DUQUENNOY Antoine
GONTARD Benjamin
Groupe de TD numéro 3
Groupe de rendu Q

COMPTE RENDU TP NUMERO 4 : Lecteurs-rédacteurs

Compilation des programmes :

-make
-make all

Lancement des programmes :

Tout les programmes s'exécutent de la même façon : ./NOM_PROGRAMME suivi du nombre de lecteurs souhaité, puis du nombre de rédacteurs souhaité et enfin, le nombre d'itérations (lecture/écriture) que chaque thread devra effectuer.

./redacteur_prio <NB_LECTEURS> <NB_REDACTEURS> <NB_ITERATIONS>
./lecteur_prio <NB_LECTEURS> <NB_REDACTEURS> <NB_ITERATIONS>
./file_prio <NB_LECTEURS> <NB_REDACTEURS> <NB_ITERATIONS>
./file_prio2 <NB_LECTEURS> <NB_REDACTEURS> <NB_ITERATIONS>
./file_prio3 <NB_LECTEURS> <NB_REDACTEURS> <NB_ITERATIONS>

Nous vous recommandons de tester avec 5 lecteurs, 5 écrivains et 5 itérations afin de vous convaincre que les différentes politiques que nous avons implémentées respectent bien leur spécifications. Pour les lecteurs ou rédacteurs prioritaires, vous pourrez essayer de tester avec un grand nombre de threads afin de constater que tout fonctionne bien.
Nous n'avons pas trouvé le moyen de réaliser des jeux de tests efficaces : tout repose sur l'ordonnancement des threads et par conséquent, il est impossible de savoir quel thread prendra la main à quel moment.
Ainsi, pour vous convaincre du bon fonctionnement du nos programmes, nous vous invitons à les essayer plusieurs fois de suite.

Tests :

Chaque test lance le programme avec 5 lecteurs, 5 rédacteurs et 5 itérations pour chaque thread. Si vous souhaitez lancer plus de lecteurs, plus de rédacteurs ou effectuer plus d'itérations, nous vous invitons à lancer le programme de la manière vue dans lancement des programmes.

make test_red pour tester la version rédacteur prioritaire
make test_lect pour tester la version lecteur prioritaire
make test_file_1 pour tester la version avec la file
make test_file_2 pour tester la deuxième version avec la file
make test_file_3 pour tester la troisième version avec la file

Introduction :

Au cours de ce TP, nous avons été amenés à implémenter plusieurs solutions afin de résoudre le problème des Lecteurs/Rédacteurs. Chaque version a une spécificité particulière que nous allons détailler par la suite.

Implémentation :

Pour chaque version, nous détaillerons les éléments de synchronisation utilisés, puis le comportement général du début de lecture/écriture et de la fin de lecture/écriture.
Les fonctions importantes du code seront commentées.

-Version 1 : Priorité aux rédacteurs.

Fichiers utilisés : redacteur_prio.c et lecteur_redacteur.h

Pour donner la priorité aux rédacteurs, notre démarche est la suivante :
-Nous avons deux variables conditions qui vont chacune représenter une « file d'attente » : une pour les lecteurs (fileL) et une autre pour les rédacteurs (fileR).
-Nous avons des entiers permettant de savoir si des lecteurs sont en train de lire, ou si des rédacteurs sont en train d'attendre de pouvoir écrire, de plus, nous avons aussi un entier permettant de savoir si un rédacteur est en train d'écrire ou non (afin de n'autoriser qu'un rédacteur à écrire à la fois).
-Finalement, pour protéger notre section critique, nous avons un mutex nommé global.

Chaque début ou fin de lecture écriture commence par l'acquisition du mutex global.

Lorsque les lecteurs commencent leur lecture, ils regardent tout d'abord si il n'y a pas de rédacteurs en attente, auquel cas ils s'endorment dans leur variable condition, jusqu'à ce que le dernier rédacteur vienne les réveiller. Une fois qu'un lecteur est sorti de cette boucle d'attente, il incrémente le nombre de lecteur et part faire sa lecture.

Lorsqu'un lecteur fini sa lecture, il décrémente le nombre de lecteur et si il était le dernier lecteur en cours, alors il envoi un signal sur la file d'attente des rédacteurs afin de leur indiquer qu'un rédacteur peut prendre la main.

Lorsque les rédacteurs commencent leur écriture, ils commencent par incrémenter le nombre de rédacteurs en attente, afin que les lecteurs voient qu'un rédacteur est arrivé et qu'il faut par conséquent le prioriser. Le rédacteur se met en attente tant qu'il y a un rédacteur qui est en train d'écrire ou qu'il reste des lecteurs qui doivent finir leur lecture.
Une fois sorti de cette boucle d'attente, le rédacteur indique qu'il est en train d'écrire en positionnant l'entier à 1.

Lorsqu'un rédacteur termine son écriture, il décrémente le nombre de rédacteur en attente de 1 et indique qu'il a fini son écriture en positionnant l'entier à 0.
Enfin, si il reste des rédacteurs en attente, il envoi un signal sur la file des rédacteurs, mais si il n'y a plus de rédacteurs en attente, alors un signal de diffusion (broadcast) est envoyé sur la file des lecteurs afin qu'ils puissent réaliser leur lectures en parallèle.

-Version 2 : Priorité aux lecteurs.

Fichiers utilisés : lecteur_prio.c et lecteur_redacteur2.h

Pour donner la priorité aux lecteurs, notre démarche est la suivante :

Un sémaphore qui va représenter l'accès à la lecture/écriture, un entier pour connaître le nombre de lecteurs encore en train de lire et un mutex pour protéger l'accès à cette variable.
Nous avons aussi un mutex pour les redacteurs, ce mutex nous garantira qu'un seul et unique rédacteur pourra venir réaliser l'opération wait sur le sémaphore en début de lecture : lorsqu'un rédacteur finira sa lecture, il va d'abord réaliser l'opération post sur le sémaphore avant de débloquer le mutex, si des lecteurs sont en attente, alors ce seront eux qui prendront la main, mais si il n'y avait pas de lecteurs en attente, alors le rédacteur suivant, lorsqu'il pourra prendre le verrou sur le mutex des rédacteurs, réalisera l'opération wait sur le sémaphore et constatera qu'il y a un jeton et pourra donc passer. Dès que le message ATTENTE LECTURE sera affiché sur le terminal, alors le rédacteur terminera sa rédaction et passera la main aux lecteurs.

Lorsqu'un lecteur commence sa lecture, si il est le tout premier lecteur, ou le premier lecteur après une rédaction terminée, il va essayer de prendre l'accès au sémaphore : seul le premier lecteur a besoin de prendre l'accès au sémaphore, ainsi tout les autres lecteurs pourront directement réaliser leur lecture. On incrémente ensuite le nombre de lecteurs. (les autres lecteurs ne font pas de wait sur le sémaphore mais vont directement effectuer leur lecture)

Lorsqu'un lecteur termine sa lecture, il décrémente le nombre de lecteurs : si c'était le dernier lecteur, alors il réalise l'opération post sur le sémaphore afin de donner un jeton pour les prochains threads qui se seraient bloqués sur le sémaphore (donc un des rédacteurs pourra passer).

Lorsqu'un rédacteur commence son écriture, il commence par prendre le lock sur le mutex des rédacteurs, puis il essaie de prendre un jeton dans le sémaphore : il ne pourra donc passer qu'après que le dernier lecteur ait réalisé l'opération post, ou alors quand un autre rédacteur aura terminé d'écrire.

Lorsqu'un rédacteur termine son écriture, il réalise l'opération post sur le sémaphore afin de passer la main aux lecteurs. Ensuite, il libère le verrou posé sur le mutex des rédacteurs.

Ainsi, dès l'instant où un lecteur prendra l'accès au sémaphore, tout les lecteurs pourront réaliser leur lecture sans que les rédacteurs ne viennent les interrompre. Seul le dernier lecteur pourra mettre un jeton dans le sémaphore, afin de donner la main aux rédacteurs.

-Version 3 : Priorité selon l'ordre partiel d'arrivée.

Fichiers utilisés : file_prio.c et thread_safe_list.c/.h

Pour les éléments de synchronisation, nous reprenons la structure vue à la version 1, en ajoutant une file.

Pour cette version, nous avons utilisé une file afin de prioriser les threads selon leur ordre d'arrivée, tout en gardant la contrainte que plusieurs lecteurs peuvent lire en même temps.
Ainsi, à chaque début de lecture ou écriture, nous commençons par insérer le thread (son id et son type) dans la file.

Lorsqu'un lecteur commence sa lecture, il se bloquera dans la variable condition des lecteurs si il y a un rédacteur en train d'écrire ou si il n'est pas la tête de file.
Après être sorti de la boucle d'attente, il envoi un broadcast sur la file des lecteurs afin de réveiller les lecteurs endormis et permettre à un autre lecteur d'effectuer simultanément sa lecture si il est la tête de file. Si un rédacteur venait à être en tête de file, ce broadcast n'aurait aucun effet. Ainsi, les les lecteurs se réveillent simultanément.

Lorsqu'un lecteur termine sa lecture, il regarde si il est le dernier lecteur et si la tête de file est un rédacteur : auquel cas il enverra un broadcast sur la file des rédacteurs afin d'être sur de réveiller le rédacteur en tête de file.

Lorsqu'un rédacteur commence son écriture, il s'endort si un autre rédacteur est en train d'écrire ou si il reste des lecteurs en cours de lecture ou si il n'est pas la tête de file.
Quand il sortira de cette boucle d'attente, il positionnera à 1 l'entier qui indique qu'une rédaction est en cours.

Lorsqu'un rédacteur termine sa lecture, il positionnera à 0 l'entier qui indique qu'une rédaction est en cours et, si la tête de la liste est un rédacteur alors il enverra un signal de diffusion sur la variable condition des rédacteurs, sinon un broadcast sur les lecteurs.

-Version 4 : Optimisation de la version3.

Fichiers utilisés : file_prio2.c et thread_safe_list_version2.c/.h

La version 4 est presque identique à la version 3, cependant une différence majeure est à noter :
-Lorsque les lecteurs examinent si ils sont sur la tête de la file, en réalité ils seront considérés comme tête de file si ils sont avant le premier rédacteur dans la file.
Ainsi, nous n'avons plus besoin de broadcast les autres lecteurs après être sorti de la boucle d'attente.
Plusieurs lecteurs pourront sortir à la fois quand un rédacteur fera un broadcast sur les lecteurs.

-Version 5 : Priorité selon l'ordre partiel d'arrivée, autre implémentation.

Fichiers utilisés : file_prio3.c et thread_safe_list_version3.c/.h

Pour réaliser cette version, en plus d'une file, nous utilisons un tableau statique de variable condition de taille nb_lecteurs + nb_redacteurs : chaque thread aura un id permettant de savoir quelle case de notre tableau de variable condition lui sera associée. Ainsi, il nous sera possible de cibler le thread à réveiller (il sera en tête de notre file et nous pourrons récupérer son id, qui correspondra à la case du tableau).
Nous avons toujours notre mutex global et les variables permettant de savoir le nombre de lecteurs/rédacteurs en cours de lecture/rédaction ainsi que la variable permettant de savoir si un rédacteur est en train d'écrire.

Lorsqu'un lecteur commence sa lecture, il s'endormira dans la variable condition qui lui est associée si il y a des rédacteurs en train d'écrire ou si il n'est pas la tête de la file.
Une fois sorti de la boucle d'attente, il enverra un signal à la nouvelle tête de file, afin de permettre possiblement à un autre lecteur de passer en même temps (comme dans la version3).

Lorsqu’un lecteur termine son écriture, si il est le dernier lecteur et que la tête de file est un rédacteur, il enverra un signal à la tête de file afin de le réveiller.

Le code des rédacteurs est très similaire à celui des versions précédentes, à la différence qu’il s’endormira sur la variable condition qui lui est associée si ce n’est pas à son tour de prendre la main et qu’en fin de rédaction il enverra un signal afin de réveiller le thread en tête de file.

Avec cette version, nous n’avons plus la semi attente active que nous avions avec les versions 3 et 4 : nous pouvons cibler le thread à réveiller et ainsi réveiller uniquement celui en tête de file, alors que dans les versions précédentes, nous devions réveiller tout les threads afin que seul celui qui était en tête de liste puisse prendre la main.