LINFO1252 — Projet 1
(Groupe 01 — Charleroi)

Kaan Akman Houari-Boumeddien Benkechida

1 décembre 2025

1 Introduction

Dans ce projet, nous avons appris à évaluer et à expliquer la performance de plusieurs applications via l'utilisation de plusieurs threads et des primitives de synchronisation (qu'on a implémentées par nous-mêmes dans la deuxième partie du projet).

Ces applications correspondent aux problèmes vus en cours et en TD : le problème des philosophes, producteurs/consommateurs avec un buffer de taille fixe et finalement lecteurs/écrivains avec la priorité donnée aux écrivains.

Notre but ici est d'analyser nos choix d'implémentations qu'on a effectué et de mesurer les performances en se basant sur le temps d'exécution et le nombre de threads utilisés, tout en tenant compte des limitations dues aux nombres de cœurs limités de notre machine virtuelle.

2 Première partie : Utilisation des primitives de synchronisation POSIX

Pour cette première partie, on a utilisé les primitives de synchronisations POSIX pour les problèmes des philosophes, producteurs/consommateurs et lecteurs/écrivains avec la bibliothèque en C de POSIX. On a utilisé des mutex et sémaphores avec <pthread.h> et puis on a évalué leur performance, en fonction du nombre de cœurs utilisés, sur des plots en Python. Ici, notre but était d'analyser l'allure des courbes sur le graphique lorsqu'on augmente le nombre de threads sur une machine virtuelle à 4 cœurs.

2.1 Tâche 1.1 — Problème des Philosophes

2.1.1 Mise en œuvre

Pour ce problème, on a utilisé un mutex pour chaque baguette. Pour éviter qu'il y ait un deadlock (interblocage — lorsque chaque philosophe tient sa baguette gauche et attend à l'infini que celui de droite se libère). On a forcé un ordre pour prendre la baguette : chacun des philosophes essaie de verrouiller la baguette avec l'identifiant le plus petit en premier. Le nombre de philosophes (N) est obtenu à partir de la ligne de commande, chaque philosophe effectue 1 000 000 de cycles "penser/manger". Mais il n'y a pas de méthodes penser() ou manger() dans l'implémentation pour ne pas nuire la performance des opérations de synchronisation (donc ces actions sont immédiates).

2.1.2 Analyse des performances

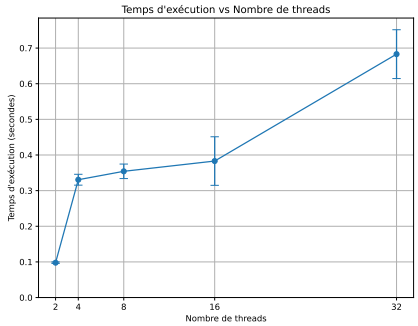


FIGURE 1 – Temps d'exécution du problème des philosophes

On analyse que le temps d'exécution croît en augmentant le nombre de threads. Comme l'exécution des cycles penser/manger sont immédiates, ce qu'on analyse c'est surtout la gestion des mutex et les changements de contexte. Donc plus le nombre de threads augmente, plus la limitation sur les verrous est importante (ceci dégrade la performance).

2.2 Tâche 1.2 — Producteurs/Consommateurs

2.2.1 Mise en œuvre

Pour ce problème, on a utilisé un buffer de taille N=8, contenant des entiers (int), qui est protégé par un mutex pour garantir l'exclusion mutuelle. La gestion des places est assurée par deux sémaphores : 'empty', initialisé à N (places libres), et 'full', initialisé à 0 (places occupées). Entre chaque consommation ou production, il y a un thread qui simule un traitement en utilisant la ressource CPU (avec une boucle for à 10.000 itérations). Ce traitement se réalise en dehors de la section critique pour permettre le parallélisme et le nombre d'éléments produits/consommés est toujours de 131 072, divisé en tâche équitable entre les threads.

2.2.2 Analyse des performances

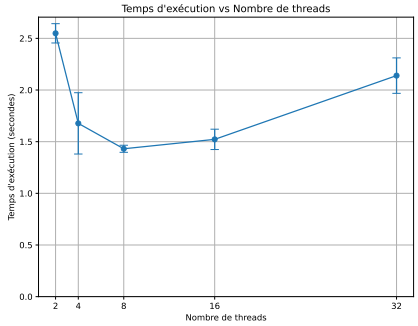


FIGURE 2 – Temps d'exécution de producteurs/consommateurs

On constate que le temps d'exécution chute lorsqu'on passe de 2 à 4 threads. Comme on a 4 cœurs dans notre machine virtuelle, celui-ci illustre bien qu'on arrive à saturation à 4 threads, en d'autre terme c'est ici que le parallélisme atteint sa capacité maximale. Au delà des 4 threads, on aura une augmentation du temps d'exécution car les threads devront être partagés sur les 4 cœurs limités et la performance ne s'améliorera plus.

2.3 Tâche 1.3 — Lecteurs/Écrivains

2.3.1 Mise en œuvre

Pour ce problème, on a placé une priorité aux écrivains comme dans le TD pour éviter la famine des écrivains. car les lecteurs privatisent la ressource. On a implémenté un mutex en plus, nommé 'z', qui va pouvoir forcer l'attente des lecteurs tant qu'il n'y ait pas eu le signal venant de l'écrivain qui prend sa sortie. Cela permettra d'éviter que les lecteurs puisent la ressource abondamment sans laisser la chance aux écrivains. Ici, contrairement au problème 1.2, un écrivain/lecteur simule un accès en écriture ou lecture à la base de données (avec une boucle for à 10 000 itérations) et il n'y a pas d'attente entre les deux tentatives d'accès. Cette fois-ci, le traitement se fait dans la section critique. C'est un accès exclusif aux écrivains et concurrent pour les lecteurs.

2.3.2 Analyse des performances

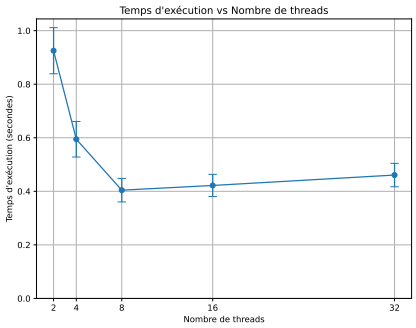


FIGURE 3 – Temps d'exécution de lecteurs/écrivains

On analyse une chute notable dans le temps d'exécution jusqu'à 8 threads.

— **Jusqu'à 8 threads :** Comme le nombre de threads est divisé équitablement (exemple, pour 8 threads : 4 lecteurs et 4 écrivains), les 4 lecteurs peuvent accéder simultanément à la ressource. Cela permet d'occuper pleinement les 4 cœurs de notre machine virtuelle durant la lecture.

— **Au-delà de 8 threads :** La performance va perdre en performance et le temps d'exécution va augmenter légèrement due à la saturation des 4 cœurs.

3 Deuxième partie : Mise en œuvre des primitives de synchronisation par attente active

Dans cette deuxième partie, on a implémenté nos propres mécanismes de synchronisation dans l'environnement d'instruction machine. Contrairement à l'utilisation des primitives POSIX où les threads étaient en état dormants, ici nos implémentations vont utiliser une boucle infinie (while(1)) qui va sans cesse vérifier si la ressource est disponible ou pas (attente active). On a pu réaliser à :

— Implémenter des verrous (spinlocks) en utilisant du code assembleur inline et l'instruction atomique 'xchgl' pour swap directement une valeur avec la variable globale.

— Comparer les deux logiques de verrouillage : le test-and-set et le test-and-test-and-set.

— Implémenter notre sémaphore sur base du meilleur verrou entre les deux algorithmes sans la librarie standard POSIX.

— Comparer les performances de nos interfaces avec les primitives POSIX.

3.1 Tâches 2.1 et 2.3 — Implémentation des verrous (spinlocks)

3.1.1 Mise en œuvre

On a mis en place comme souhaité les verrous avec xchgl (instruction atomique) qui va pouvoir lire et swap une valeur de la mémoire.

1. **test-and-set :** L'instruction atomique 'xchgl' sera lancé par un thread en boucle pour réussir à passer le verrou de 0 à 1. L'instruction n'ai pas lourde mais le bus de mémoire aura un traffic important parce qu'à chaque fois on écrit dans la mémoire.
2. **test-and-test-and-set :** On s'est rendu compte que test-and-set devenait coûteuse avec l'écriture en mémoire, ce dont pourquoi avec cette approche on va faire une lecture avec while(*lock == 1). Du moment que le verrou est à 1 (occupé), on va boucler avec une autre boucle pour faire une lecture dans le cache sans vraiment déranger le bus de mémoire. Au moment où le verrou passe à 0 que l'instruction atomique 'xchgl' sera lancée.

3.1.2 Analyse des performances — test-and-set et test-and-test-and-set

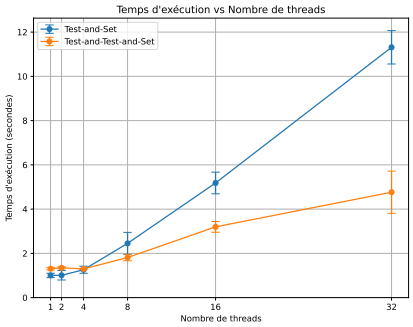


FIGURE 4 – Comparaison de performance entre les deux algorithmes

Ici sur ce graphe, on peut voir l'allure du temps d'exécution des deux algorithmes. D'un côté il y a la version la plus 'performante' en orange (test-and-test-and-set) car avec l'augmentation des threads, celui-ci reste meilleur que la première version en bleu (test-and-set). Ceci, comme mentionné plus haut, est due au fait que le bus de mémoire n'est pas dérangé à chaque instruction pour écrire dans la mémoire lorsque le verrou est pris.

3.2 Tâches 2.4 et 2.5 — Les sémaphores et adaptation de la partie 1

3.2.1 Mise en œuvre

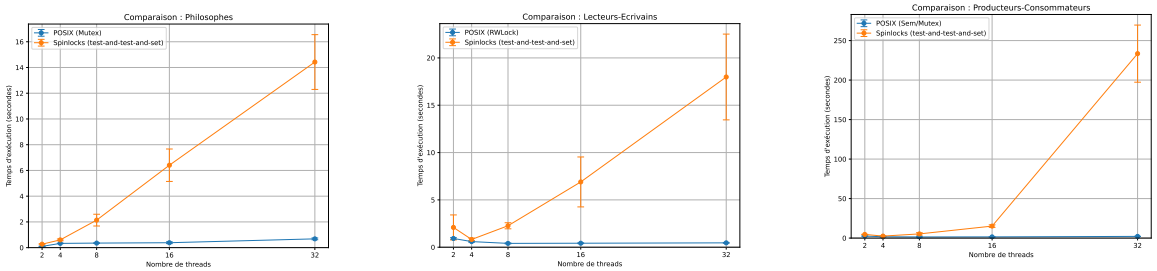
On nous a été demandé d'implémenter une interface de sémaphore, qu'on nommera sa structure 'my_sem_t' qui est composée du verrou de protection de l'algorithme le plus performant des deux (donc test-and-test-and-set) et d'un compteur.

— **my_sem_wait(my_sem_t *sem) :** Avec while(1), le thread va boucler tant qu'on a pas notre compteur x à 0. Dès que la ressource est disponible, on la prend en décrémentant le compteur et puis on déverrouille le verrou. Dans le cas où il n'y a pas de ressources, on libère le verrou pour d'autres.

— **my_sem_post(my_sem_t *sem) :** Ici on verrouille simplement le verrou, puis on incrémente le compteur x (comme on libère la ressource) et finalement on déverrouille le verrou.

Après, on a adapté les trois problèmes de la partie 1 en remplaçant les lock()/unlock() de la librairie standard POSIX par nos propres primitives d'attente active (nommés lock2() et unlock2() avec l'algorithme test-and-test-and-set).

3.2.2 Comparaison des performances : POSIX et spinlocks



(a) Philosophes (b) Lecteurs-Écrivains (c) Producteurs-Consommateurs

FIGURE 5 – Comparaison des performances : POSIX (bleu) et Spinlocks (orange)

On constate que les lignes oranges (spinlocks) ont une allure linéaire très importante après 4 threads pour le problème des philosophes et lecteurs/écrivains. Et de même à partir de 8 threads (vu que la charge est divisé par 2 entre threads) pour le problème des producteurs/consommateurs. Surtout pour 32 threads, tous les trois problèmes ont littéralement été terrible pour les spinlocks tandis que la courbe bleu (POSIX) reste très bas. Ce que font les primitives POSIX, c'est qu'ils font passer leurs threads en mode sleep pour pas consommer le CPU pour rien tandis que dans notre implémentation avec les spinlocks, les threads qui attendent un verrou ne vont pas passer en mode sleep et vont quand même tourner ce qui va consommer du CPU en abondance.

4 Conclusion

Dans ce projet, nous avons appris à évaluer et à expliquer la performance de plusieurs applications via l'utilisation des threads et des primitives de synchronisation (que nous avons implémentées par nous-mêmes dans la deuxième partie du projet). Durant le projet, on a été contraint par la limite du nombre de cœurs de notre machine virtuelle et l'augmentation du nombre de threads n'améliorait pas nos performances. Nous avons donc conclu, après les analyses de performances, que l'utilisation des primitives POSIX avec la logique d'endormissement des threads en évitant de consommer du CPU inutilement était le meilleur choix à faire. Que ce soit en terme de performance ou en terme de gestion du CPU même si l'on augmente le nombre de threads, il restait bien meilleur que nos spinlocks qu'on a implémenté.

Références

— **Documentation sur l'assembleur inline :**
`https://cristal.univ-lille.fr/~marquet/ens/ctx/doc/l-ia.html`
— **Instructions atomiques (xchg) :**
`https://stackoverflow.com/questions/14301644/atomic-inc-and-atomic-xchg-in-gcc-assembly`
— **Implémentation de Spinlocks en C :**
`https://stackoverflow.com/questions/68388918/would-this-be-a-functioning-spinlock-in-c`
— **Implémentation de Sémaphores :**
`https://stackoverflow.com/questions/54898512/implementation-of-a-semaphore-in-c`