LINFO1252 - Projet 1

Aydin Matya
NOMA: 37592100

Debelle Thomas
NOMA: 30002100

I. Introduction

Dans le cadre du cours LINFO1252, il nous a tout d'abord été demandé d'implémenter des problèmes classiques de programmation concurrente tels que le problème des philosophes ou du producteur-consommateur.
Ensuite, il a fallu implémenter des mises en oeuvres de synchronisation de threads basées sur des attentes actives et des opérations atomiques et de s'en servir pour les problèmes de programmation concurrente implémentés précédemment.
Les rapidités d'exécution de ces différentes situations ont été mesurées en fonction du nombre de threads et sont commentées dans les sections suivantes.

II. Implémentation et résultats

Remarque préalable: Nous sommes conscients du manque de sens physique des plots de valeurs continues dans le cadre de mesures par rapport au nombre de threads qui ne peut prendre que des valeurs entières. Ceux-ci sont fournis afin de pouvoir observer des tendances et de rapidement distinguer notre implémentation de `lock` et celle de la librairie standard.

A. Partie 1

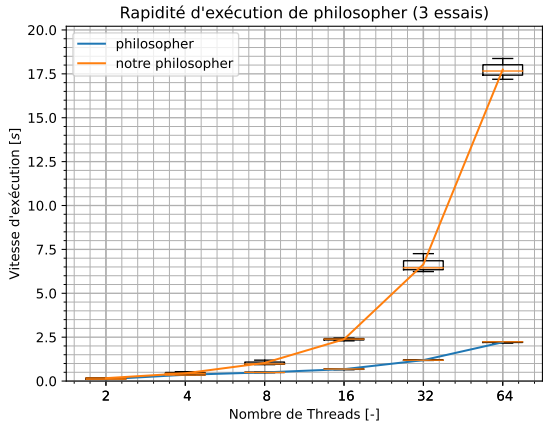

Fig. 1. Plot du problèmes de philosophes avec les 2 versions de synchronisation

1) Problème des Philosophes

Le problème des philosophes est le suivant, nous avons N philosophes autour d'une table qui possèdent N baguettes qui sont chacune positionnée entre chaque philosophe. Pour manger leur bol de riz, ils doivent posséder 2 baguettes dont chacune est partagée respectivement avec les voisins de droite et gauche. Il faut donc que les philosophes se partagent les baguettes et éviter le cas où tout le monde en a une et personne ne peut manger.

Outre cette représentation imagée du problème, ce problème est un cas où l'utilisation de l'exclusion mutuelle (*mutex*) est primordiale. Il faut bloquer l'accès à une baguette (ici, un *mutex*) dans un ordre spécifique pour éviter le cas de *deadlock*[1]. On va toujours bloquer la baguette de gauche puis celle de droite à part quand on est au N-ème philosophe qui lui bloquera celle de droite puis celle de gauche pour éviter le deadlock. On n'effectue pas l'action de *manger* pour des questions de rapidité. Finalement, on libère la baguette de gauche puis celle de droite.

Chaque philosophes (ici, chaque *threads*) réalisent cette tâche 1000000 de fois via la fonction `philosophe`. On voit une augmentation du temps d'exécution qui est linéaire ce qui fait du sens car pour N threads on effectue $N \cdot 1000000$ actions de manger donc de bloquer des mutex c'est-à-dire de rentre en *section critique*. On peut être surpris vis-à-vis du fait que notre programme reste linéaire et ce même pour 64 threads mais on peut supposer que la machine qui nous a été donné possède 32 coeurs physiques et 64 coeurs virtuels ce qui expliquerait la conservation de cette augmentation linéaire même pour 64 threads.

Nous réalisons les threads via les threads POSIX et utilisons des mutex de la libraire `pthread.h`.

L'analyse du temps d'exécution basé sur notre algorithme de synchronisation est détaillée en II-B4.

2) Problème du Producteur-Consommateur

Nos 2 fonctions `producer` et `consumer` (dont l'exécution est répartie de manière égale par rapport au nombre de threads) produisent en permanence via un `while(true)` dont la condition d'arrêt est l'atteinte du nombre de productions (=nombre de consomnation) demandé. Pour cela, une variable `item_produced` (resp. `item_consumed`) est incrémentée à chaque production/consomnation. Cette opération se réalise en section critique protégée par un *mutex*. On y trouve

[1] *deadlock*: le programme ne peut plus avancer car chacun des *threads* attendent que des *mutex* se libèrent mais cela n'arrivera jamais.

également la mise à jour d'une variable `count` qui fournit l'indice du premier emplacement libre dans le buffer.

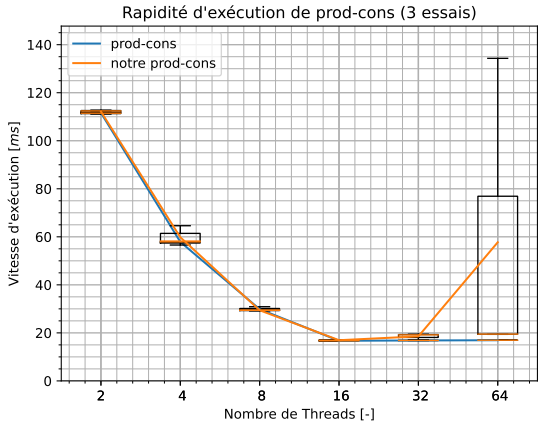

Fig. 2. Plot du producteur consommateur avec les 2 versions de synchronisation

Le graphe ci-dessus présente le temps d'exécution du producteur-consommateur en fonction du nombre de threads. En ce qui concerne la synchronisation basée sur les threads POSIX (ligne bleue), on constate une décroissance du temps d'exécution. En effet, le nombre de sections critiques à exécuter est inchangé mais si celles-ci sont exécutables par un plus grand nombre de threads dont le parallélisme est garanti par le nombre de coeurs physiques de la machine, alors le temps d'exécution total diminue. Cependant, à partir de 16 threads, le temps d'exécution devient constant, ce qui témoigne de l'optimisation de l'algorithme de synchronisation des mutex POSIX: le surcoût en est négligeable à l'échelle de nos mesures et de l'application.
L'analyse du temps d'exécution basé sur notre algorithme de synchronisation est détaillée en II-B4.

3) Problème des Lecteurs et Écrivains

Nous avons implémenté la solution donnée lors du TD S8. Nos 2 fonctions `reader` et `writer` sont réparties de manière égale et s'exécutent en permanence via un `while(true)` dont la condition d'arrêt est la réalisation du nombre de lectures et d'écritures demandés. Nous nous sommes servis de 3 *mutex* et 2 *semaphore*. Le problème de *starvation* des écrivains a été réglé en leur donnant la priorité: une variable partagée `writecount` protégée par un *mutex_write_count* permet de faire attendre les lecteurs via à un appel `sem_wait(rsem)` lorsqu'elle est incrémentée. L'exclusion par rapport aux autres écrivains est quant à elle garantie par un sémaphore *wsem*.

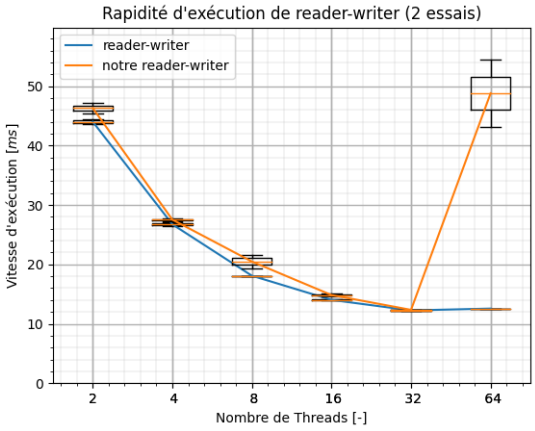

Fig. 3. Plot du lecteur-écrivain avec les 2 versions de synchronisation

Le graphe ci-dessus illustre le temps d'exécution total du problèmes des lecteurs-écrivains en fonction du nombre de threads.
Similairement au cas des producteurs-consommateurs, on observe une décroissance du temps d'exécution tant que le nombre de threads utilisés n'atteint pas le nombre de coeurs physiques. En effet, avec un nombre de threads inférieur à 32, le parallélisme d'exécution n'est pas exploité à son plein potentiel.
On observe ensuite qu'à partir de 32 threads, le temps d'exécution devient constant: le parallélisme est maximisé et l'algorithme de synchronisation des threads et sémaphore POSIX n'admet pas de surcoût.
L'analyse du temps d'exécution basé sur notre algorithme de synchronisation est détaillée en II-B4.

B. Partie 2

1) Implémentation du Test-and-Set

Nous nous sommes fortement inspirés du code fourni dans le syllabus que nous avons divisé en 2 fonctions `lock` et `unlock`. La fonction `lock` exécute une boucle écrite en assembleur: l'instruction atomique `xchgl` est exécutée sur un registre contenant la valeur 1 et une zone de mémoire *verrou*. Si après cet échange, la valeur contenue dans le registre est de 1, cela signifie que la zone mémoire était occupée et qu'un thread était en section critique. On réexécute alors la suite d'opérations ci-dessous. Si le registre contient 0, le thread a pu rentrer en section critique et l'exécution de `lock` se termine pour ce thread.
L'implémentation de `unlock` est quant à elle plus simple: On vient simplement placer la valeur 0 dans *verrou*. Cela est également réalisé de manière atomique via l'instruction `xchgl` pour éviter que la valeur 0 contenue dans le registre avec lequel l'échange s'opère soit mise à 1 par un autre thread, ce qui pourrait causer un problème de *deadlock*.

2) Implémentation du Test-and-Test-and-Set

Cet algorithme de synchronisation consiste en une amélioration de `test-and-set` basée sur le fait que lorsque le *verrou* est à 1, on ne tente pas l'opération `xchgl` car elle limite la contention du bus. Le cas échéant, on privilégiera une attente active tant que *verrou* ne passe pas à 0.

La fonction `unlock` reste quant à elle inchangée. On observe

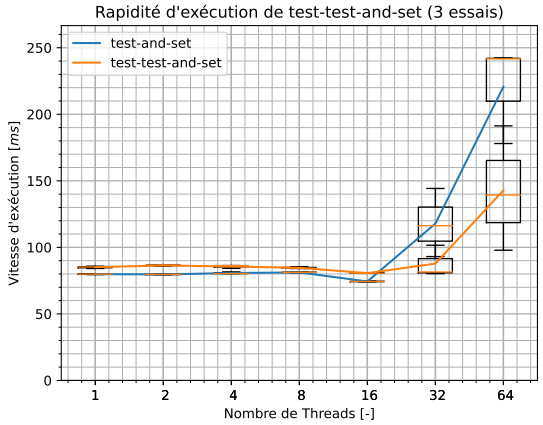

Fig. 4. Plot de test-and-set et test-test-and-set

des temps d'exécution similaires et à faible variance entre les 2 algorithmes pour des nombres de threads inférieurs ou égaux à 32. Cependant, à partir de 64 threads, l'écart entre *test-and-set* et *test-test-and-set* se creusent: *test-test-and-set* a un temps d'exécution inférieur d'environ 100 ms. Cela témoigne de son bon fonctionnement: les attentes actives réalisées à la place des appels continus à `xchgl` lorsque la valeur lue du *verrou* indique qu'il est occupé limite la contention du bus et par conséquent, ne ralentit pas l'exécution des autres threads.

On notera que la différence de performances se fait principalement ressentir qu'à partir de valeurs supérieures au nombre de coeurs physiques de la machine (32): valeur à partir de laquelle chaque thread ne peut plus occuper un seul coeur et le scheduler doit alterner l'exécution de ceux-ci. On observe également une variance plus faible pour `test-test-and-set`.

3) Implémentation des Sémaphores

Notre implémentation utilise 3 fonctions similaires à celles de la librairie standard: `my_sem_init`, `my_sem_wait` et `my_sem_post`. `my_sem_init` permet d'initialiser une structure `my_sem_t` en précisant le nombre d'*autorisations* désirées décrit par l'élément `amount`.

`my_sem_wait` consomme une autorisation en décrémentant `amount` si cette valeur est strictement supérieure à 0. Cette opération est réalisée dans une section critique implémentée par nos fonctions `lock` et `unlock` décrite en II-B1. Si `amount` n'a pas pu être décrémenté, le thread est mis en attente active via un `while(true)`.

Finalement `my_sem_post` incrémente simplement `amount`, également de manière exclusive, libérant ainsi un thread en attente active si `amount` est inférieur ou égal à 0.

4) Adaptation des 3 Programmes

Pour chaque problème, nous avons remplacé les mutex POSIX par notre implémentation de synchronisation utilisant l'algorithme du *test-and-set* décrit en II-B1. Similairement, nous avons remplacé les sémaphores POSIX par notre implémentation décrite en II-B3. Le reste des programmes n'a pas été changé.

a) Adaptation du problème des Philosophes

Nous avons donc remplacé les `pthread_mutex_lock` par nos `lock` qui sont implémentés avec une stratégie *test-and-set*. On voit tout de suite que cette stratégie est extrêmement coûteuse en ressource et donc en temps. En effet, l'opération atomique va bloquer notre bus et envoyer des signaux d'invalidation aux processus utilisant une variable partagée avec ce dernier. On conserve tout du moins notre propriété de linéarité. Notre implémentation du `lock` est environ 2.625 fois plus lente que celle de la librairie `pthread.h`.

b) Adaptation du problème du Producteur-Consommateur

Le temps d'exécution est similaire à celui de l'implémentation standard pour un nombre de threads inférieur à 32. A partir de cette valeur, on observe une croissance linéaire du temps d'exécution. En effet, notre implémentation ne réalise pas d'appels systèmes et le scheduler peut par conséquent allouer du temps à un thread pour qu'il exécute une attente active `while(true)`, ce qui ne fait pas avancer l'exécution du programme.

Les appels constants à l'instruction atomique `xchgl` durant cette attente ralentissent également l'exécution de tous les autres threads.

c) Adaptation du problème des Lecteurs et Écrivains

Les observations sont similaires à celles du problème du Producteur-consommateur. On remarque cependant une croissance significativement plus importante (croissance toujours linéaire mais avec une pente bien plus raide) qui est probablement dû au fait qu'un écrivain doit obtenir l'accès exclusif.

III. Conclusion

Au cours de ce projet, nous avons pu comprendre plus en profondeur l'amélioration de performances qui peut être atteinte en *multi-threadant* un programme. Nous avons également été confronté aux différents problèmes qui peuvent subvenir via l'implémentation de ce paradigme et réalisé l'importance d'un bon algorithme de synchronisation étant donné le coût des attentes actives et de l'opération atomique `xchgl` sur les performances. ■