Systèmes informatiques: Projet 1

Pierre LEBOUTTE NOMA : 43302100
Giovanni KARRA NOMA : 45032100

I. Partie 1 : Utilisation des primitives de synchronisation POSIX

A. *Problème des philosophes*

Notre algorithme implémente une solution au problème des philosophes, en évitant le problème des deadlocks

Chaque thread effectue `N_CYCLES` cycles, dans lesquels les mutex entrent en jeu. Si l'indice de la baguette à gauche (`left`) est inférieur à celui de la baguette à droite (`right`), le philosophe verrouille d'abord la baguette à gauche, puis celle à droite. Sinon, il procède dans l'ordre inverse, ce qui prévient les deadlocks puisque l'ordre avec lequel les philosophes acquièrent leur baguette suit une logique précise. Une fois cela fait, le philosophe mange puis libère les baguettes en déverrouillant les mutex.

Le problème des philosophes tel qu'implémenté ici permet de se prémunir d'un deadlock potentiel, puisque nous avons ordonné nos baguettes dans un ordre bien précis.

En effet, un thread en attente d'une baguette l'obtiendra dès qu'elle sera disponible. Pour que nous soyons bloqués, il faudrait qu'un thread cherche à acquérir une fourchette d'ordre supérieur.

Néanmoins, les attentes successives nous font monter dans l'ordre des baguettes, et cette chaîne s'arrête dès que nous avons atteint l'ordre le plus élevé. Cette baguette, étant la dernière acquise, sera finalement libérée, libérant ainsi les fourchettes d'ordres inférieurs dans la file d'attente.

Penchons nous maintenant sur l'analyse de performance de la fonction philosophes, ici pour `N_CYCLES = 1 000 000` (Fig. 1). Nos pouvons voir que le temps d'exécution du programme augmente exponentiellement avec le nombre de threads. En effet, en augmentant le nombre de threads, nous augmentons le nombre de philosophes, et donc non seulement le nombre de cycles total, mais aussi le quantité de lock et unlock des mutex, ce qui provoque des délais. Nous remarquons que l'implémentation POSIX des mutex a une bien meilleure performance que notre attente active.

Ceci s'explique par le fait que les philosophes doivent souvent attendre que les baguettes se libères, donc beaucoup de temps de processeur est alloué à des philosophes qui bouclent en attendant, ce qui retarde la libération des baguettes.

Ce problème est plus prononcé à partir de 32 threads, vu que la machine studsrv ne possède que 16 cœurs, et donc c'est encore plus probable d'avoir des threads à sections critiques en attente des threads qui bouclent pour attendre.

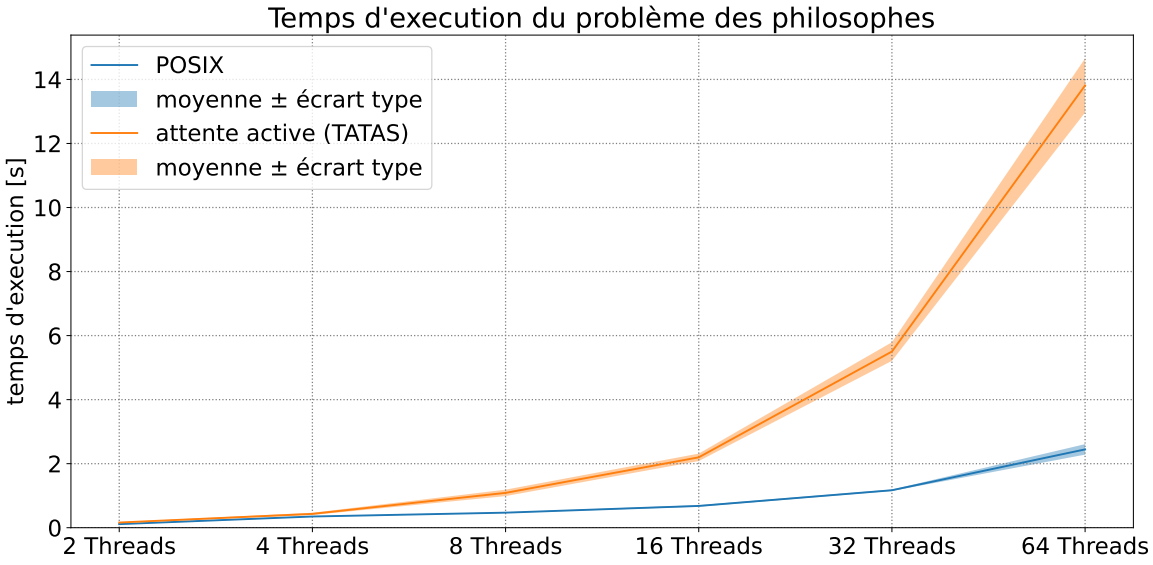

Fig. 1

B. Problème des producteurs-consommateurs

Notre second algorithme implémente le problème des producteurs-consommateurs, dans lequel les threads producteurs ajoutent des éléments à un buffer, lesquels éléments sont retirés par les threads consommateurs. L'objectif essentiel est de synchroniser l'accès au buffer pour éviter les conditions de concurrence.

Nous définissons d'abord certaines constantes (notamment la taille du buffer, 8) et intialisons deux compteurs. La fonction (`prod`) est constituée d'une boucle infinie, au sein de laquelle nous nous assurons d'abord qu'il y a de la place dans le buffer, en attendant le sémaphore `empty`. S'il n'est pas vide (`empty = 0`), le producteur peut continuer. Nous entrons dans une section critique, garantie par le `buffer_mutex`. Un nombre aléatoire est ensuite inséré dans le buffer, et la variable `prod_count` est incrémentée. Nous sortons de la section critique en libérant le mutex, puis nous libérons également le sémaphore `full`, pour permettre aux consommateurs d'accéder au buffer après avoir ajouté un élément.

La fonction `cons` a un fonctionnement similaire à la fonction `prod`. Elle boucle aussi indéfiniment mais commence par attendre le sémaphore `full`, pour voir si le buffer n'est pas vide. Si le sémaphore renvoie un nombre positif, l'exécution peut se poursuivre. Dans la section critique correspondante, un élément est retiré du buffer à l'index `(cons_count % BUFFER_SIZE)`, tout en incrémentant un compteur de consommateurs. Après être sortis de la section critique, nous incrémentons notre sémaphore `empty` pour indiquer qu'un nouvel emplacement est disponible dans le buffer.

Notre implémentation permet aux producteurs et consommateurs d'avancer comme il leur semble sans blocage entre eux, et sans connaître à l'avance le rythme et la durée de production/consommation. Nous nous prémunissons également de cas de figures problématiques (production dans un buffer plein, consommation dans un buffer vide) avec l'aide des sémaphores `empty` et `full`.

On peut voir que le temps d'exécution (Fig. 2) décroit exponetiellement, que ce soit pour notre implémentation des mutex et sémaphores ou celles de POSIX. C'est normal, vu qu'en augmentant le nombre de threads, nous départageons mieux le travaille entre eux.

Une observation que nous pouvons faire est la divérgence du graphe de l'attente active par rapport à POSIX à partir de threads = 32. Ceci est attendu pour les mêmes raisons que le problème des philosophes: quand nous avons plus de threads que de cœurs, l'attente active retarde l'execution de la section critique.

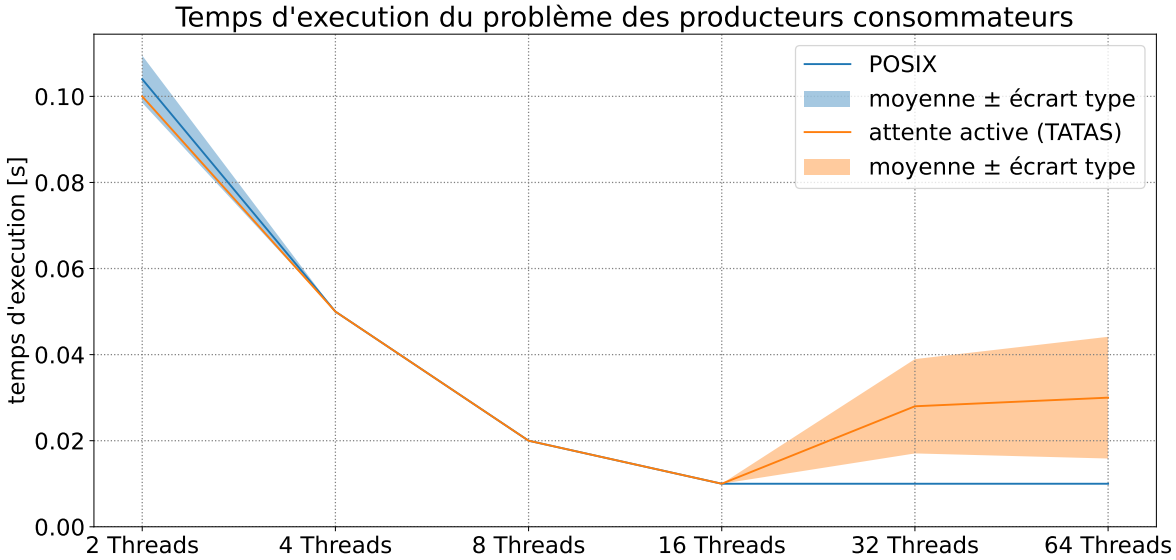

Fig. 2

C. Problème des lecteurs-écrivains

Notre dernier algorithme implémente le problème des lecteurs-écrivains. Nous commençons par définir le nombre de lecteurs et d'écrivains, puis nous initialisons des compteurs pour suivre le nombre d'écrivains/lecteurs totaux ou actifs, ainsi que deux sémaphores, `wsem` et `rsem`, l'un pour bloquer l'accès aux écrivains, l'autre aux lecteurs. Nous avons également besoin de trois mutex, `mutex_readcount`, `mutex_writecount`, qui assurent l'accès exclusif aux variables `readcount`, `writecount`, et `z`, qui garantit qu'un seul lecteur peut attendre à la fois.

Intéressons-nous à la fonction `writer`. Elle est exécutée par chaque thread écrivain. Nous bloquons d'abord le `mutex_writecount` pour incrémenter `writecount` correctement. De plus, si le thread est le premier écrivain, il bloque et attend sur le sémaphore

`rsem`. Une fois libéré, la variable `mutex_writecount` est libérée, et le thread attend le sémaphore `wsem` pour pouvoir effectuer son action de manière exclusive. Ce sémaphore est ensuite libéré pour laisser la place à d'autres écrivains. La variable `writecount` est ensuite décrémentée au sein d'une section critique. Si l'écrivain était le dernier, il libère l'accès en relâchant `rsem`.

Attardons-nous ensuite sur la fonction `reader`. Elle permet à plusieurs lecteurs d'accéder simultanément à un document. À l'image de `writer`, la fonction entre dans une boucle tant que le nombre total de lecteurs n'est pas atteint. Nous nous assurons qu'il n'y a qu'un seul lecteur en attente avec le mutex `z`. `rsem` bloque ensuite les lecteurs lorsqu'il y a une écriture en cours. L'incrémentation de `readcount` est réalisée dans la section critique couverte par `mutex_readcount`. Similairement à la fonction `writer`, si le thread est le premier lecteur, il bloque l'accès aux écrivains en attendant `wsem`. Après simulation de la lecture, nous décrémentons dans une section critique `readcount`, qui illustre le nombre temporaire de lecteurs. Si `readcount == 0` (c'était le dernier lecteur), le lecteur libère les écrivains en relâchant `wsem`.

Cette implémentation remplit deux objectifs. D'une part, nous nous assurons qu'il n'y a qu'un et un seul writer qui accède à la base de donnée, seul (sans autre writer/reader). Les lecteurs peuvent eux être multiples. De plus, notre implémentation donne aussi la priorité aux écrivains. En effet, lorsqu'un écrivain est en attente, nous empêchons de nouveaux lecteurs d'accéder à la base de donnée, tout en permettant aux lecteur actifs de terminer leur travail.

Pour le problème des lecteurs-écrivains, on voit également que le temps d'exécution diminue avec le nombre de threads (Fig. 3). Les raisons de divérgence de l'attente active restent les mêmes que pour le problème précédent.

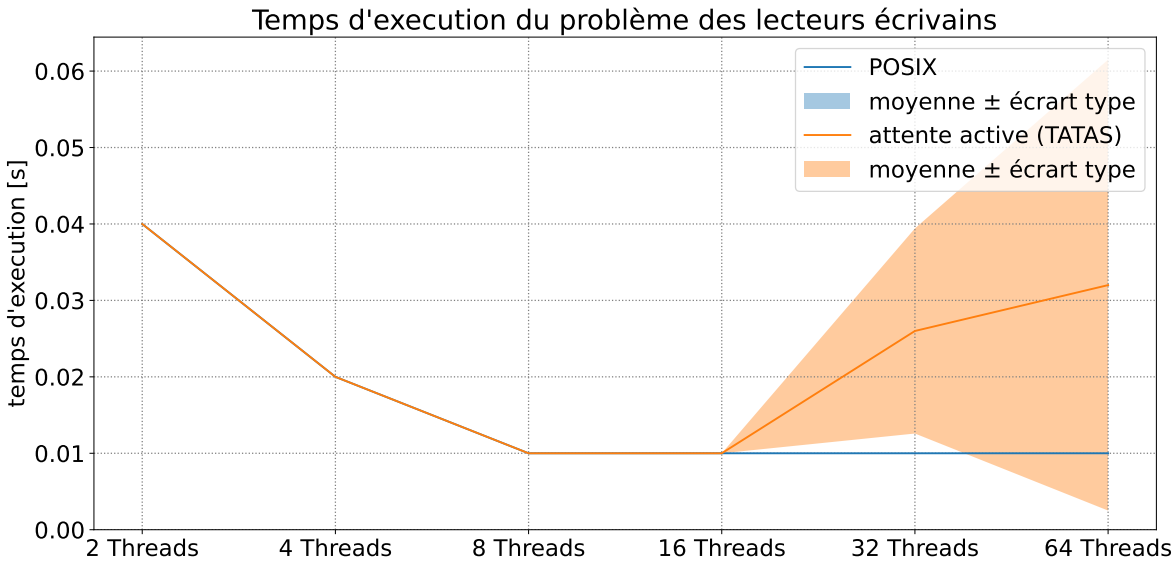

Fig. 3

II. Partie 2 : Mise en œuvre des primitives de synchronisation par attente active

A. *Instruction atomique : `xchg`*

Afin de mettre en œuvre les verrous par attente active, nous avons utilisé l'instruction atomique `xchg`. Le principe est simple : lorsque nous voulons verrouiller un mutex, nous allons échanger sa valeur avec une variable contenant 1. Si après l'échange notre variable contient 0, alors nous obtenons le mutex et celui-ci est désormais verrouillé. Si elle contient 1, alors le mutex est déjà verrouillé, et nous devons attendre qu'il se débloque.

Il existe différentes implémentations de l'attente active. Nous détaillerons ci-dessous les implémentations `test-and-set`, `test-and-test-and-set`, et `backoff-test-and-test-and-set`.

B. *`test-and-set`*

Lorsqu'on appelle `lock` sur un de nos mutex, nous effectuons une boucle `while` qui échange 1 avec la valeur du mutex jusqu'au moment où nous obtenons 0, comme expliqué ci-dessus, dans quel cas nous pouvons quitter la boucle. L'inconvénient principal de cette méthode est que boucler la commande `xchg` a un effet néfaste sur la performance à cause du fonctionnement du partage d'une même zone de mémoire par plusieurs cœurs comme détaillé dans le syllabus.

C. `test-and-test-and-set`

Cette implémentation est très similaire à la précédente, sauf qu'au lieu d'effectuer `xchg` en boucle, nous allons attendre que la valeur du mutex ne soit plus 1 avant de refaire l'échange pour tenter d'obtenir le mutex, ce qui diminue massivement le nombre de modifications apportés à la zone mémoire du mutex. L'inconvénient majeur est que lorsque le mutex est déverouillé, tous les threads qui étaient bloqués vont tenter d'obtenir le mutex en même temps, ce qui génère des pics de ralentissement.

Dans notre code, nous activons cette implémentation en ajoutant le paramètre `-D METH2` lors de la compilation.

D. `backoff-test-and-test-and-set`

Dans cette implémentation, chaque thread va attendre un temps aléatoire (qui augmente à chaque échec) avant d'essayer d'obtenir le mutex, ainsi répartissant équitablement dans le temps les tentatives d'accès au mutex.

Nous l'activons dans notre code en ajoutant le paramètre `-D BACKOFF` lors de la compilation.

E. Sémaphores

L'implémentation de sémaphores en utilisant l'attente active n'est pas bien complexe. Chaque structure sémaphore contient sa valeur et un mutex, et lorsque nous appelons `wait`, notre thread va boucler jusqu'au moment où la valuer du sémaphore est positive, dans quel cas il va tenter d'obtenir l'accès, et s'il l'obtient, il décrémente la valeur, si pas il va retourner dans la boucle. Quand nous appelons `post`, le thread va simplement incrémenter la valeur du sémaphore. Le mutex sert à protéger la valeur du sémaphore lors des accès à celle-ci.

F. Comparaison des performances

Dans ce graphe (Fig. 4), nous comparons le temps d'execution des différentes implémentations de l'attente active sur un programme où chaque thread effectue `6400/N` sections critiques.

Nous remarquons que le `test-and-test-and-set` a une meilleure performance que le test-and-set, comme attendu. Ce qui est surprenant est le `backoff-test-and-test-and-test` qui arrive à avoir un temps d'execution plus ou moins constant malgré le fait que la machine n'aie que 16 cœurs. Les valeurs utilisés ici sont un temps d'attente minimum `MIN_WAIT = 10` itérations d'une boucle for, et le temps d'attente maximum `MAX_WAIT = 1000`.

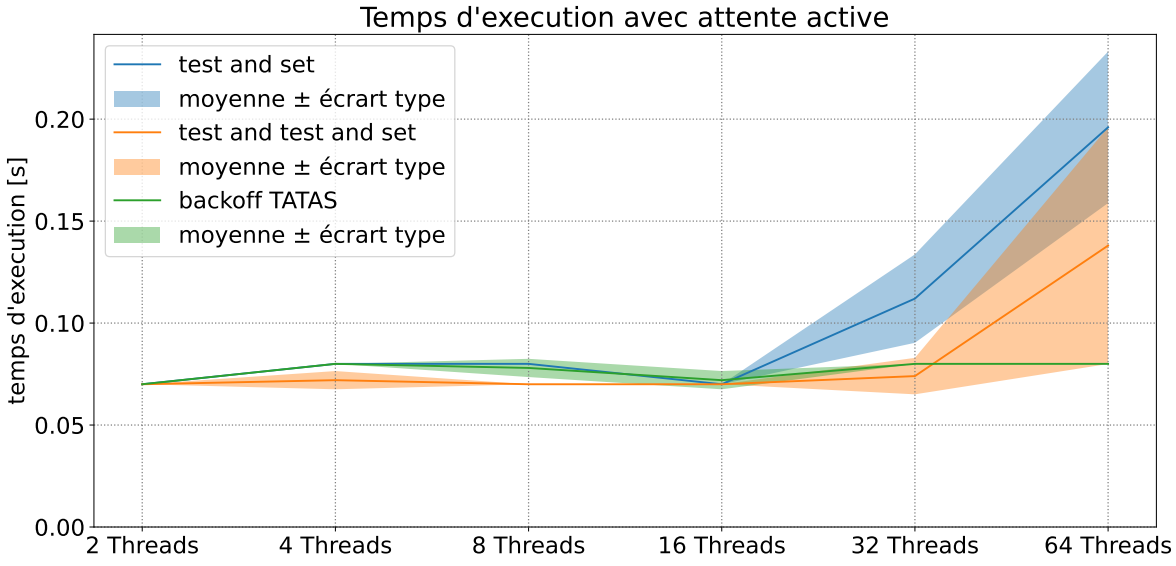

Fig. 4

III. Conclusion

Ce projet nous a appris beaucoup de choses sur l'utilisation les primitives de synchronisation, ainsi que leur implémentation. Il met bien en évidence qu'il y a très rarement une solution universelle en informatique, et qu'il y a très souvent des avantages et inconvénients à tout, la meilleure méthode dépend au final de ce que nous voulons réaliser et du matériel à notre disposition.