ASR2 : Librairie de thread

Aloïs BRUNEL et Vincent DELAITRE

25 avril 2008

1 Options de la librairie

Nous avons implémenté une librairie de threads qui fonctionne en coopératif ou en préemptif, avec un ordonnanceur FIFO ou FIFO avec priorités. Voiçi les variables permettant de régler ces différents paramètres (situés au début de mythread.c) :

- FONCTIONNEMENT : 0 pour une gestion coopérative des threads - 1 pour une gestion préemptive.
- VIRTUAL_TIMER : Le timer utilisé pour le fonctionnement préemptif est le même que celui de la fonction **sleep**, celle-ci est donc inutilisable lorsque VIRTUAL_TIMER vaut 0. Il est possible d'éviter ce conflit en réglant VIRTUAL_TIMER sur 1, mais alors le temps de pause du lors de l'utilisation de **sleep** se déroulera sans préemption de la librairie.
- PREEMPT_INTERVAL : Intervalle de préemption en microsecondes
- ORDONNANCEMENT : 0 pour un ordonnancement FIFO, 1 pour un ordonnancement personnalisé.

2 Ordonnanceur coopératif

2.1 Ordonnanceur simple

- Type des threads : nous avons défini le type de base des threads : **mythread_t**. Il contient les champs suivants :
  - context : contient le contexte d'exécution du thread.
  - state : état du thread : prêt, en attente sur un join, en attente sur une condition, terminé.
  - depending_threads : (utile pour l'ordonnanceur perso) indique si d'autres threads dépendent de ce thread : aucun n'en dépend, certains threads attendent le thread courant pour continuer un join, certains threads attendent pour entrer dans un mutex dont le thread courant est propriétaire.
  - join : thread qui attend le thread courant sur un join.
  - value_ptr : adresse où stocker la valeur de retour du thread courant.
  - attr : attribut du thread.

  Les attributs du thread sont représentés par le type **mythread_attr_t** :
  - priority : priorité du thread.
- Variables globales :

  - current_thread : de type mythread_t
  - scheduler : ordonnanceur que l'on identifie pour l'instant à une variable global ready_list de type mythread_t_list
- Fonctions de gestion des threads (voir manuel) :
  - **mythread_create** : initialise le thread et le rajoute à ready_list.
  - **mythread_self** : renvoit le thread courant grâce à current_thread.
  - **mythread_swap** : (fonction auxiliaire, ne pas utiliser) prend le premier thread de ready_list, le stocke dans current_thread et l'exécute en appelant **mythread_handler**.
  - **mythread_handler** : (fonction auxiliaire, ne pas utiliser) lance la fonction principale du thread passée en argument puis appelle **mythread_exit** avec la valeur de retour du thread.
  - **mythread_join** : place le thread courant en attente, met son état sur join_wait et le retire de la liste des threads prêts. Indique aussi au thread passé en argument qu'il est attendu.
  - **mythread_yield** : place le thread courant dans la liste des threads prêts et appelle **mythread_swap**.
  - **mythread_exit** : stocke la valeur de retour du thread qui se termine. S'il est attendu par un thread (sur un join) : place ce thread dans ready_list et met son état sur ready.

3 Interblocages

Il existe des situations d'interblocage. Par exemple si un premier thread pose un verrou sur le mutex A, puis demande le verrou sur le mutex B et que dans le même temps un thread pose un verrou sur le mutex B, puis demande le verrou sur le mutex A. Il y a effectivement interblocages puisqu'aucun des accès aux ressources ne sera disponible. Tous les threads sont ainsi bloqués.

Il semble difficile de détecter ces inter-blocages. Un raccourci douteux qui est une sorte d'explication qualitative est que l'abscence d'interblocages correspond à peu près à la terminaison du programme multithreadé. Or il est impossible de s'assurer de la terminaison d'un programme quelconque.

4 Préemption

La bibliothèque possède deux modes : un mode coopératif décrit plus haut et un mode préemptif. Quelles sont les différences ? Le fonctionnement en mode coopératif signifie que ce seuls les threads décident du moment où ils rendent la main à l'ordonnanceur de threads. Ainsi c'est au programmeur de s'assurer qu'un thread ne va pas bloquer les autres. Au contraire, en fonctionnement préemptif, un thread peut être *préempté* c'est-à-dire interrompu, sans qu'il l'ait décidé lui-même, c'est la librairie de threads qui décide du moment où chaque thread rend la main. Evidemment, chaque thread a toujours la possibilité de rendre la main de lui-même. Dans notre implémentation, un timer est mis en place et à intervalle régulier, les threads sont préemptés.

Notons qu'une erreur aurait été possible : si l'on est encore en train d'exécuter une fonction de la librairie (typiquement mythread_mutex_lock), il ne faut pas que le timer interrompe la

tâche en cours sous peine de déstabiliser complètement la gestion des threads. Ainsi nous avons placé un verrou global qui est activé lors de l'entrée dans ces fonctions. Quand ce verrou est détecté et que le timer décide de changer de thread, l'opération d'ordonnancement est retardée jusqu'à la sortie de la fonction critique (mythread_mutex_lock dans notre exemple).

L'implémentation de ce timer a été réalisée avec **setitimer** et **sigaction**. Par défaut le timer est fixé à 100 micro secondes, plus ou moins 10 microsecondes de délai de la part du système.

Quels sont les avantages d'un tel mode de fonctionnement par rapport aux threads coopératifs ?

- Tout d'abord, un tel système permet de distribuer de manière équitable le temps de calcul entre les différents threads. Comme nous le verrons dans la partie suivante, cela permet également de répartir le temps de calcul différement en fonction de *priorités* différentes accordées aux threads.
- Cela diminue également le temps de traitement maximum d'un thread.
- Quand un thread devient prioritaire sur le thread courant, on peut lui donner la main

Il existe néanmoins plusieurs désavantages :

- Cette technique déteriore le temps *moyen* de traitement. En effet, on change beaucoup plus souvent de thread courant, ce qui entraîne des frais de gestion plus lourds qu'auparavant, on passe plus de temps dans les appels systèmes, on a donc perdu du temps utilisateur. Les threads, pour s'exécuter, demanderont en moyenne plus de temps.

Finalement, le mode préemptif est donc un compromis entre efficacité en temps moyen d'exécution et équité du partage du temps d'exécution.

5 Ordonnancement

Deux politiques d'ordonnancement ont été implémentées. La première est une politique FIFO, et l'autre est une politique FIFO couplée à un système de priorité.

5.1 FIFO

Ici les threads prêts à être exécutés sont simplement mis dans une liste **ready**. A chaque création de thread, celui qui vient d'être créé est mis à la fin de cette liste. L'allocation est donc effectuée dans l'ordre d'arrivée. Le problème de cette politique est qu'elle défavorise largement les threads n'ayant besoin que d'un court laps de temps pour s'exécuter. Le temps d'attente n'est pas du tout corrélé au temps d'utilisation, ce qui n'est évidemment pas équitable et entraîne un coût de traitement en moyenne élevé. Autant de désavantages qui poussent naturellement vers une solution plus élaborée : la politique FIFO avec priorités et prise en compte de l'ancienneté.

5.2 Priorités

Nous avons implémenté une deuxième politique d'ordonnancement. Nous utilisons 4 listes différentes, correspondant aux priorités possibles : basse, moyenne, haute, ultra-haute. Il est possible de choisir lors de la création d'un thread, de choisir la priorité grâce à l'argument *attr* (voir le début de ce rapport). Notre librairie **garantit** que les threads se partagent le temps de calcul comme suit :

ultra-haut : 4/10 du temps de calcul

haut : 3/10 du temps de calcul
moyen : 2/10 du temps de calcul
bas : 1/10 du temps de calcul

Pendant la création on peut choisir entre bas moyen et haut uniquement. Puis, la librairie décide d'ajouter des bonifications dans les cas-suivant :

- si le thread possède le verrou d'un mutex, il bloque potentiellement d'autres thread : on le gratifie de 1. (si il est en priorité basse, on le monte en moyen, s'il est en moyen on le monte en haut, etc...)
- si le thread est attendu via un join, on fait de même que précédemment.

Cela justifie l'utilisation d'une quatrième liste.

Le problème posé est qu'il y a un certain délaissement des tâches de faibles priorités. C'est pourquoi nous prenons également en compte le temps depuis lequel un thread n'a pas été exécuté pour le gratifier en priorité. Un autre problème est qu'il y a de nombreux changements de priorités pour que le tout cela soit équitable. Ainsi cela nécessite une bonne gestion des files de priorité, ce qui reste assez difficile à évaluer.

6 Comparaisons avec la pthread

Il est utile de comparer notre librairie à la librairie pthread qui est la librairie de référence. Notons déjà une différence fondamentale entre pthread et mythread : notre librairie ne respecte pas pleinement les standards POSIX et ne remplit pas du tout la totalité des spécifications. Il n'est donc pas possible de substituer notre librairie à la pthread (évidemment). De plus notre librairie ne gère pas les situations d'interblocage, et ne les évite donc pas le moins du monde.

6.1 Primitives de synchronisation

Nous avons testé les performances temporelles comparées de pthread et mythread. Voici les tests que nous avons effectué et les résultats trouvés.

- Le premier test consiste à mesurer l'exécution d'un thread qui ne fait rien. (Il y a donc uniquement un mythread_create et ce qui suit ...). Les résultats sont : 92 micro-secondes (pthread) et 20 micro-secondes (mythread).
- Le deuxième test consiste à verrouiller puis déverrouiller un mutex et mesurer le temps que cette opération prend. Les résultats sont : 49 micro-secondes (pthread) et 20 micro-secondes (mythread).
- Le troisième test consiste à mesurer le temps qui s'écoule entre le moment où un thread envoie un signal via **mythread_cond_signal** et le moment où le thread qui attendait cette condition via **mythread_cond_wait** est effectivement débloqué. Les résultats sont : 97 micro-secondes (pthread) et 22 micro-secondes (mythread).

7 Conclusion

Finalement, on se rend compte que les résultats donnés par notre librairie sont assez bons. Cela vient du fait que notre librairie implémente les threads de manière beaucoup plus simple

que la pthread. De plus nous ne cherchons pas à détecter les interblocages, ce qui est un gain de temps certainement considérable. Nos structures sont plus faciles à utiliser car répondent à des besoins plus faibles.

Si un travail est à faire sur notre librairie c'est certainement du côté de la détection d'interblocages et dans l'amélioration de l'ordonnancement avec priorités qui pourrait prendre en compte en plus du temps depuis lequel un thread n'a pas été exécuté, le temps depuis lequel un thread a été créé. Il serait également utile de rajouter une priorité spéciale "Tâche de fond".