LINFO1252 – Systèmes informatiques

Volders Maxime (26521700) - Gauthier Arnold (56571700)
Date de soumission : 25 Novembre 2020

Table des matières

1 Introduction

Dans le cadre du cours LINFO1252, il nous a été demandé d'implémenter différents problèmes et algorithmes et de faire une analyse de leur performance. Les 3 différents problèmes sont : le problème des philosophes, le problème du producteur-consommateur, et le problème du lecteur-écrivain. Ceux-ci ont été exécutés 2 fois : une fois avec les verrous et sémaphores présents sur C dans la librairie POSIX, et une autre fois avec les verrous et sémaphores par attente active que nous avons implémentés.

Le rapport est séparé en 2 parties : d'une part nous regarderons aux 3 problèmes, et d'autre part, nous parlerons des verrous que nous avons implémentés et nous analyserons les performances des verrous test and set, test and test and set, et backoff test and test and set.

Dans les différents graphiques qui suivent, les barres verticales représentent l'écart type, quant aux points, ils représentent la moyenne de temps d'exécution. Les couleurs des graphiques sont également cohérentes entre elles dans les 3 problèmes ; le bleu représente toujours l'implémentation en utilisant les mutex et sémaphores fournies par la librairie POSIX, et le rouge représente notre propre implémentation par attente active.

Pour faire nos tests nous avons utilisé les ordinateurs de la salle Intel afin d'avoir des machines avec 4 coeurs.

2 Performance des problèmes

2.1 Problème des philosophes

Le premier problème que nous avons analysé est le problème du philosophe. Après avoir exécuté notre code, nous nous sommes aperçu que les temps obtenus avec 10.000 cycles penser/manger n'étaient pas suffisants, nous avons donc décidé suite au conseil d'un tuteur de faire 1.000.000 de cycles penser/manger. Nous nous sommes aperçu que lorsqu'un philosophe était seul, il n'avait qu'une baguette. Nous avons décidé de mettre une deuxième baguette lorsqu'il n'y a qu'un seul thread pour que les cycles penser/manger puissent être réalisés.
Les graphes ci-dessous représentent les résultats obtenus avec le verrou POSIX et avec notre verrou en attente active.

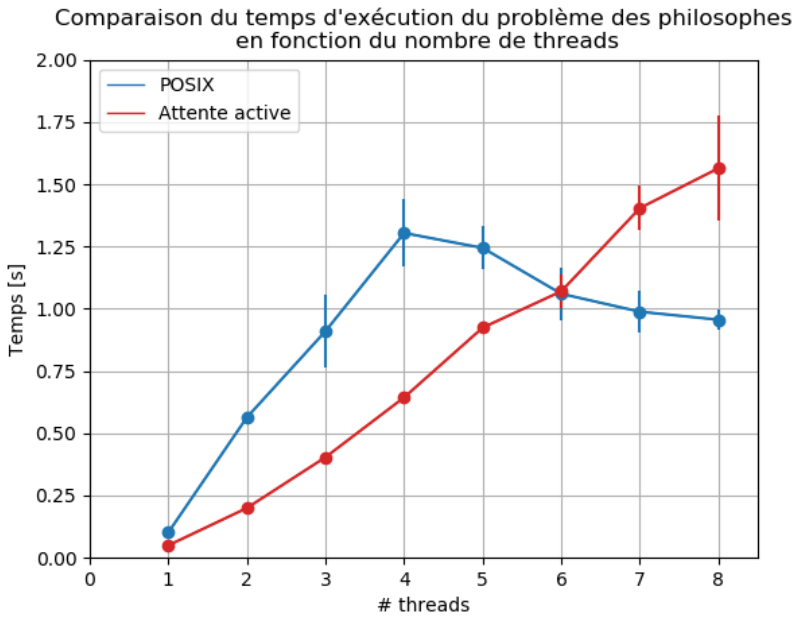

FIGURE 1 – Performance problème des philosophes

La figure 1 représente le temps d'exécution en seconde de notre implémentation du problème des philosophes en fonction du nombre de thread.
Nous nous attendions à avoir des graphes plus ou moins tels que ceux que nous avons obtenus. Il est normal que avec un seul thread le temps d'exécution soit si faible. En effet, lorsqu'un philosophe est seul, il n'a pas à attendre que les baguettes se libèrent pour pouvoir les saisir et manger, donc tout s'effectue rapidement. De plus, les 1.000.000 de cycles ne sont faits qu'une seule fois, ce qui permet aussi d'aller plus vite.
Ensuite, comme le nombre de thread (et donc le nombre de philosophes) augmentait, il est normal de voir une augmentation du temps moyen nécessaire à l'exécution du programme entre 1 et 4 threads. Nous avons trouvé 2 raisons à cela : d'une part, il y a de plus en plus de cycles penser/manger à faire, et d'autre part car à partir de 2 threads, on commence à avoir des temps d'attente.
Dans le cas des threads POSIX, nous pensions que le temps serait constant ou augmenterait à partir de 4 threads, comme les ordinateurs utilisés disposaient de 4 coeurs. Ici, nous avons une diminution du temps d'exécution. Comme on a plus de philosophes, il y a moins de chance que 2 philosophes voisins veuillent manger exactement au même moment, ce qui peut expliquer cette légère diminution
Dans le cas de l'attente active, nous nous attendions à toujours avoir une augmentation. En effet, de plus en plus de threads peuvent être mis en attente active, ce qui va causer une perte de temps considérable.

Abordons les différences entre l'exécution avec POSIX et l'exécution avec attente active.
Entre 1 et 5 threads, nous voyons que l'exécution avec attente active est plus rapide que l'exécution avec POSIX. Ceci peut venir du fait que en POSIX, lorsque les threads sont en attente, ils libèrent leur coeur, qu'ils doivent récupérer plus tard. Le POSIX passe par l'état "ready" avant l'état "running", alors qu'en attente active, les threads passent de l'état "blocked" (quand ils sont dans la boucle) à l'état "running"

directement. Ils ne libèrent jamais leur coeur. Cette non-libération de coeur commence à avoir un plus gros impact que la demande de coeur à partir du sixième thread. A partir de 6 threads, le temps d'exécution est plus rapide en POSIX qu'en attente active. Comme en attente active, les threads ne sont pas à l'arrêt total, ils continuent d'occuper les coeurs. Et comme leur nombre augmente, les coeurs sont de plus en plus souvent utilisés inutilement. Ce qui fait que le POSIX finit par être plus rapide.

2.2 Problème des producteurs et consommateurs

Nous avons ensuite analysé le problème des producteurs et des consommateurs. Le nombre de threads représenté est la somme des threads producteurs et consommateurs. Lorsque ce nombre est impair, il y a un consommateur en plus qu'il n'y a de producteurs, et lorsqu'on a un seul thread, donc un seul consommateur, nous ajoutons un producteur.

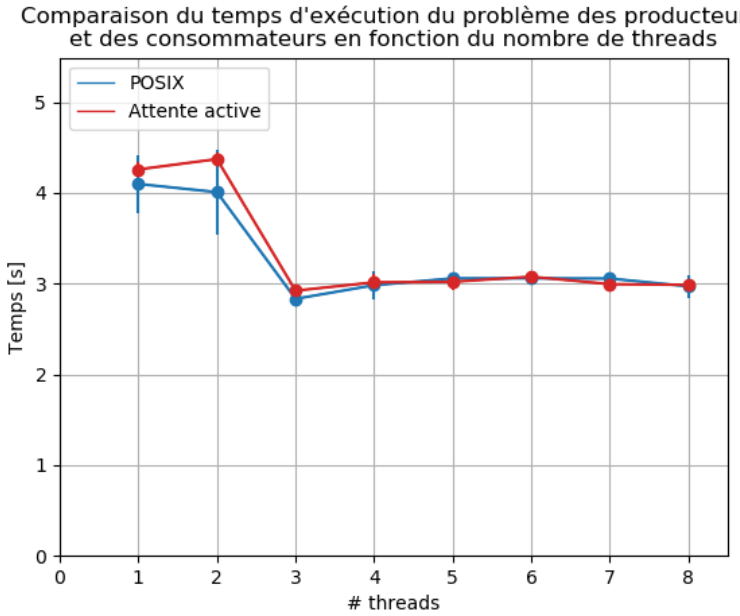

FIGURE 2 – Performance problème des producteurs et consommateurs

La figure 2 représente les différents temps d'exécution en secondes de notre implémentation du problème des producteurs et des consommateurs en fonction du nombre total de thread.
On constate d'abord qu'entre un et deux threads, nos performances sont constantes. Ceci s'explique simplement par le fait que lorsqu'on a un seul thread, nous sommes forcés de rajouter un producteur en plus sinon le problème ne se termine jamais, car il ne peut même pas commencer. Dans le cas d'un thread, nous sommes donc dans exactement la même situation que dans le cas de deux threads.
Ensuite, lorsqu'on passe à trois threads (deux consommateurs et un producteur), on a une meilleure performance. On peut expliquer cela par le fait qu'à partir du moment où on a plus d'un consommateur qui traite les données, il est logique que ça se fasse plus rapidement. C'est pourquoi, nous nous attendions à obtenir un graphique décroissant : plus le nombre de thread augmente, plus la performance est bonne, et donc le temps d'exécution faible. Or, nous constatons qu'à partir de 3 threads, peu importe l'augmentation du nombre de producteurs et de consommateurs, le temps d'exécution ne varie pas et la performance reste donc la même. Cela peut s'expliquer par le fait que malgré que les producteurs et consommateurs sont plus nombreux pour traiter les données, le temps nécessaire pour insérer/extraire une donnée dans le buffer reste inchangée, et donc plus il y a de threads, plus il y a de concurrence afin d'accéder à la section critique, et donc plus il y a de l'attente. Cette augmentation de l'attente, compensée par l'augmentation du nombre de producteurs et consommateurs, résulte donc bien en une courbe constante.
Finalement, on peut constater qu'en général sur tout le graphique, le temps d'exécution du problème avec les locks POSIX est légèrement plus performant que le temps d'éxécution avec l'attente active. C'est tout à fait logique, étant donné que les locks fournis par la librairie POSIX ont une attente passive et ne diminuent donc pas les performances.

2.3 Problème des lecteurs et écrivains

Le dernier problème à analyser était le problème des lecteurs et écrivains. Le nombre de thread représente la somme des thread lecteurs et consommateurs. Lorsque ce nombre est impair, il y un lecteur en plus qu'il n'y a d'écrivains.

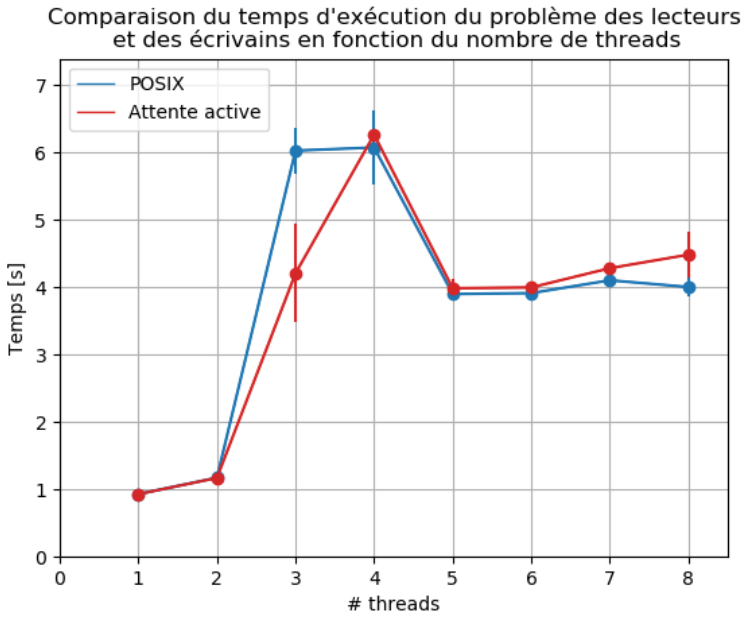

FIGURE 3 – Performance problème des lecteurs et écrivains

La figure 3 représente les différents temps d'exécution en seconde de notre implémentation du problème des lecteurs et écrivains en fonction du nombre de thread.

On peut d'abord voir que les temps avec 1 et 2 threads sont très semblables. Lorsqu'on a 1 thread, on va juste avoir un lecteur qui va effectuer ses lectures. Il n'y a donc aucun temps d'attente. Lorsqu'on a 2 threads, la seule différence est qu'on va avoir 1 écrivains en plus, qui va faire ses écritures. Le temps nécessaire pour faire les écriture est minimes, donc il y a une toute petite augmentation.
Ensuite, dans le cas du programme utilisant les threads et sémaphores POSIX, nous avons une multiplication par 6 du temps moyen pour exécuter le programme entre 2 et 4 threads. Dans ce cas-ci, contrairement au problème des producteurs et des consommateurs, le temps d'exécution de la zone critique est beaucoup plus élevé. Dans le cas du programme utilisant des attentes actives, cette multiplication par 6 s'effectue entre 2 et 3 threads.
Lors de nos tests, nous avons remarqué que c'est notre fonction `read_database` qui prend beaucoup plus de temps. Lorsqu'on passe de 1 à 2 lecteurs, le temps d'exécution moyen de cette fonction est multiplié par 10. Mais l'ajout d'un lecteur diminue ce facteur pour le temps d'exécution global du programme, ce qui fait que nous avons une multiplication de temps par 6, et pas par 10. Après, les temps des 2 exécutions vont diminuer de 1/3 avant de plus ou moins se stabiliser. Cette diminution a lieu car même si le temps d'exécution de la fonction de lecture est 10% plus long en moyenne d'après nos tests entre 2 et 3 lecteurs, nous avons un lecteur supplémentaire, donc la lecture s'effectue plus rapidement.
Nous voyons néanmoins une légère augmentation entre 6 et 7 threads qui s'explique par le fait qu'il y a un lecteur en plus en attente active pendant que les écrivains sont occupés.

3 Performance des verrous

Nous avons implémenté les 3 différents verrous par attente active, et nous avons ensuite comparés leurs performances, représentées sur ce graphique.

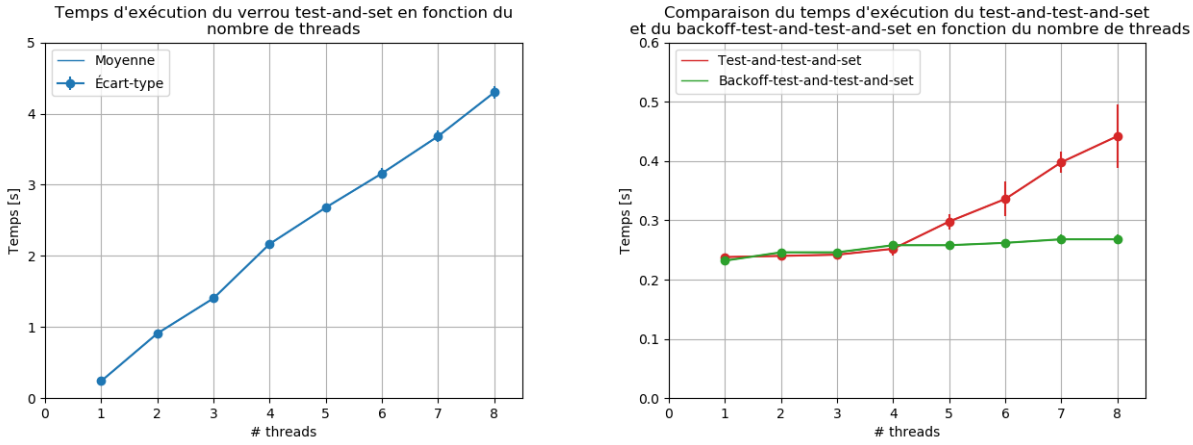

FIGURE 4 – Performance des verrous

Pour le test-and-set (graph 4, graphique de gauche), on observe une augmentation linéaire du temps d'exécution. Ceci peut s'expliquer par le fait que de nombreux threads sont en attente active, et sont donc bloqués dans des boucles `while`, qui testent continuellement la valeur du verrou. Ils monopolisent donc le processeur pour "aucune raison".

Pour le test-and-test-and-set (graph 4, graphique de droite), on constate que la performance diminue légèrement avec le nombre de threads, mais reste assez constante. La différence entre les performances d'un thread et de 8 threads reste minime, nous avons une différence de un peu plus de 0.2 seconde. On remarque donc que le test-and-test-and-set est bien plus efficace, car tant que le verrou reste à 1 (en position `"lock"`), il ne monopolise pas le processeur pour aller `xchg` la valeur du verrou.

Lorsque nous avons implémenté notre sémaphore, nous nous sommes donc logiquement servis de l'algorithme test and test and set pour réaliser notre programme, plutôt que d'utiliser test and set qui aurait été beaucoup plus lent.

Finalement, nous avons également implémenté le bonus backoff-test-and-test-and-set (graph 4, graphique de droite). On constate rapidement que peu importe le nombre de threads, la performance est toujours égale (à quelques centièmes de seconde près). Nous avons fait plusieurs tests afin de trouver nos valeurs d'attente minimale et maximale, et avons fini par trouver une fourchette allant de 2 à 13 000 μs (2μs à 13 ms). Ces valeurs nous ont fortement étonnés car nous nous attendions à obtenir des valeurs de l'ordre de la nano-seconde.

4 Conclusion

À la suite de ce travail, nous avons pu constater que la part d'aléatoire présente dans les threads, due au scheduler, joue un rôle important dans l'implémentation de nos codes. On peut constater sur la plupart des graphiques que pour un même code, pour un même nombre de threads, l'écart type est parfois très important. Or, théoriquement, pour un même code exécuté avec le même nombre de threads, les temps d'exécution devraient être semblables.

De même, lors de l'implémentation de notre sémaphore par attente active, nous avons rencontré un autre problème. Lors de nos tests, ceux-ci échouaient seulement 1 fois sur 10 environ. Cela posait problème car nous avons mit du temps à nous en rendre compte, étant donné qu'instinctivement, nous ne pensions pas à faire nos tests 10 fois de suite, mais nous nous contentions de les exécuter 2-3 fois. Nous avons finalement déduit que ce problème était du au scheduler et à l'ordre par lequel les différents threads s'éxécutent, ce qui nous a permis de corriger nos codes. Nous avons aussi passé beaucoup de temps à comprendre d'où venait l'énorme augmentation de temps du problème des lecteurs-écrivains. C'est grâce à de multiples mesures de temps que nous avons pu déterminer la source du problème.