Projet 1

Travail réalisé dans le cadre du cours LINFO1252

Systèmes informatiques

Le 20 Novembre 2020

Année académique 2020 - 2021

Groupe I :

Nicolas jadoul **(21641800)**

Saskia Juffern **(24411800)**

Professeur :

Etienne Riviere

Assistant :

Tom Rousseaux

Table des matières

Introduction

Dans le cadre du cours LINFO1252 - Sytstèmes informatiques, on devait réaliser ce Projet 1 dont le but est d'apprendre à évaluer et expliquer la performance d'applications utilisant plusieurs threads et des primitives de synchronisation : mutex, sémaphores (dans la première partie) ainsi que les verrous avec attente active, que nous devons implémenter nous même (dans la deuxième partie). Dans un premier temps, nous avons réalisé trois applications : Le problème des philosophes, les producteurs/consommateurs avec un buffer de taille fixe et lecteurs/écrivains avec la priorité aux écrivains. Puis nous avons écrit des scripts pour réaliser des performances. On prends au moins 5 mesures pour chaque configuration de 1 à 8 threads (notre machine ayant 4 threads) pour chacun des trois problèmes.

Dans un deuxième temps, nous devons mettre en oeuvre un verrou par attente active utilisant une opération atomique sur le modèle de test-and-set vu en cours, puis implémenter l'algorithme test-and-test-and-set et comparer les résultats de performance. Finalement nous devons implémenter une interface sémaphore sur base de notre primitives d'attente active et adapter nos 3 algorithmes pour utiliser ces primitives et comparer les performances.

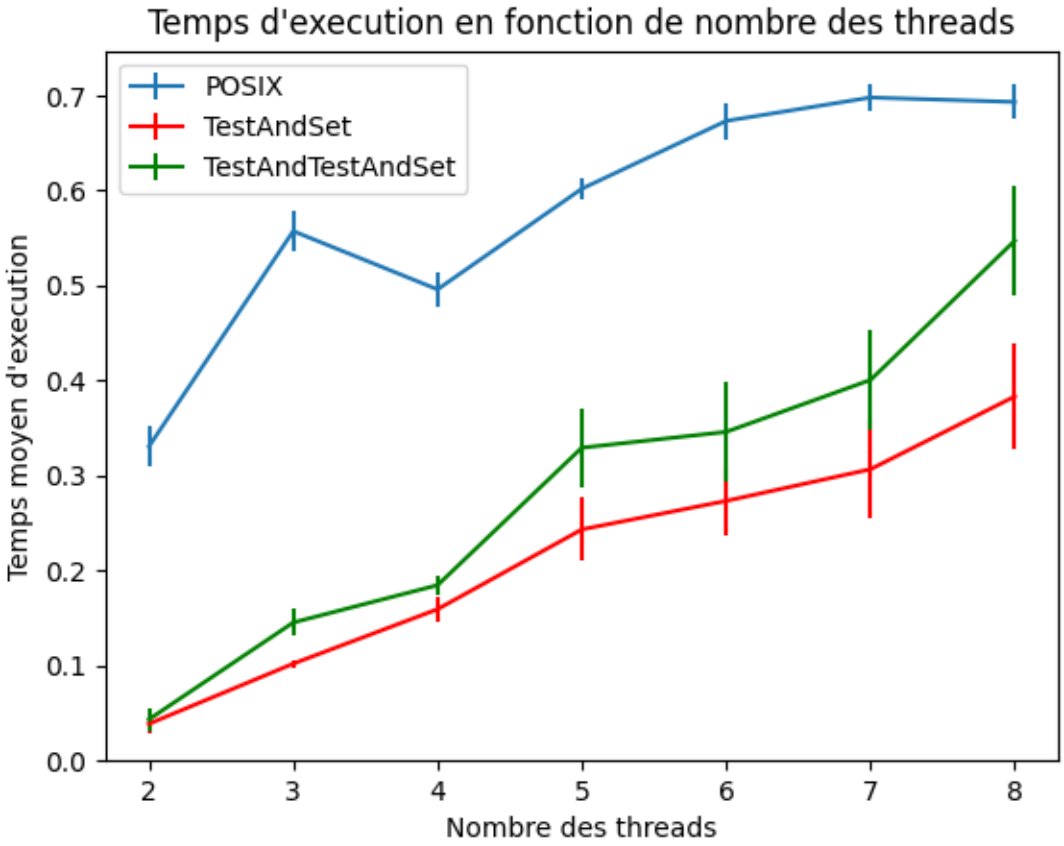
1 Le problème des philosophes

Le problème des philosophes est que N philosophes sont assis autour d'une table circulaire avec une fourchette entre chaque philosophe. Un philosophe peut manger s'il peut ramasser les deux fourchettes qui lui sont adjacentes. Une fourchette peut être ramassée par n'importe lequel de ses adeptes adjacents, mais pas les deux. Ce problème permet d'illustrer le principe du deadlock.
La solution que nous avons choisie pour ce problème est avec des moniteurs pour chaque philosophe. Ces moniteurs seront implémentés par des sémaphores qui se débloque en fonction de l'état du philosophe lié.

1.1 Remarque implémentation

A cause d'un soucis de performance nous avons 2 implémentations différentes du problème. Lors de notre implémentation avec les mutex POSIX, nous avons remarqué une mauvaise répartition des opérations. Nous avons donc créé deux variables pour chaque philosophes qui surveillent si un de ses 2 voisins a faim. Ceci permet de mieux répartir les opérations des philosophes. Ce problème n'apparaissait pas avec l'implémentation utilisant nos mutex. Mais si nous essayions d'ajouter ces 2 nouvelles variables à cette dernière, nous observions une augmentation drastique de performance (de 400ms à 100s). Nous avons donc opté pour 2 implémentations un peu différentes.

1.2 Analyse de performance



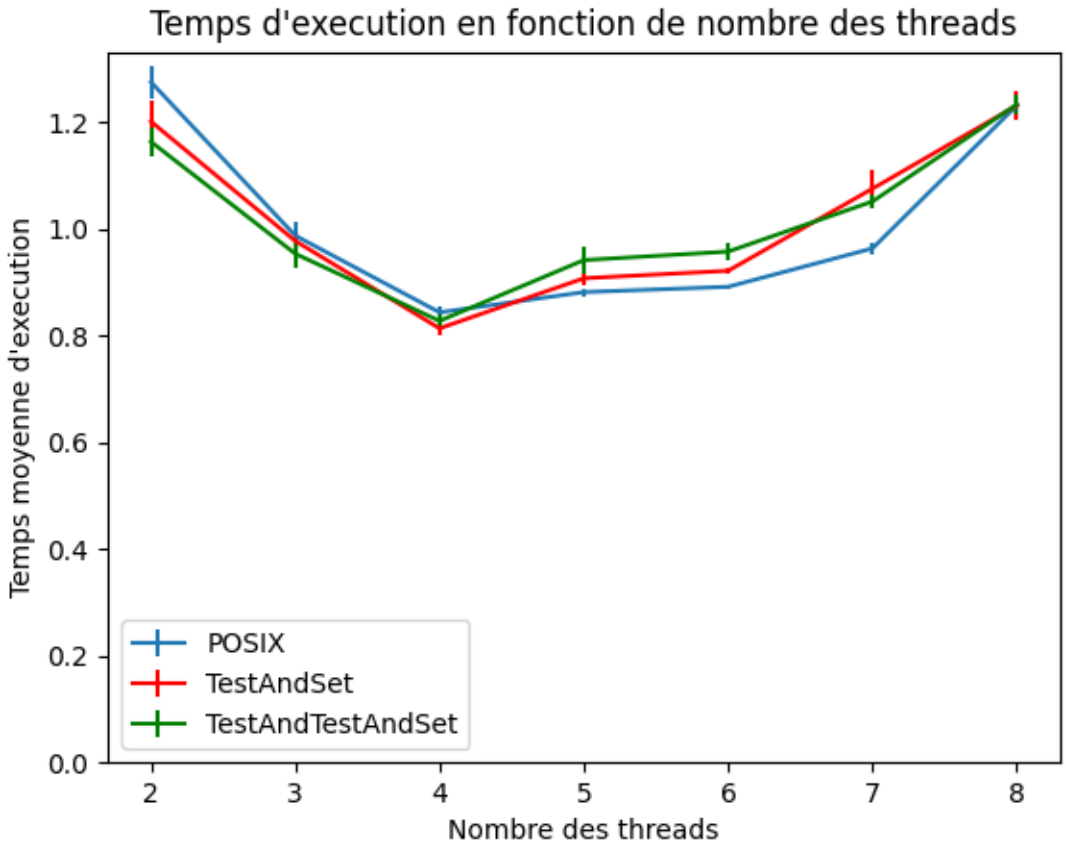
De manière générale le temps d'exécution augmente avec le nombre de threads. Ce qui est tout à fait normal car il y a plus d'opérations à faire en augmentant le nombre de threads et plus de partage de fourchettes. Idéalement l'augmentation devrait être linéaire, si l'implémentation est bonne. En effet, dans ce problème, il n'y a qu'un thread qui travaille à la foi en bloquant les autres peu importe le nombre de thread donné.
Malheureusement notre implémentation avec les mutex POSIX n'est pas idéale et ne représente pas bien la performance attendue. Mais celle avec nos mutex le montre parfaitement.
Nous remarquons aussi que le "test and test and set" est plus lent que le "test and set". Cela est dû au fait que, lors des collisions dans le "test and test and set", les threads doivent à chaque fois vérifier la condition de la boucle intérieur lié au cache et puis seulement déterminer atomiquement le blocage. Contrairement au "test and set" qui ne fait que la partie atomique.

2 Le problème des producteur-consommateur

Le problème des producteurs-consommateurs est un programme où il y a N producteurs et M consommateurs qui communiquent à l'aide d'un buffer. Les producteurs produisent des éléments et déposent les dans le buffer et les consommateurs prennent ces éléments du buffer et utilisent les. Le buffer, qui dans notre cas a une taille de 8 éléments, doit être protégé par un mutex. Il faut au moins une place libre pour que les producteurs peuvent y ajouter des éléments (si tout est rempli, il est bloqué jusqu'à ce qu'une place se libère) et pour que le consommateur puisse prendre des

éléments du buffer, il faut qu'il y a au moins un élément dedans (s'il y a aucun élément dans le buffer, il attend jusqu'à ce qu'un élément soit inséré). Pour satisfaire ces conditions, nous utilisons 2 sémaphores : Le premier, nommé "empty" sert à compter les places libre dans le buffer et le deuxième, "full", compte les places occupées.

2.1 Analyse de performance

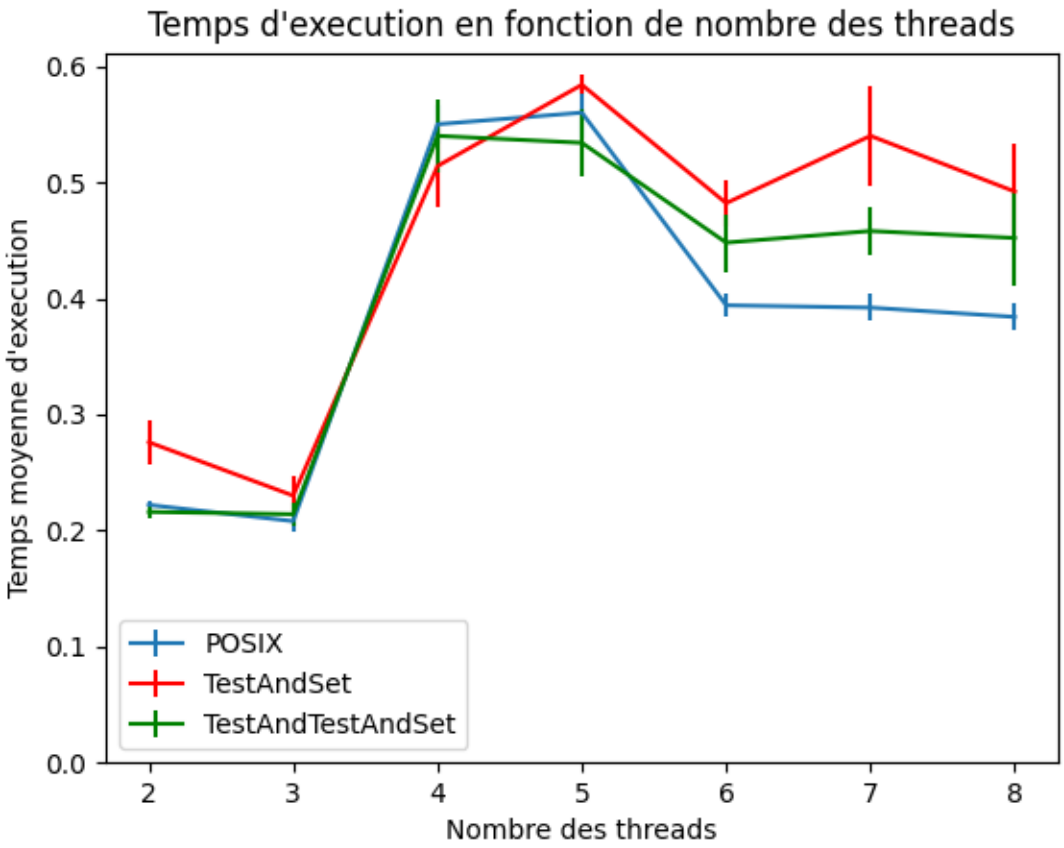


En général, le programme est le plus rapide si on utiliser 4 threads. Pour 2 et 3 threads, la variante avec test-and-test-and-set est un peu plus rapide, suivi près par la variante test-and-set et la variante POSIX est la plus lente. Après 4 threads, le variante POSIX est la plus rapide et les deux autres sont un peu plus lentes.

3 Le problème des écrivains et lecteurs

Dans ce problème,il y a des écrivains et des lecteurs, qui, toutes les deux accèdent à une structure de données. Plusieurs lecteurs peuvent accéder en même temps, mais qu'un seule écrivain peut accéder au structure, quand il accède aucun lecteur ni d'autres écrivains peuvent y accéder. Au total, il y a 5 mutex et 2 sémaphores et dans cette implémentation et les écrivains sont prioritaire.

3.1 Analyse de performance

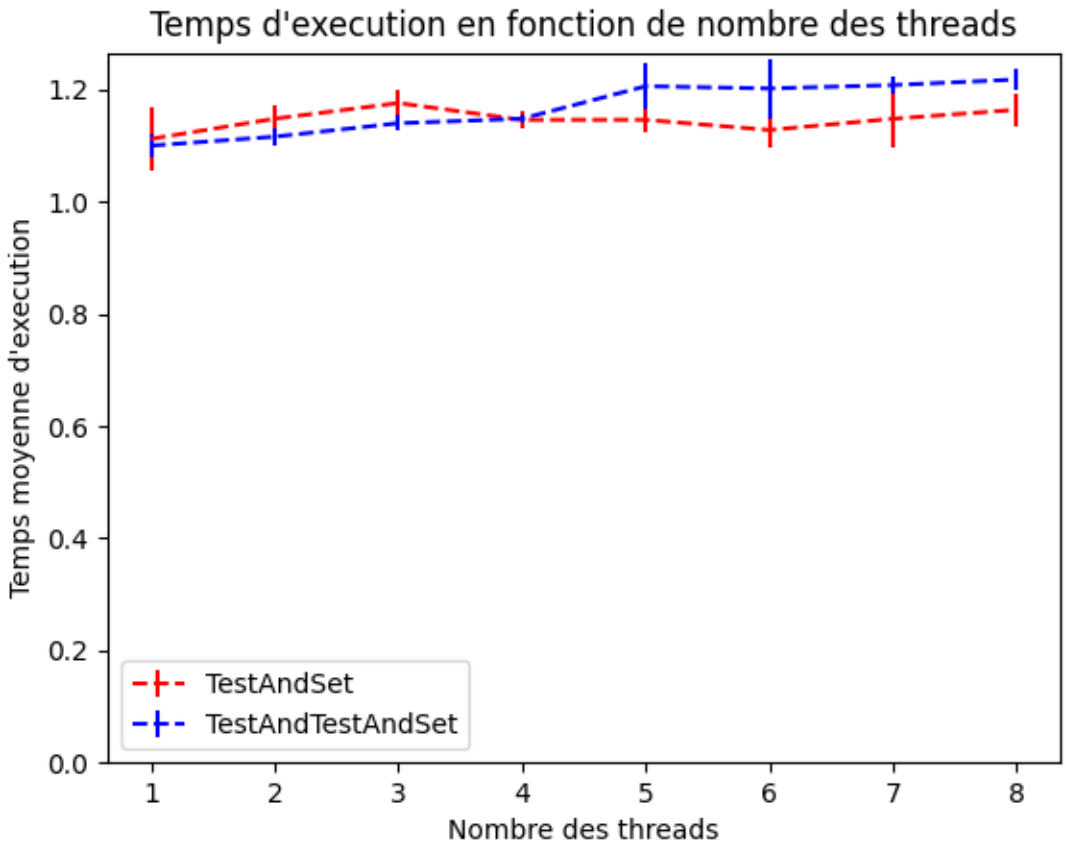


La performance est la plus bonne si on a trois threads, si on a plus que trois, le temps augmente et reste dans cette hauteur. Avec 2 ou trois threads, POSIX et test-and-test-and-set on presque la même performance et test-and-set est un peu plus lente. Avec 3 ou 4 threads, la performance est presque identique est si on surmonte 4 threads, POSIX est plus rapide, suivi par test-and-test-and-set.

4 Test-and-Set et Test-and-Test-and-Set

Le "test and set" une instruction atomique pour écrire une valeur prédéterminée dans un emplacement mémoire et retourner la valeur d'origine de cet emplacement. Elle permet l'implémentation d'un mutex qui permet de bloquer des threads. La base de la concurrence. Le "test and test and set" est identique sauf qu'il rajoute une condition qui dépend de la valeur en cache du mutex. Cette condition limite les appels à l'échange atomique coûteuse.

4.1 Analyse de performance



Les deux implémentations de mutex prennent globalement le même temps d'éxecution. Le temps ne provenant que de l'attente random. Nous remarquons que le "test and test and set" est plus lent à partir de 4 threads et cela pour la même raison que dans le problème des philosophes. En dessous de 4 threads c'est l'inverse car "test and set" fait plus d'échange atomique coûteux.

Conclusion

Mis-à-part l'implémentation des philosophes avec les mutex POSIX, nous pensons que ce projet a été réalisé avec succès. Il manque aussi une analyse plus poussée pour les performances des problèmes des écrivains-lecteurs et producteurs-consommateurs. Nous avons cependant beaucoup appris, que ce soit sur la manière de gérer la concurrence en informatique, comment les threads s'enchaînent et opèrent en simultané dans un ordinateur, comment les mutex et les sémaphores fonctionnent concrètement et nous avons pu voir une application du langage assembleur qui fait maintenant moins peur à première vue.