LINFO1252 - Systèmes informatiques

Rapport projet 1

Introduction

Lors de ce projet, nous avons appris à analyser les performances de primitives de synchronisation entre plusieurs threads (*mutex* et *sémaphores*) ainsi que les verrous avec attente active (aka. *spinlocks*) que nous avons dû implémenter.
Nous allons dans un premier temps présenter notre implémentation et les performances des *spinlocks* avec les deux algorithmes *test and set* et *test and test and set* ainsi que les *sémaphores* basées sur ceux-ci.
Ensuite, nous utiliserons ces *spinlocks* et *sémaphores* pour résoudre les trois tâches : problème des philosophes, des producteurs/consommateurs et des lecteurs/écrivains. Nous comparerons les performances des deux algorithmes *test and set* et *test and test and set* ainsi que les performances des *mutex* de la librairie *pthread_h*.

Verrou par attente active (*spinlock*)

Dans cette section, nous allons montrer nos deux primitives de synchronisation par attente active, comparer leur performances et enfin présenter deux interfaces *sémaphores* basées sur celles-ci.

Test and set

Un pseudocode pour cet algorithme est décrit ci-dessous :

```
1 function test_and_set(boolean lock) {
2   boolean old_value = lock;
3   lock = true;
4   return old_value
5 }
```

Liste 1. – Pseudo-code pour l'algorithme *test and set*

Notons que les deux étapes aux lignes 2 et 3 de Liste 1 doivent impérativement être réalisées de manière atomique. De plus, le keyword `volatile` est nécessaire lors de l'accès/la modification du lock car celui-ci peut avoir des comportements qui ne peuvent être prédits par le compilateur (car il est modifiable par plusieurs threads). En mettant ce keyword, nous évitons ainsi que le compilateur optimise l'accès à la valeur du lock en mémoire en utilisant une version mise en cache dans un registre, pouvant induire une boucle infinie.
Nous avons défini une interface qui est reprise ici :

```c
1 typedef int Mutex_t;
2 int test_and_set(Mutex_t *mutex);
3 void lock_TAS(Mutex_t *mutex);
4 void unlock_TAS(Mutex_t *mutex);
```

Liste 2. – Interface du *spinlock test and set*

La fonction `test_and_set` prend un pointeur vers un `Mutex_t` et exécute l'algorithme Liste 1 de manière atomique grâce à l'instruction `xchg` qui permet d'échanger deux

adresses de manière atomique. La procédure `lock_TAS` effectue un *spinlock* en attente d'un changement de valeur du lock de 0 à 1. Enfin, la procédure `unlock_TAS` permet de mettre la valeur du lock à 0, toujours de manière atomique.

Test and test and set

L'algorithme *test and test and set* se base sur l'algorithme précédent pour tenter de mettre le lock à 1. La différence est que cet algorithme boucle en attente que le lock passe à 0 avant de tenter d'appeler `test_and_set`. Cela permet d'éviter de continuellement effectuer des instructions atomiques qui peuvent être un peu plus lentes que leur équivalents non-atomiques.

```
1 boolean lock := UNLOCKED;
2 procedure lock_test_and_test_and_set() {
3   do {
4       while (lock == LOCKED) {skip} // non-atomique
5   } while (test_and_set(lock) == LOCKED)  // atomique
6 }
```

Liste 3. – Pseudo-code pour le *spinlock test and test and set*

Notre interface comprend donc uniquement les procédures permettant de verrouiller et déverrouiller le lock car la fonction `test_and_test` a déjà été implémenté :

```c
1 void lock_TATAS(Mutex_t *mutex);
2 void unlock_TATAS(Mutex_t *mutex);
```

Liste 4. – Interface du *spinlock test and test and set*

Comparaison des performances

Nous avons également évalué les performances de nos deux primitives de synchronisation sur un test basique. Nous lançons un programme contenant une section critique qui effectue simplement une boucle `for(int i = 0; i < 10000; i++)`. Chaque thread exécute $\frac{32768}{N}$ fois leur section critique où N est le nombre de threads. Nous lançons l'expérience 5 fois pour chaque nombre de threads dans $\{1, 2, 4, 8, 16, 32\}$.

Voici un graphe représentant le temps moyen et la déviation standard entre chaque itération du test en fonction du nombre de threads :

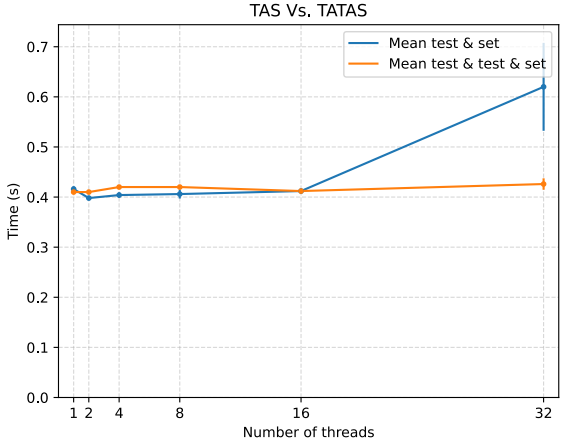

Fig. 1. – Temps moyens et déviations standards des exécutions du test de performance en fonction du nombre de thread

Nous pouvons déduire du graphe que le spinlock *test and test and test* est plus performant car le temps d'exécution est pratiquement constant. En effet, un algorithme d'exclusion mutuelle « idéal » donnerait lieu à un temps d'exécution constant. Ici, l'augmentation du temps d'exécution est due au temps des opérations des spinlocks. Les instructions atomiques étant lente, le spinlock implémentant *test and set* est donc moins performant.

Interface sémaphore

Nous avons également dû implémenter une interface sémaphore se basant sur nos deux primitives de synchronisation.

Les deux interfaces (fort similaires) sont reprises ci-dessous :

```c
#include "test_and_set.h"
typedef struct {
    volatile int value;
    Mutex_t* mutex_tas;
} sem_tas_t;
void sem_tas_init(sem_tas_t* sem, int value);
void sem_tas_destroy(sem_tas_t* sem);
void sem_tas_wait(sem_tas_t* sem);
void sem_tas_post(sem_tas_t* sem);
```

Liste 5. – Interface du sémaphore *test and set*

```c
#include "test_and_test_and_set.h"
typedef struct {
    volatile int value;
    Mutex_t* mutex_tatas;
} sem_tatas_t;
void sem_tatas_init(sem_tatas_t* sem, int value);
void sem_tatas_destroy(sem_tatas_t* sem);
void sem_tatas_wait(sem_tatas_t* sem);
void sem_tatas_post(sem_tatas_t* sem);
```

Liste 6. – Interface du sémaphore *test and set*

Notons l'utilisation du keyword `volatile` qui permet au compilateur de ne pas faire d'optimisation pour la boucle qui vérifie la valeur du sémaphore.

Tâches

Pour chacune des trois tâches, nous avons lancé 5 itérations pour chaque nombre de threads dans $\{2, 4, 8, 16, 32\}$. Pour les tâches qui requièrent deux arguments, nous avons divisé le nombre de threads de manière égale. Nous avons ensuite calculé la moyenne ainsi que la déviation standard pour ces 5 itérations et avons représenté cela sur les graphes suivants en fonction du nombre de threads.

Philosophe

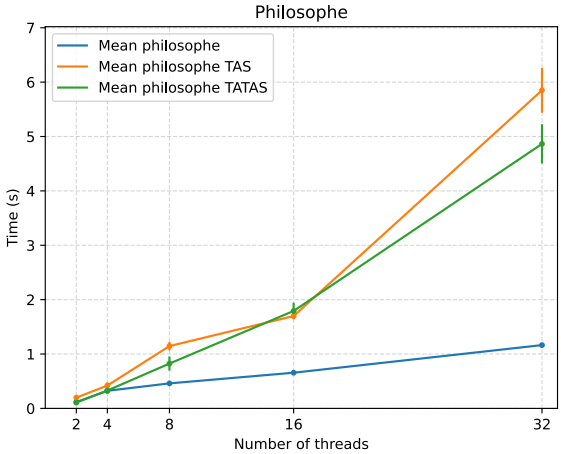

Fig. 2. – Temps moyens et déviations standards des exécutions du programme « philosophe » en fonction du nombre de thread

Nous observons que notre implémentation des mutex appliquée au problème des philosophes a des performances qui diminuent linéairement au plus il y a de threads, avec une plus grande pente qu'avec les mutex `phtread`. Dans ce problème, il y a beaucoup de contention[1] , ce qui favorise les mutex `phtread` qui utilisent l'appel système `futex` du kernel et qui sont plus optimisés en cas de contention[1] élevée.

Producteurs/Consommateurs

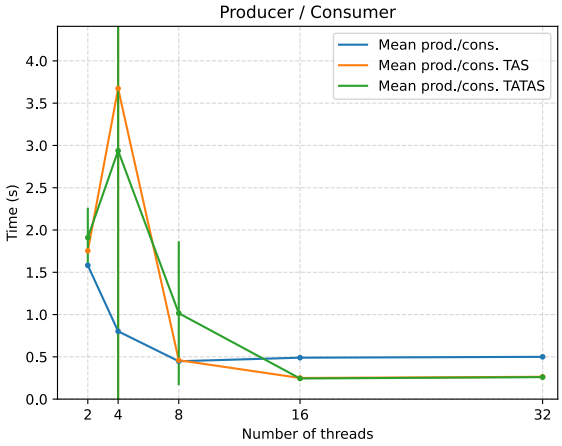

Fig. 3. – Temps moyens et déviations standards des exécutions du programme « producteur/consommateur » en fonction du nombre de thread

Ce problème possède une contention[1] plus faible car les threads accèdent souvent des zones différentes du buffer. Nous pouvons observer qu'à terme notre implémentation des mutex est plus rapide que celle de `pthread`. Notre théorie est que, dans un problème avec une contention[1] plus faible, le coût des appels systèmes tels que `futex` est plus élevé que l'attente active du `test and set` et induit des coûts de synchronisation. Ces coûts sont négligeables sur un cas avec peu de threads mais visibles quand il y en a plus comme dans le cas à 16 et 32 threads.

[1]situation où plusieurs threads ou processus tentent simultanément d'accéder à une ressource partagée

Lecteurs/écrivain

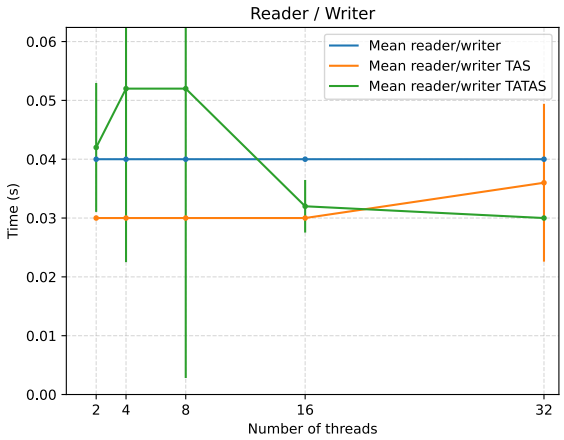

Fig. 4. – Temps moyens et déviations standards des exécutions du programme « lecteur/écrivain » en fonction du nombre de thread

Nous observons des caractéristiques de performance équivalentes au problème précédent explicables par leur similarité.

Conclusion

Bien que l'implémentation des mutex avec attente active puisse être plus rapide que `pthread_mutex_t` dans certains programmes avec peu de contention[1], elle présente beaucoup de désavantages par rapport à une implémentation plus conventionnelle. Son plus gros défaut est qu'elle gâche plein de cycles du CPU à attendre dans une boucle. Un avantage est que cette implémentation est simple à implémenter et qu'elle ne dépend pas d'appels systèmes. En conclusion, il convient mieux d'utiliser une implémentation des mutex qui repose sur des appels système au kernel qu'une implémentation qui utilise l'attente active dans un programme général.