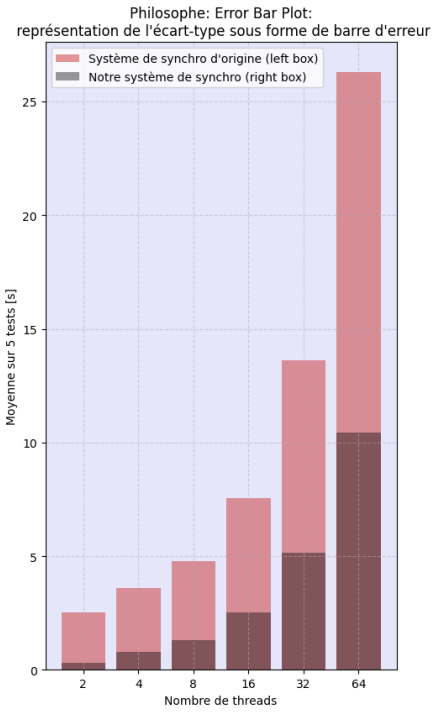
1.2 Implémentation

Nous avons décidé d'éviter les deadlocks avec une solution simple, les philosophes commencent toujours par la baguette gauche sauf le dernier qui commencera par la baguette droite, cela rend impossible toute situation de deadlock. Pour se faire, nous stockons les philosophes dans un tableau de thread `pthread_t* phil;` et les baguettes dans un tableau de `pthread_mutex_t* forks;`. Ils ont tous un ID qui correspond à leur position de leur tableau respectif. ainsi chaque philosophe tentera de prendre la baguette avec le plus petit ID en premier. Le philosophe à la position `X` prendre la fourchette `X` puis la `X+1`, sauf pour le dernier philosophe à la place `Y` qui prendra la fourchette `0` puis `Y`.

1.3 Résultat







Warning : Le premier graphe représente des box plots. Pour des raisons de rendus (pour ne pas qu'elle soit l'une sur l'autre), nous n'avons pas aligné verticalement les boxs. Celles-ci sont cependant bien relatives au même nombre de threads !

D'après les graphiques, on peut observer que, pour les deux systèmes, le temps d'exécution moyen augmente avec le nombre de threads. Cela est dû à la concurrence accrue pour les ressources, ainsi que l'augmentation du nombre de philosophe et donc de cycle (manger, penser). De plus le code semble rester stable indépendamment du nombre de threads.

2 Producteur et consommateur

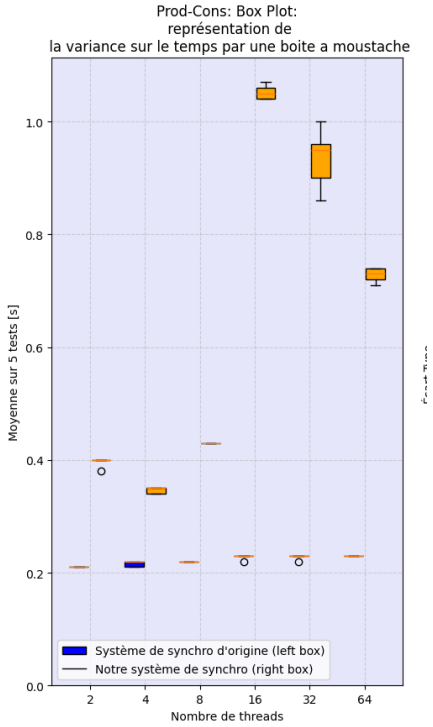
2.1 Explication

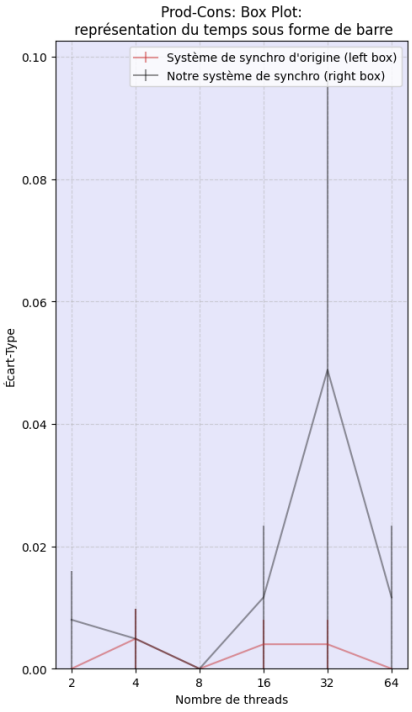
C'est un problème classique qui requiert un producteurs produisant des données afin de les placer dans une zone mémoire accessible aux consommateurs. Les consommateurs lisent les données produites, chaque donnée doit être consommée exactement une seule fois, les threads producteurs et consommateurs progressent à un rythme quelconque, et leur nombre n'est pas forcément connu à l'avance.

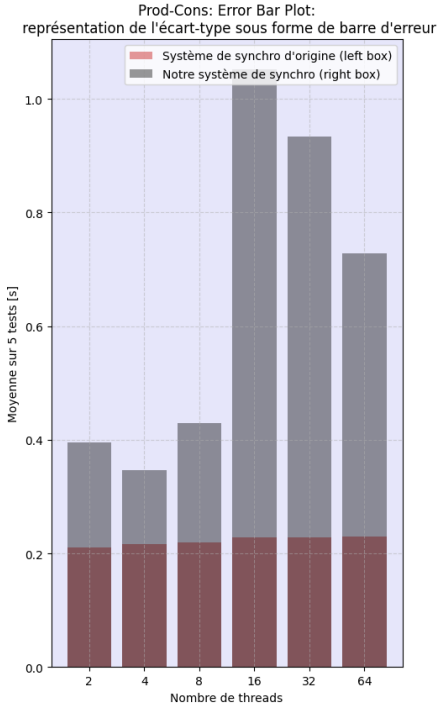
2.2 Implémentation

Pour rappel, le nombre de threads consommateurs et de threads producteurs sont égaux, et la taille du buffer `N` est égal à `8`. Nous avons donc profiter de ces caractéristique pour lancer les 2 types de thread en même temps à travers une boucle for. Pour respecter le nombre imposé d'éléments produit et consomé `(8192)` nous avons utilisé deux variables qui valent `8192` et qui sont décrémentées une fois qu'un thread est en zone critique. Il reste à gérer l'attente des producteurs si le buffer est plein et l'attente des consommateurs si le buffer est vide, pour se faire nous utilisons les deux sémaphores (empty et full), `sem_init(&empty, 0, N);` et `sem_init(&full, 0 , 0);` afin d'attendre si le buffer est vide ou plein.

2.3 Résultat







Il y a une tendance claire que le temps moyen augmente avec le nombre de threads pour notre implémentation. Ce n'est pas le cas pour l'implémentation d'origine, où le temps semble rester relativement constant, indépendamment du nombre de threads. Notre système de sémaphore semble moins efficace lorsque le nombre de threads augmente, car le temps moyen augmente considérablement. De plus les box-plots nous montre que le code semble moins stable avec notre implémentation de sémaphore.

3 Lecteur et écrivain

3.1 Explication

Le problème des lecteurs / écrivains met en avant la complexité d'interactions avec une ressource partagés par différents threads. En particulier, nous avons ici `nb_writers` et `nb_readers` threads souhaitant intéragir avec la ressource partagée. Les règles sont simples :

— Plusieurs readers peuvent accéder simultanément à la ressource partagée ;
— Les écrivains effectuent des modifications sur la ressource partagée, en exclusivité.

3.2 Implémentation

Notre implémentation se base sur l'utilisation de mutex et de sémaphores afin d'assurer le respect aux règles et assurer une bonne synchronisation. Les sémaphores sont utilisés pour contrôler l'accès à la ressource partagée, tandis que les mutex aident à incrémenter et décrémenter les compteurs de lectures et écritures de manière correcte.

Dans `reader`, nous utilisons des mutex et des sémaphores pour assurer que plusieurs lecteurs puissent accéder à la ressource simultanément, mais toujours en exclusion avec les écrivains. Pour cela nous utilisons un système de compteur (`read_count`), qui nous permettra, en fonction de sa valeur, de donner/refuser l'accès à la ressource partagée aux autres threads par des sémaphores. Nous-mêmes sommes aussi contraints par ces sémaphores.

Le schéma est fort similaire dans `writer`, où nos sémaphores s'assurent qu'un seul écrivain, en exclusivité sur la ressource, puisse travailler.

3.3 Résultat

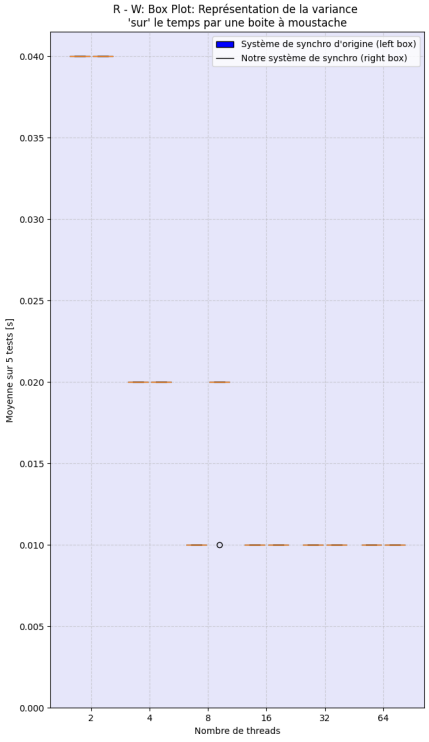


FIGURE 1 – First Image

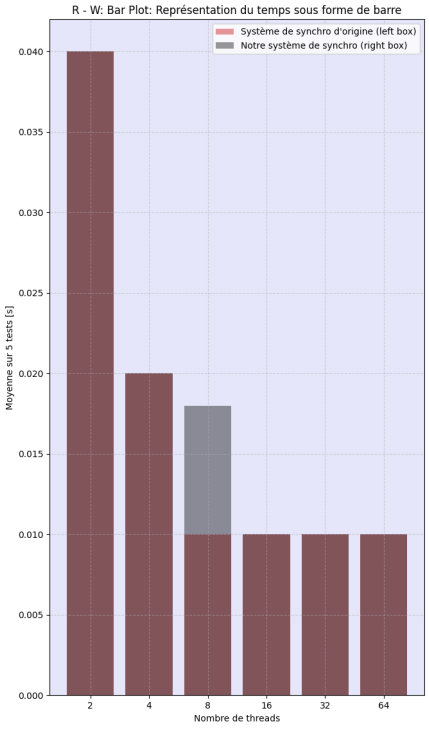


FIGURE 2 – Second Image

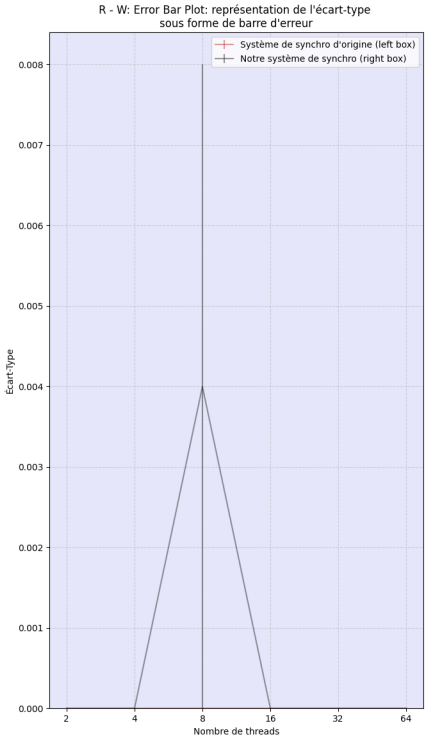


FIGURE 3 – Third Image

Nous pouvons voir que de manière générale, notre variance est très faible entre les tests. Cela est expliqué par la stabilité de la machine INGInious. En terme de comparaison de temps entre les deux implémentations du système de synchronisation, nous pouvons voir que dans ce cas, notre implémentation est aussi rapide que celle de base. On voit aussi que plus le nombre de threads augmente, plus le programme s'exécute rapidement. Cela est simplement dû au fait que chaque thread aura une charge de travail moins grande. On voit cependant qu'à partir de 16 threads, le rapport n'évolue plus. Cela est dû au fait que

la création d'autant de threads prend du temps, et n'est pas compensée par la diminution de la charge de travail par chaque thread.

4 Test-And-Set et Test-And-Test-And-Set

4.0.1 Test-And-Set

L'algorithme "test-and-set" est une technique de synchronisation utilisée pour garantir que deux ou plusieurs threads ne puissent accéder simultanément à une ressource partagée, telle qu'une variable ou une section critique. L'idée de base est de créer une opération atomique, c'est-à-dire une opération qui s'exécute en une seule étape indivisible . cette opération s'effectuera seulement une fois qu'elle aura passer le test garantissant la liberté de la ressource utilisé.

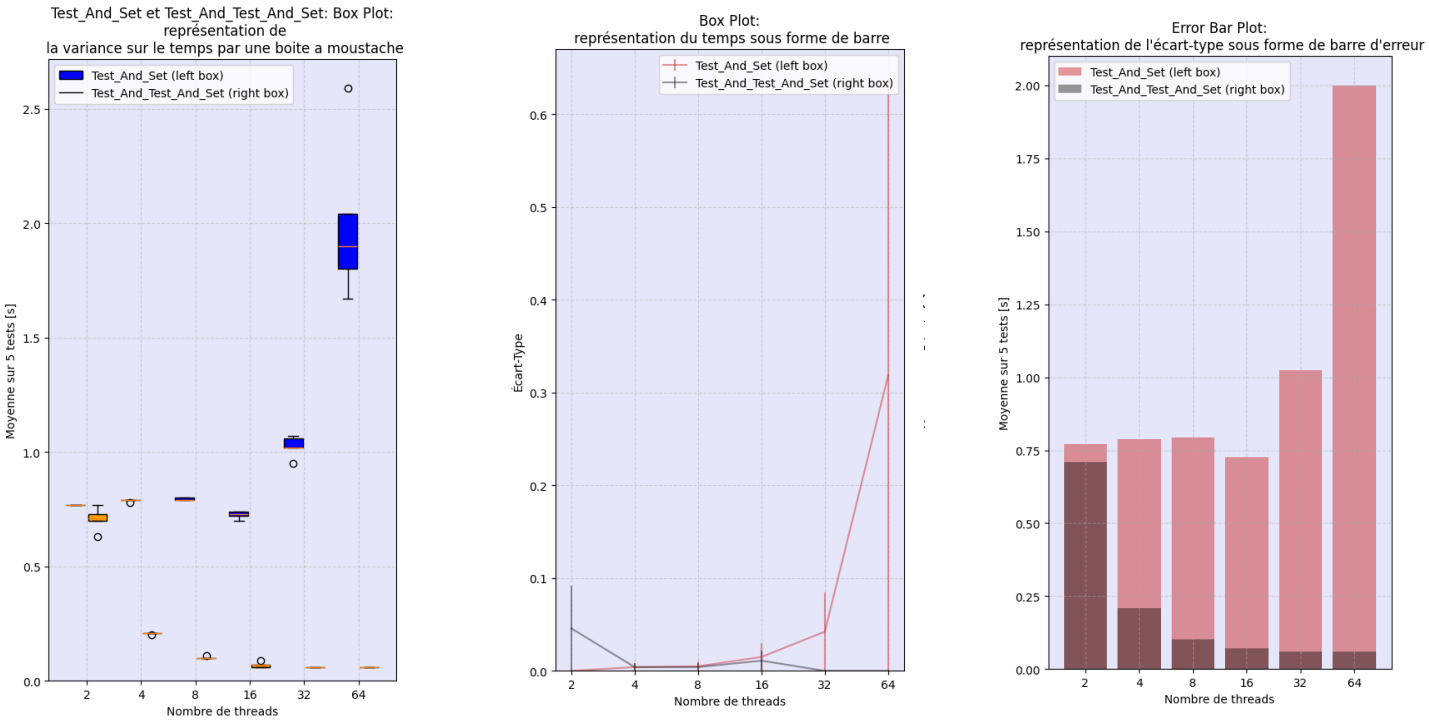
Nous utilisons une variable temporaire initialisé à 1 qui échangera sa valeur avec celle du verrou `int verrou = 0;` qui est initialisé à 0 pour signifier qu'il est ouvert. Si le verrou == 0, alors le thread échange la valeur de la variable temporaire avec celle du verrou, donc le verrou = 1. Tant que le verrou est à 1, le thread qui exécute sa section critique n'a pas terminé. Si un thread lit `verrou == 0` alors il tente d'appeler l'opération atomique xchg avec `"xchgl %%eax, %0;"`, cela échangera les valeurs de la variable temporaire et du verrou.

4.0.2 Test-And-Test-And-Set

L'algorithme "test-and-test-and-set" est une variante de l'algorithme "test-and-set" qui vise à améliorer l'efficacité en réduisant l'immobilisation du bus quand la contention est élevée, mais l'impact sur la performance reste non négligeable.

Comparer à l'algorithme Test-And-Set si le premier test de la partie `Test-And-Set` échoue, on va lancer un deuxième test `while (verrou == 1) {}` afin d'attendre que la condition soit vrai avant de tester à nouveau cette même partie.

4.1 Résultat



Les résultats montrent que, pour le `Test-And-Set` la moyenne de temps augmente à mesure que le nombre de threads augmente, ce qui pourrait indiquer que l'opération devient plus coûteuse en temps lorsqu'il y a beaucoup de thread. Pour le

`Test-And-Test-And-Set` , il semble etre plus efficace lorsqu'il y a de plus en plus de thread ce qui pourrait signifier que cette opération est moins sensible à la concurrence ou subit moins de contention entre les threads.

5 Conclusion

En conclusion, ce projet nous a permis de mieux comprendre les nuances de la synchronisation dans les systèmes informatiques. À travers les diverses tâches, nous avons pu acquérir une meilleure compréhension des primitives de synchronisation POSIX ainsi que de la mise en œuvre des verrous par attente active. De plus nous avons amélioré notre compréhension de l'impact de la concurrence sur les performances. En codant les problèmes classiques des `philosophes`, `producteurs/consommateurs`, et `lecteurs/écrivains`, et en évaluant leurs performances sur une machine multi-coeurs, nous avons appris à analyser le comportement d'applications en multithreading.

L'utilisation de mutex et de sémaphores, suivie de l'implémentions de verrous avec attente active, a révélé des différences subtiles dans la manière dont les threads interagissent et les effets de ces interactions sur le temps d'exécution total. La nécessité de mesures répétées pour obtenir des données fiables et la représentation graphique des résultats ont renforcé notre compréhension sur l'analyse des données et de leurs performances.