Université Catholique de Louvain-la-neuve

LINFO1252 - Systèmes Informatiques
Projet 1 : Programmation multi-threadée et évaluation de performances

Groupe 7 - 2 :
Louis Ridelle : 65481700
Ethan Samson : 79021600

Tuteur : Florian Vranckx

Professeur : Etienne Rivière

25 novembre 2020
Année académique 2020 - 2021

1 Introduction

Ce projet avait pour but de mettre en oeuvre différents algorithmes de synchronisation de threads et d'en évaluer les performances. Cette évaluation s'est faite à travers trois problèmes typiques : les philosophes, les lecteurs/écrivains et les producteurs/consommateurs.

Le projet était divisé en deux tâches. Dans la première, nous avons codé une implémentation du problème des philosophes, des producteurs/consommateurs et des lecteurs/écrivains à l'aide des threads POSIX.
Dans la deuxième, nous avons créé nos propres primitives de synchronisations par attente active en nous servant de *l'assembleur en ligne*. Nous avons ensuite comparé leurs performances avec celles des POSIX en les utilisant pour les 3 problèmes de la première tâche.

Nous devions ensuite créer des script bash et python pour évaluer et analyser les performances des différents programmes.

2 Structure du projet et choix d'implémentation

Pour plus de clarté, nous avons décidé de séparer chaque tâche dans un répertoire distinct avec un `Makefile` par tâche, de plus chaque problème a son propre fichier. La structure du projet est expliquée plus en détail dans le `README` qui donne plus d'information sur le projet d'un point de vue général.

De plus, il nous a semblé pertinent d'y intégrer un `Makefile` qui permet d'automatiser correctement la compilation des fichiers et le lancement des tests. Dans le but de garder les 2 tâches bien distinctes, nous avons créé un `Makefile` propre à chacune d'elle qui est dirigé par le `Makefile` principal. Les commandes globales de celui-ci sont reprises dans le `README` que nous vous invitons à lire.

Nous avons également fait plusieurs choix d'implémentation lors de l'écriture de certains fichiers. Nous allons survoler ceux-ci pour permettre une meilleure lisibilité de notre code.
Vu que chaque coeur de notre machine a 2 threads, nous avons décidé de calculer uniquement le temps pour un nombre pair de threads.

Pour être sûrs d'avoir un avantage du multithreading assez visible sur les courbes, nous avons décidé de mettre la plupart du temps de calcul en dehors des sections critiques pour les algorithmes producteurs/consommateurs et lecteurs/écrivains. Nous avons donc placé les boucles *while* aléatoires sur les fonctions `produce()`, `consume()`, `prepare_data()` et `process_data()` (Voir code source).

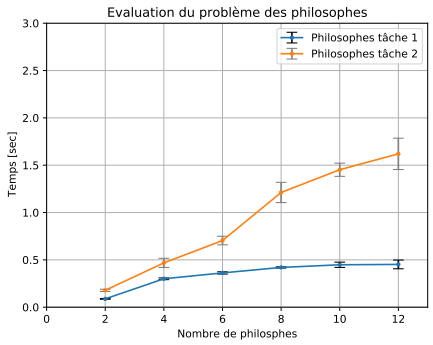
Pour l'implémentation des verrous en attente active *test and set* et *test and test and set*, nous avons décidé d'utiliser une structure plutôt qu'une variable globale pour ne pas être limités dans le nombre de mutexes et de sémaphores. le verrou *test and set* (la fonction `my_mutex_lock()`) est juste une adaptation du fragment de code en assembleur vu en cours (voir page 38 des slides).
Pour le verrou *test and test and set*, nous nous sommes inspiré du pseudo code wikipédia[1]. Nous avons créé une fonction auxiliaire `test_and_set(my_mutex* mutex)` qui prend un mutex en argument, met son état sur 1 avec l'instruction `xchgl` et renvoie son ancien état. On se sert du retour de cette fonction pour boucler sur le même principe que `my_mutex_lock()`. Mais au lieu de boucler en permanence sur cette fonction qui va surcharger le programme d'appels à `xchgl`, on a créé une boucle à l'intérieur de la première qui teste l'état du mutex en mode lecture uniquement et boucle tant que son était est égale à 1 (mode bloquant). Pendant cette boucle, aucun appel à `xchgl` n'est effectué et donc la section critique des autres programmes n'est pas rallongée.
Pour des raisons évidentes, la lecture et l'écriture de l'état du sémaphore est protégée par nos mutex. Nous avons aussi utilisé le principe du *test and test and set* pour la fonction `wait()` et de l'adapter aux sémaphores.

[1] lien : https://en.wikipedia.org/wiki/Test_and_test-and-set

3 Analyse des résultats

3.1 Problème des Philosophes

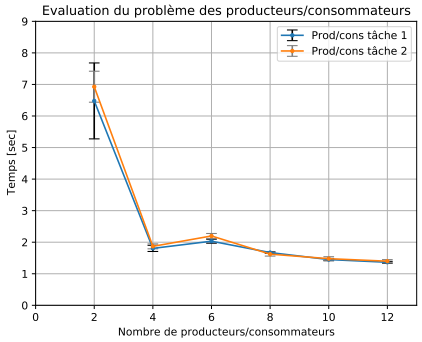
Dans notre implémentation, les philosophes sont représentés par des threads et la baguette par un tableau de mutex. Pour plus d'informations sur le le problème des philosophes, cliquez ici.



On constate ici une augmentation du temps proportionnel au nombre de philosophes, alors que théoriquement, 12 philosophes qui effectuent un certain nombre de cycles en même temps prennent autant de temps qu'un seul philosophe qui effectue ce même nombre de cycle. Il y a donc une perte de performance due à la synchronisation.
On constate aussi une grosse différence entre notre implémentation des mutex et celle des POSIX. Ce résultat nous parait cohérent, vu que, comme vu en cours, l'attente active est moins efficaces quand le temps d'attente entre 2 SC[2] est court par rapport aux SC elles-mêmes. En effet, comparé à l'implémentation des autres problèmes, il n'y a aucune attente entre 2 tentatives d'accès à la SC (l'attente est représentée par la fonction `mange()`, qui est vide donc instantanée).

3.2 Problème des Producteurs - Consommateurs

Ceux-ci, représentés par les threads se partagent les données via un buffer de taille fixe. Pour plus de détails, cliquez ici.

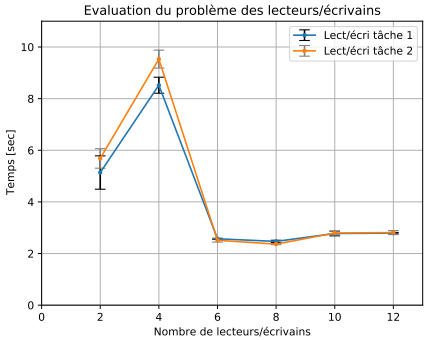


Vu que nous avons placé la boucle aléatoire sur les fonctions `produce()` et `consume()`, la section critique des producteurs et des consommateurs est très courte par rapport au reste de l'algorithme. On peut voir sur le graphe que la différence qu'il y avait entre notre implémentation et celle POSIX avec le problème des philosophes a totalement disparu, ce qui concorde avec le raccourcissement de la longueur de la section critique par rapport au reste.

[2] Section Critique

3.3 Problème des Lecteurs - Ecrivains

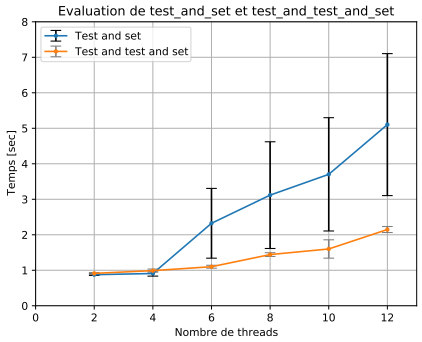
Au même titre que les producteurs et consommateurs, les écrivains et lecteurs sont les threads qui travaillent sur des données. Celles-ci sont reprisent dans une base de données partagée. L'idée de ce problème est qu'il peut y avoir un nombre infini de lecteur qui lisent en même temps la base de donnée, mais qu'un écrivain doit toujours être seul (sans lecteur non plus); plus de détails sur ce problème via ce lien.



Les résultats sont plus ou moins similaires que ceux du producteurs/consommateurs, au détail près qu'il y a un gros pic avec 2 lecteurs/2 écrivains. On pense que le problème vient de la fonction `void reader(void)`, à la ligne `sem_wait(&reader_block)`. Notre hypothèse est que l'appel à `sem_wait(&reader_block)` empêche aussi les lecteurs de lire simultanément la base de données.
Nous n'avons malheureusement pas trouvé une autre méthode pour donner la priorité aux écrivains avant la deadline de remise du projet. Notre piste était de mettre une condition sur l'appel `sem_wait(&reader_block)`, en faisant appel à `sem_wait(&reader_block)` uniquement si le nombre d'écrivains est strictement plus grand que 0. Le problème était que cette condition rend la condition de l'appel à `sem_post(&reader_block)` très compliquée.

3.4 Test and set - Test and test and set

Ici, nous parlons de notre propre implémentation de verrous par attente active. Nous nous sommes basée sur 2 algorithmes, le *test and set* (TAS[3]) et le *test and test and set* (TATAS[4]). L'idée principale est de créer une variable `lock` dont la valeur sera changée de manière atomique grâce à *l'assembleur en ligne*.



Les résultats obtenu semble tout à fait réaliste étant donné que le TATAS est une amélioration du TAS.

En effet, on voit que le temps du TAS augmente nettement avec le nombre de thread. Durant le test, les threads ne pouvant pas accéder à la SC boucle en créant des appels répétitifs à l'instruction `xchgl`.

[3]Documentation sur l'algorithme TAS
[4]Documentation sur l'algorithme TATAS

Ceux-ci saturent le bus d'appels qui bloquent le processeur ce qui joue sur les performance générales de l'ordinateur. On peut aussi remarquer que l'écart-type est assez important avec le nombre de thread grandissant. Notre hypothèse est que les appels permanents à `xchgl` rendent la vitesse exécution très dépendante des autres processus entrain de s'exécuter sur la machine.
Si nous regardons l'algorithme TATAS on peut remarquer un nette amélioration du temps d'exécution. Celle-ci est dû à une nouvelle condition qui évite des appels inutiles à `xchgl`. Cette condition ne fait qu'une lecture de l'état du mutex et donc n'exécute aucune instructions bloquante.

4 Conclusion

Au cours de ce projet nous avons pu approfondir nos connaissances, notamment sur le fonctionnement des processeur. L'utilisation de *l'assembleur en ligne* nous a permit de mieux comprendre les opérations atomiques qu'exécutait le processeur. Cela nous a aussi donné un aperçu plus concret et pratique de la gestion des registres, de la mémoire et l'échange des données à travers ceux-ci.

Nous avons beaucoup appris sur l'implémentation de tests de performance. Dans la même idée et par soucis de clarté, nous avons enrichi nos connaissances sur la création de `Makefile` et `sous-Makefile` plus complets. En effet, nous nous sommes retrouvé face à un projet qui mettait en lien plusieurs fichiers situés dans différents dossiers. Il nous a semblé presque indispensable de créer des `Makefile` complets afin de faciliter la compilation, l'exécution des tests et l'affichage des résultats.