Bibliothèque de Threads

Rapport

Auteurs : *BOURDEAU Thibaud, DANDO Louis-Marie, HONORAT Alexandre, RINCEL Guillaume, SAGARDIA Elorri*

Encadrant : *M. FAVERGE Mathieu*

Enseignant : *M. GOGLIN Brice*

Deuxième année, filière informatique
Date : 18 avril 2014
Enseirb-Matmeca

Introduction

Le projet de système d'exploitation consiste à développer une bibliothèque de *threads*, basée sur l'interface de `p_thread`. Dans un premier temps, celle-ci devra se contenter d'un seul thread noyau, ainsi tous les *threads* que créera l'utilisateur seront exécutés de manière séquentielle.

L'objectif minimal est de pouvoir implémenter toutes les fonctions de prototypes identiques à ceux de `p_thread`, de sorte que tous les tests fournis foncionnent. Ce premier rapport présente l'avancement actuel du projet, en précisant les parties opérationnelles et non opérationnelles, ainsi que les choix effectués et à venir.

1 Fonctionnement de la bibliothèque

La bibliothèque, s'inspirant de `p_thread`, implémente les fonctions suivantes :
- `thread_self`
- `thread_create`
- `thread_yield`
- `thread_join`
- `thread_exit`

1.1 Organisation des sources

L'implémentation de ces fonctions requiert la définition de certaines structures regroupées dans le répertoire `src/includes`. Les principales structures sont :
- La structure de thread.
- La structure de liste.

La bibliothèque `ccan_list` a été choisie pour représenter les listes. Mais afin de pouvoir adapter le code à un autre type de liste, des fonctions d'abstraction ont été écrites dans `src/others/manip_list.c`. Celles-ci permettent d'effectuer les opérations basiques sur les listes : parcours, ajout, suppression. De même, aucun ordonnanceur – autre que l'utilisation basique des listes – n'est implémenté pour l'instant, mais ses deux fonctions principales (i.e. l'ajout d'un thread à la liste des threads en attente ou prêts, ainsi que la récupération du prochain thread à exécuter) sont regroupées dans un module à part : `src/core/ordo.c`.

Enfin les programmes de tests fournis ont été rassemblés dans le répertoire `src/prog_tests` et le fichier `thread.c` a été placé dans le répertoire `src/core` et contient le code principal du programme. La bibliothèque dynamique créée est `thread.so`, ce qui permet de ne compiler qu'une seule fois les tests.

1.2 Choix effectués

La programmation d'une bibliothèque de threads peut s'effectuer de nombreuses façons, en effet différentes politiques d'ordonnancement, de préemption, etc existent. La section suivante précisent ces choix.

1.2.1 Stockage des threads

La manière de stocker les threads en mémoire est déterminante sur l'ordonnancement. Les threads sont ainsi stockés selon leur état dans une des trois structures suivantes :
- la `waiting_list`, qui contient les threads ayant terminé mais n'ayant pas été récupérés par leur parent ;
- la `ready_list`, qui contient la liste des threads prêts à s'exécuter ;

– le thread `running`, qui contient le threads en cours d'exécution.

Actuellement, un seul thread noyau est employé pour exécuter l'ensemble des threads créés par l'utilisateur. L'utilité d'une liste `running` ne nous a, par conséquent, pas semblé nécessaire pour ce thread particulier. Par ailleurs notons que les deux listes implémentées sont utilisées comme des FIFOs : un thread y est toujours ajouté à la fin, et c'est le premier qui est récupéré.

1.2.2 La structure thread

Premièrement, un choix s'est imposé sur la définition de la structure thread. Un thread est de manière évidente, attaché à un contexte et la structure doit donc le contenir.

D'autre part, un thread, à chaque instant, est soit en attente, soit prêt à être executé, soit en execution. Le statut du thread est stocké au sein de la structure `thread` elle-même pour des raisons d'optimisation. En effet, il serait inutilement coûteux de devoir parcourir chaque liste à la recherche d'un certain thread pour obtenir son statut, notamment dans le cas d'un `join` qui doit vérifier si un thread est dans la `waiting_list`.

Par ailleurs, au vu du contenu des fonctions de `thread.c`, nous avons choisi de marquer le thread du `main` d'une manière différente. En effet, son allocation/desallocation s'effectue distinctement des autres : sa pile n'est pas allouée par la bibliothèque. Ce marquage se traduit par un attribut `is_main`.

Enfin, un attribut `retval` permet le stockage de la valeur renvoyée au niveau du `thread_join`. Pour gérer le cas d'un thread qui n'appelle pas la fonction `thread_exit`, une fonction auxiliaire a été implémentée dans la bibliothèque et prend comme argument la fonction passée à `thread_create` ainsi que son argument d'appel. Il s'agit d'un *wrapper* qui lui appelle la fonction `thread_exit` dans tous les cas.

1.2.3 Initialisation et Terminaison

Les fonctions de la bibliothèque nécessitent que certains objets soient instanciés au préalable. C'est le cas des deux listes initialisées vides et du thread running qui est initialisé avec le thread du `main`. De la même façon, avant de quitter le programme, le thread running doit être détruit (les listes ready et waiting sont déjà vides à ce moment là). C'est pourquoi les fonctions `thread_init` et `thread_quit` sont appelées automatiquement car définies avec des attributs *constructor* et *destructor* respectivement.

D'autre part, afin d'éviter la duplication de code, une fonction d'allocation `thread_construct` et de désallocation `thread_destruct` internes au fichier `thread.c` ont été ajoutées. La fonction d'allocation est appelée à la fois par `thread_init` et par `thread_create` et c'est elle qui effectuera des traitements différents en fonction du thread (running et/ou main ou cas général). De la même manière, la fonction de désallocation peut être appelée par `thread_quit` ou `thread_join`. En effet, dès qu'un thread est récupéré par son parent, il doit être détruit.

1.2.4 Création de contexte

Le choix entre `swapcontext` et `setcontext` dépend du besoin de retenir le contexte précédent ou non. C'est ainsi que `thread_yield` utilise `swapcontext` alors que `run_thread`, appelé uniquement lorsque la `ready_list` ne contient qu'un élément, utilise `setcontext`.

1.2.5 Le thread issu de main

Le thread principal est le seul thread que la bibliothèque ne crée pas : son traitement demande par conséquent quelques adaptations. En particulier, sa pile n'est pas allouée. Il est donc hors de question de la libérer - du reste, nous avons été incapables de récupérer son adresse.

Par ailleurs, la fonction `exit` est la seule à pouvoir libérer proprement la mémoire allouée pour le thread principal. Cependant, si elle est exécutée depuis un autre contexte que le thread principal (avec sa propre pile), la fonction `exit` dysfoncionne. Il faut donc recourir à une astuce pour, au moment où le thread `main` appelle `thread_exit`, sauvegarder un contexte sur le point d'apppeler `exit` tout en le remplaçant par celui d'un thread prêt.

Au moment de quitter le programme (si lors d'un appel à `thread_exit`, il n'y a plus de thread prêt à qui donner la main – cf test 12-join), on rend la main à ce contexte, dont la pile est la pile principale, et qui exécute donc `exit` correctement.

2 Validation des résultats

À ce stade d'avancement du projet, la validation des résultats n'est pas totale. Cependant quelques mesures ont été effectuées et un script permet de comparer le comportement de la bibliothèque implémentée par rapport à `p_thread`.

2.1 Passage des tests fournis

Tous les tests fournis par M. Goglin fonctionnent avec la bibliothèque implémentée `thread.so`. Aucune erreur mémoire ne se présente pour ces tests (aucune fuite mémoire, aucune erreur de lecture). Valgrind détecte cependant un changement de pile (et émet un avertissement). Mais cette manipulation est maîtrisée : elle permet en effet de désallouer correctement la pile dans le test `12-join-main`.

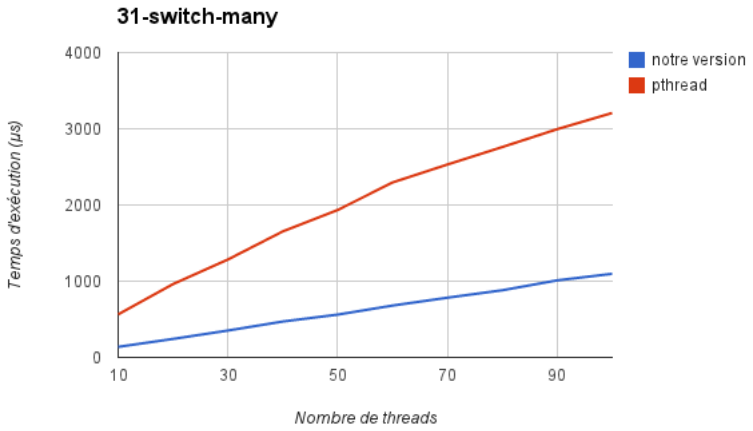
Les résultats sont ensuite comparés à ceux de `p_thread` afin de les valider. Le script `run_tests.sh` permet de lancer tous les tests avec chacune des deux bibliothèques et affiche un `diff` pour l'affichage des tests, ainsi qu'un autre pour les résultats de Valgrind.

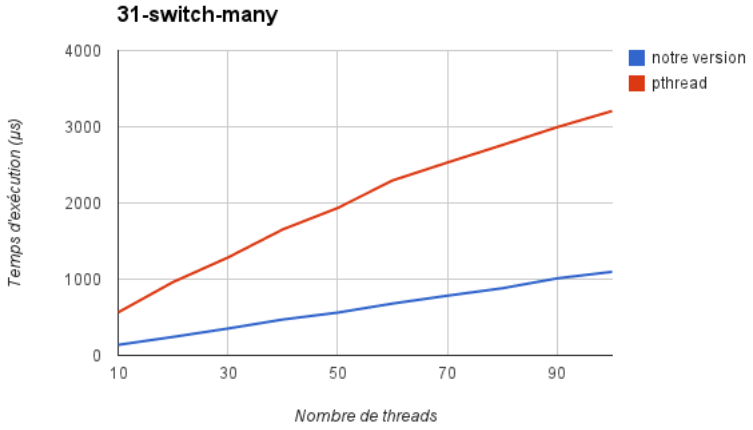
2.2 Comparaison avec `p_thread`

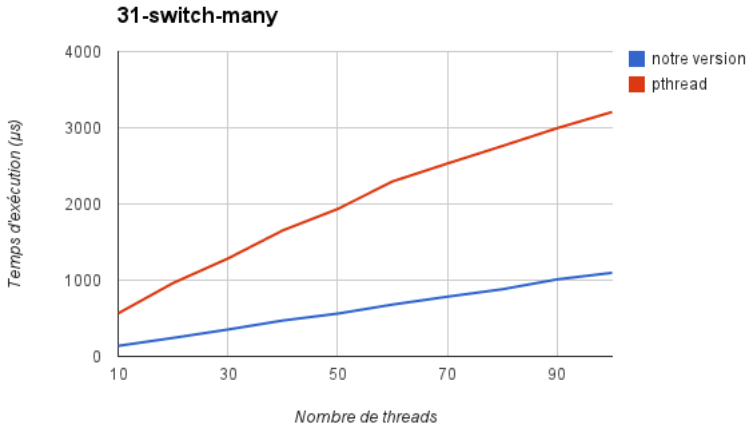
Une comparaison de performance avec `p_thread` peut sembler controversable puisque nous n'utilisons qu'une seul thread noyau. Les temps de création et destruction des threads seront donc inférieurs pour `thread.so`, en revanche les threads ne s'exécuteront pas en parallèle donc les temps de calculs peuvent être plus longs.

Ainsi la comparaison des temps sur les tests 31, 32 et 51 (respectivement création de threads et yield, création récursive de threads et yield et enfin fibonacci) devraient donc donner les résultats suivants : 31 et 32 plus rapides pour `thread.so`, 51 plus rapide pour `p_thread`.

Tous les tests ne sont toutefois pas en accord avec les prévisions faites : notamment en ce qui concerne les tests 32 et 51.


31-switch-many
4000
3000
2000
1000
0
10
30
50
70
90
Temps d'exécution (µs)
Nombre de threads
notre version
pthread


31-switch-many
4000
3000
2000
1000
0
10
30
50
70
90
Temps d'exécution (µs)
Nombre de threads
notre version
pthread


31-switch-many
4000
3000
2000
1000
0
10
30
50
70
90
Temps d'exécution (µs)
Nombre de threads
notre version
pthread