Set and set and set

On appelle souvent cette solution le ***test-and-test-and-set***, par opposition à la version de l'algorithme précédent qui est appelé test-and-set. L'idée est de ne tenter l'opération atomique ***xchgl***, qui teste et assigne (set) la valeur de façon atomique, que lorsque le verrou semble libre. Cette situation apparait lorsque le thread qui effectuait sa section critique écrit 0 dans le verrou : l'entrée correspondant à ***cette*** adresse passe en état M dans son cache après invalidation dans le cache des autres processeurs, qui lisent alors 0 grâce à une requête ***sur le bus***. Le pseudocode de cette opération peut être :

Le test-and-test-and-set limite fortement la contention sur le but pendant l'exécution de la section critique du thread possédant le verrou. Elle n'est toutefois pas parfaite quand il y a un grand nombre de threads qui souhaitent accéder à leur section critique de façon répétée : à chaque fois qu'un de ces threads quitte sa section critique, l'invalidation de l'adresse du verrou dans les caches de tous les autres processeurs entraine une rafale de lecture de la nouvelle valeur, suivi d'une rafale d'opérations xchgl. De toutes ces opérations, une seule sera couronnée de succès, mais toutes devront bloquer le bus et limiter le progrès général du système. Le caractère synchrone et répété de ces pics d'occupation du bus peut réduire la performance générale du système, sans toutefois le faire ***autant que le test-and-set seul***.

Les instructions atomiques comme xchgl impliquant une adresse mémoire doivent non seulement obtenir l'accès en mode M à ce mot mémoire dans leur cache, mais aussi bloquer l'accès au bus par les autres processeurs pour empêcher son invalidation par un autre processeur pendant l'exécution de l'instruction atomique. Ce blocage du bus a un impact important sur la performance de l'ensemble des threads en cours d'exécution : ***les accès mémoire sont bloqués pour tous les processeurs*** en attendant que l'opération atomique sur un de ces processeurs soit terminée.

Le coût du blocage du bus explique la performance décevante du code présenté précédemment, où chaque thread souhaitant accéder à sa section critique utilise une succession d'opérations xchgl continue. La situation créée est illustrée par la figure ci-dessous.

--

Comparaison les 2 SET

L’idée du set-and-set est la mise en œuvre d’une attente active grâce à l'instruction atomique xchgl. Cette méthode permet ainsi d’éviter les changements de contexte état Blocked- Ready – Running. Notre thread ne devra donc plus attendre que le scheduler lui assigne un processeur lorsque celui-ci se trouvera dans l’ état Ready,

L’idée principal du test-and-set-and-set est de tenter l’opération atomique ***xchgl***, que **lorsque le verrou semble libre**. Ainsi, celui-ci limite la contention en réduisant drastiquement le nombre de tentative de l’opération atomique.

***Bref rappel***

***Philos***

***SCREEN PHILO***

« Chaque philosophe dispose d'une place qui lui est réservée sur cette table. La table comprend autant de baguettes que de chaises et une baguette est placée entre chaque paire de chaises. Chaque philosophe est modélisé sous la forme d'un thread qui effectue deux types d'actions : penser et manger. Pour pouvoir manger, un philosophe doit obtenir la baguette qui se trouve à sa gauche et la baguette qui se trouve à sa droite. Lorsqu'il a fini de manger, il peut retourner à son activité philosophale. »

Plus nous avons de philosophes/threads et plus le temps d’exécution augmente. En effet, puisque le problème impose que chaque thread accède à la section critique seulement si celui-ci possède la baguette qui se trouve à sa gauche et la baguette qui se trouve à sa droite, c’est-à-dire 2 mutex. Ces 2 mutex sont partagés avec les threads voisins du tableau. De ce fait, lorsqu’un threads obtient sa section critique celui-ci empêche ces 2 voisins de démarrer. Ce qui a pour résultat de nous donner une évolution plus ou moins linéaire, le temps d’attente va augmenter proportionnellement avec le nombre threads

***Producteur consommateur***

***SCREEN PRODUCTEUR CONSOMMATEUR***

Le problème des producteurs-consommateurs est un problème divisé en deux types de threads :

* les ***producteurs*** : Ce sont des threads qui produisent des données et placent le résultat de leurs calculs dans une zone mémoire accessible aux consommateurs.
* les ***consommateurs*** : Ce sont des threads qui utilisent les valeurs calculées par les producteurs.

« Ce qu’il faut savoir :

* un producteur peut arrêter de produire à n'importe quel moment
* les producteurs doivent pouvoir ajouter de l'information dans le buffer partagé tant qu'il y a au moins une place de libre dans le buffer ;
* Un producteur ne doit être bloqué que si tout le buffer est rempli ;
* les consommateurs doivent être bloqués uniquement si le buffer est entièrement vide.
* Dès qu'une donnée est ajoutée dans le buffer, un consommateur doit être réveillé pour traiter cette donnée. »

Ces deux types de threads communiquent en utilisant un buffer qui a une capacité limitée à 8 places.

Le problème suivant impose l’utilisation de mutex et de sémaphores. La sémaphore permet de s’assurer de pas écrire sur un emplacement du buffer qui n’aurait pas été vidé et de ne pas retirer . L’accès buffer est protégé par un mutex ce qui signifie que l’accès à celui-ci ce fait thread par thread, cela a pour conséquence de réduire l’efficacité d’un grand nombre de threads. En effet, nous pouvons constater que la courbe fini par s’aplatir à partir d’un certain nombre de threads, c’est l’effet du mutex qui bride la possibilité de travailler en parallèle.

***Lecteur écrivain***

***SCREEN lecteur écrivain***

En conclusion « l’avantage des mutex et des sémaphores POSIX, est que les threads ne perdent pas de temps et de ressources processeur à effectuer leur attente active, que ce soit dans un système mono ou multi-processeur. »

Toutefois, les mutex et sémaphores POSIX mis en œuvre avec le concours du noyau ont un désavantage important qui est **la latence de leurs opérations**. Cette latence est due à plusieurs facteurs :

* la combinaison entre le surcoût de la mise en attente (état Blocked) d'un thread appelant lock() sur un mutex, dans une file gérée par le noyau ;
* le temps écoulé entre le moment où le thread a été mis en état *Ready* (par un appel à unlock()) et celui où il sera de nouveau placé par le *scheduler* sur un processeur. En effet, c'est le *scheduler* qui décidera de placer plus tard ce thread sur un processeur, ce n’est pas parce que le thread est en état *Ready* qu’il obtient immédiatement un processeur ;
* le coût des différents changements de contexte au cours de cette procédure. Le temps total pour ces opérations peut être plus long que la durée de la section critique du thread qui détenait le mutex.

De par l’énumération de ces différents facteurs, nous comprenons qu’il est plus pertinent ***dans certains cas*** d’effectuer une attente active courte en attendant la fin de la section critique. Pour ce faire nous allons utiliser le concept du set-and-set.

L’idée du set-and-set est la mise en œuvre d’une attente active grâce à l'instruction atomique ***xchgl***. Cette méthode permet ainsi d’éviter les changements de contexte (état *Blocked* et *Ready*). Ainsi, notre thread ne devra donc plus attendre que le *scheduler* lui assigne un processeur lorsque celui-ci se trouvera dans l’ état *Ready*.

L’idée principal du test-and-set-and-set est de tenter l’opération atomique ***xchgl***, que **lorsque le verrou semble libre**. Ainsi, celui-ci limite la contention en réduisant drastiquement le nombre de tentative de l’opération atomique ***xchgl***.

***Comparaison des philos***

Nous pouvons constater que les attentes actives : set-and-set et set-and-set-and-set ont de très bon résultat en termes de rapidité contrairement au mutex POSIX. Par conséquent, nous pouvons conclure qu'il est plus pertinent d'utiliser la méthode d'attente active lorsqu’un thread **passe peu de temps** dans la section critique. Cela vient corroborer la conclusion de partie 1.

***Comparaison des producteurs consommateurs***

Nous pouvons constater que les attentes actives : set-and-set et set-and-set-and-set ont de très mauvais résultat en termes de rapidité lorsqu’un plus grand nombre de threads souhaitent accéder à la section critique du buffer. En effet, à chaque fois qu’un thread quitte la section critique, entraine une salve de lecture de la nouvelle valeur, accompagné d’une salve d’opérations xchgl. « Le caractère synchrone et répété de ces pics d'occupation du bus peut réduire la performance générale du système, sans toutefois le faire autant que le test-and-set seul. »

En conclusion, les attentes actives utilisant l’opération atomique xchgl n’ont d’intérêts que dans certains cas bien précis : lorsque l’accès en section critique est bref et que cette section critique ne soit pas la seul partagées entre tous les threads existant. En effet, le problème des philosophes et le problème des consommateur, producteur viennent étayer notre conclusion.