Enseirb-matmeca 2020/2021
Département Informatique / 2A

Projet Système - Threads en espace utilisateur

Rapport

Réalisé par

- Sohaib Errabii
- Saad Margoum
- Reda Chaguer
- Hamza Benmendil
- Mohamed Fayçal Boullit

Encadré par

M. Brice Goglin

Introduction

Le but de ce projet est de créer une interface de programmation proche de **pthread**, par création d'une bibliothèque des **threads en espace utilisateur**. Nous avons implémenté deux versions de la librairie, une implémentation optimale appelée `core` et une version appelée `extra`. Ce rapport décrira dans la première partie, l'implémentation `core`. Ensuite la deuxième partie présentera les fonctionnalités supplémentaires de la version `extra`. Finalement, une analyse de la performance ainsi qu'une comparaison avec la bibliothèque `pthread` est effectuée à la troisième partie.

1 L'implémentation optimale

Cette partie expliquera les détails de l'implémentation optimisée en terme de la performance et l'ordonnancement et elle justifiera les choix effectués.

1.1 L'initialisation de la librairie

La librairie utilise deux variables globales, `main_th` pour le main thread et `current_th` pour le thread en cours d'exécution. Elle utilise également trois listes de type `TAILQ` de l'API *queue.h*, les listes `runnable` et `high_prio` contiennent les threads qui ne sont pas terminés, elles sont populées selon la stratégie d'ordonnancement expliquée dans 1.4, la liste `abandoned` pour les threads qui sont terminés mais dont la mémoire n'est pas encore libérée (suite à un join). Un constructeur permet d'initialiser le contexte du main thread.

1.2 La structure thread

La structure thread contient un membre de type *ucontext_t* pour sauvegarder le contexte du thread. La fonction à exécuter, son argument et un pointeur pour sauvegarder la valeur de retour. Un membre *flags* qu'on utilise pour stocker les propriétés d'un thread, dans la version optimisée on définit deux bit flags. `MAIN` qui concerne le main thread et `JOINABLE` pour les threads qui sont terminés.
Enfin, notre stratégie d'ordonnancement nécessite l'ajout d'un enum `priority` (cf 1.4) et d'un pointeur appelé `master` ou bien `join_parent` qui pointe vers le thread qui attend sa terminaison (un join).

1.3 La création d'un thread

La création d'un nouveau thread consiste à initialiser et allouer la mémoire pour une instance de la structure thread, initialiser un nouveau contexte et allouer la mémoire pour la pile du contexte. En particulier le membre `uc_link` est mis à NULL et la fonction d'entrée du thread passée en paramètre à la fonction `makecontext` est une fonction appelée

`thread_runner` qui elle même utilise la fonction et son argument stockés dans la structure thread pour appeler la fonction fournie par l'utilisateur.

Enfin, le thread est ajouté à la fin de la liste `runnable`. On ne fait pas directement un changement de contexte vers le nouveau thread.

1.4 L'ordonnancement

Dans la version optimale, la stratégie de l'ordonnancement des threads consiste à définir trois niveaux de priorité, `HIGH`, `NORMAL` et `LOW`. Le premier niveau définit les threads ayant une priorité élevée qui devraient prendre la main dans les plus brefs délais possibles, la deuxième rassemble les threads en état normal et qui peuvent attendre la fin des threads `HIGH`, enfin `LOW` est la catégorie des threads qui doivent nécessairement attendre la fin d'éxecution des autres threads.

Concrètement, un thread est de priorité normale quand il vient d'être créé et dans ce cas il se trouve dans la liste `runnable`. Quand on fait un join sur un thread de priorité normale, on lui donne une priorité `HIGH` et on le met dans la liste `high_prio`, la figure 1 illustre le processus d'insertion lorsque la fonction `thread_join` est appelée D'autre côté, la priorité du thread qui fait le join devient `LOW` et il passe la main avec la fonction `thread_yield`, cela permet d'éviter l'attente active. Puisque le thread attendu a une référence vers son `join_parent`, on peut lui remettre la priorité à `HIGH` et l'ajouter à la liste `high_prio` quand le thread attendu termine son exécution.
Au moment du passage vers un autre thread avec `thread_yield`, le premier choix est fait à partir de la liste des threads de priorité `HIGH` si elle n'est pas vide, et sinon le thread suivant se trouve à la tête de la liste des threads de priorité `NORMAL`.

Nous avons également ajouter une deuxième implémentation du yield, qui permet de faire un yield directement vers un thread en spécifiant son identifiant. Puisque `runnable` est doublement chainée (`TAILQ`) cette opération s'effectue en temps constant. Cette fonction est utilisée en interne pour améliorer la performance quand un thread attend qu'un verrou devient disponible (Mutex 1.8).

Cette stratégie d'ordonnancement nous a permis d'obtenir un score d'équité de l'ordre de 0.999.

1.5 La terminaison d'un thread

La terminaison normale d'un thread est toujours effectuée par un appel à `thread_exit`. En effet, la fonction `thread_runner` permet d'appeler la fonction fournie par l'utilisateur et ensuite appeler `thread_exit` en passant la valeur du retour en argument.

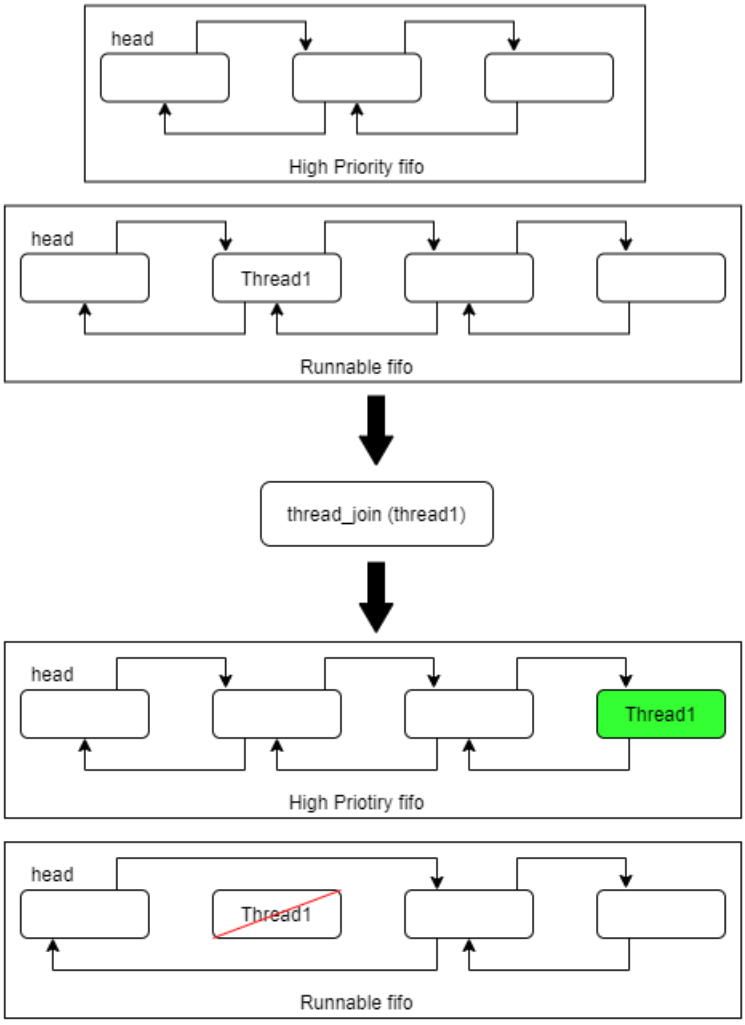

FIGURE 1 – Fonctionnement de la fonction `thread_join` avec les queues de priorité

La fonction `thread_exit`, met le thread courant dans la liste `abandonned`, et elle ajoute `JOINABLE` à l'attribut `flags` pour attendre un éventuel join. Enfin on reprend le contexte du prochain thread dans `high_prio` ou `runnable` si aucun thread n'a une priorité `HIGH`. Ce thread n'est retiré de la liste `abandonned` que si un autre thread lui fait un join et dans ce cas on libère la mémoire utilisée par son contexte.

Il existe un cas particulier quand les deux listes `runnable` et `high_prio` sont vides. Ce cas là ne se produit que si le main thread a effectué un `thread_exit`, ainsi le contexte du main thread est sauvegardé juste avant l'instruction `exit` de la fonction `thread_exit`. Donc en réactivant le contexte du main, on est capable d'appeler le destructeur de la librairie (cf 1.7).

1.6 L'attente d'un thread

La section 1.4 explique comment la stratégie d'ordonnancement avec des priorités nous a permis d'éviter l'attente active. Quand le thread attendu se termine et son `join_parent` reprend la main. Le thread attendu est enlevé de la liste `abandonned`, sa valeur de retour est récupérée et la mémoire allouée est libérée.

1.7 Destructeur de la librairie

Nous avons défini un destructeur, pour libérer la mémoire utilisée par les threads qui se sont terminés sans jamais recevoir un join. Tous ces threads seront dans la liste `abandonned` en fin du programme. Puisque les threads ne peuvent pas libérer la mémoire utilisée par leur contexte eux mêmes, le contexte du main thread doit être toujours activé à la fin du programme. (section 1.5).

1.8 Mutex

Afin de réaliser une implémentation du mutex on créer une structure **thread_mutex** qui contient deux attributs le premier nommé **is_destroyed** qui représente un indice booléen qui permet de déterminer si le **Mutex** est détruit, le deuxième nommé **locker** qui contient l'adresse du **thread** qui a verrouillé l'accès a un bloc d'instruction pour les autres threads et qui doit prendre la main dans l'exécution, si cet attribut est égale à la valeur **NULL** alors aucun des threads n'a verrouillé l'accès un certain bloc d'instruction. L'implémentation des threads réalisé fournit **4 fonctions** principales, la première est **thread_mutex_init()** qui permet l'initialisation d'un mutex, la deuxième est **thread_mutex_destroy()** qui permet la destruction d'un mutex, la troisième est **thread_mutex_lock()** qui fait **yield** pour donner la main au thread qui a verrouillé l'accès afin qu'il prend la main d'exécution et finalement la fonction **thread_mutex_unlock()** qui permet de redonner la main aux autres thread qui était suspendu par le thread appelant de la fonction **thread_mutex_lock()** .

2 Les fonctionnalités supplémentaires

Cette partie présente les fonctionnalités supplémentaires que nous avons implémentées, ils n'ont pas été intégré dans l'implémentation

2.1 Les signaux

Nous avons ajouter dans la structure du thread un attribut `sig` qui signifie le dernier signal reçu par ce thread. Nous avons implémenté aussi trois fonctions dans le fichier `signal.c` ; une qui envoie un signal à un thread (`sig_send()`), la deuxième attend jusqu'à que le thread reçoit le signal entré en paramètre et appel le handler qui est lui même un paramètre de la fonction, sinon le handler par défaut est appelé en cas d'entrer `NULL`(`sig_wait()`). Nous avons implémenté trois signales spéciaux avec leurs handler `SIGQUIT` qui quitte le thread après exécution du handler, `SIGSUSPEND` qui met le thread en attente en changeant son statue de priorité en `LOW`(les thread de priorité `LOW` sont en attente passive) et finalement `SIGCONT` qui réveille les thread en attente en changeant leur statue de priorité en `NORMAL`. Les signaux qui restent exécutent le handler par défaut qui n'est qu'un simple print.

2.2 Détection des débordements de pile

Afin de détecter le débordement de pile, on utilise `mprotect` pour mettre le flag `PROT_NONE` à une page mémoire avant et après la mémoire allouée pour la pile. Puisque `mprotect` ne peut agir que sur une adresse mémoire alignée, on utilise `mmap/munmap` pour allouer et libérer la mémoire de la pile et les pages de protection.

Ces changements permettent de recevoir un signal SIGSEGV si l'utilisateur cause un débordement de pile (on ne détecte toujours pas l'accès au-delà des pages protégées).

Afin de traiter le signal SIGSEGV dans une pile SIGALTSTACK, on l'initialise et l'installe dans le constructeur de la librairie. On ajoute également un bit flag `GOT_SIGSEGV` que le traitant ajoute à l'attribut `flags` du thread qui a causé le débordement avant d'appeler `thread_exit`. Puisque le traitant effectue des opérations non atomiques, on bloque tous les signaux reçus pendant son exécution.

La fonction `thread_join` a été modifiée pour tester si le thread en question à un flag `GOT_SIGSEGV`, dans ce cas, elle retourne -1 pour signaler l'erreur.

Nous avons ajouté un test qui utilise une fonction récursive infinie. Ce test doit être compilé sans optimisation (`-O3`), le thread se termine bien silencieusement et un join sur ce thread retourne -1 pour indiquer l'erreur.

La complexité temporelle ajoutée par cette fonctionnalité peut être négligée devant la complexité en espace. En effet, l'ajout de deux pages de mémoire pour chaque thread peut être assez conséquent quand on crée plusieurs threads.

2.3 Détection du deadlock de join

Afin de détecter le deadlock de join, nous avons ajouté un attribut `join_child` à la structure thread. On définit une relation join ; le thread qui fait le join est le parent et le thread cible est le fils. Ainsi, pour vérifier si un nouveau join peut causer un deadlock, il suffit de vérifier que le thread qui fait le join n'est pas un descendant (par rapport à la relation join) du thread cible. Sinon, la fonction `thread_join` retourne -1 pour indiquer l'erreur. En plus, on considère, comme la librairie `pthread`, que chaque thread ne peut avoir qu'un seul `join_parent`.

La complexité temporelle ajoutée par cette fonctionnalité est linéaire en la longueur des chaînes de join. C'est à dire, dans le pire des cas, elle est linéaire en le nombre des threads. La complexité en espace peut être négligée.

Il exite une autre méthode qui n'effectue pas ce parcours linéaire des relations join. En effet, puisque notre stratégie d'ordonnancement consiste à retirer des deux listes `runnable` et `high_prio` les threads ayant une priorité `LOW` (i.e. les threads qui ont fait un appel `thread_join`). Il devient possible de vérifier si il y a un deadlock quand le thread courant effectue un join alors que tous les autres threads ont une priorité `LOW` (les deux listes `runnable` et `high_prio` sont vides).

Cette méthode n'a pas été adoptée parce qu'elle ne permet pas de détecter immédiatement l'erreur et informer l'utilisateur. En effet, tant qu'il existe des threads qui ne font pas partie du cycle de join, elle ne détectera jamais le deadlock.

2.4 Préemption

Pour la préemption, on utilise un timer qui envoie un signal `SIGALRM`, ensuite le traitant appel la fonction `thread_yield`. Cette dernière fonction ainsi que la fonction `thread_exit` bloquent le signal `SIGALRM` parce qu'il n'est pas utile de préempter un thread qui est déjà entrain de passer la main ou se terminer. Cette protection consiste à bloquer le signal à travers la fonction `sigprocmask`. Avant de débloquer le signal, on utilise également le traitant `SIG_IGN` au cas où, un `SIGALRM` a été reçu pendant qu'il était bloqué.

En plus, on réinitialise le timer avant chaque appel à la fonction `swapcontext` pour assurer un ordonnancement équitable des threads.

Cette implémentation passe bien le test de la préemption, mais le temps d'exécution des tests a augmenté significativement. En effet, préempter quand les fonctions sont déjà coopératif n'est pas très efficace. Il faudra probablement intégrer la préemption dans la stratégie de l'ordonnancement par priorité, parce que la préemption uniforme indépendemment de la priorité des threads n'est pas efficace.

3 Analyse et comparaison avec pthread

Dans cette partie, on présente une comparaison de la performance des versions `core` et `extra` de notre implémenation avec la librairie pthread. Dans les figures 2, 3 et 4, les

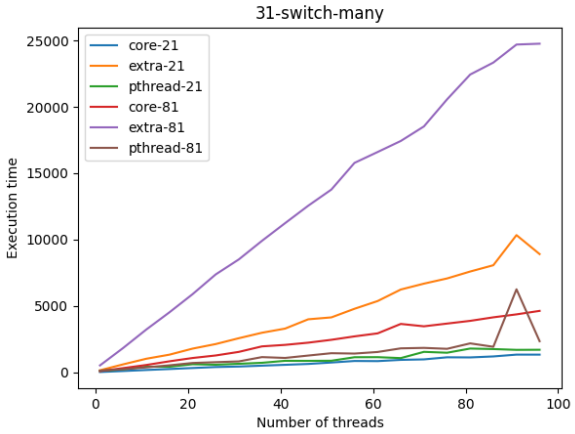

FIGURE 2 – Le test `31-switch-many`

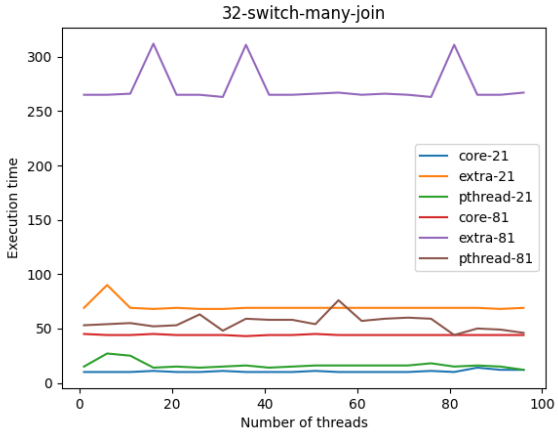

FIGURE 3 – Le test `32-switch-many-join`

numéros 21 et 81 correspondent aux nombre de yields.

D'après la figure 2, la complexité en temps de l'implémentation de la fonction yield est bien linéaire par rapport au nombre de threads. La performance de la version `core` est plutôt comparable à `pthread` mais on remarque que le pente de la version `extra` augmente significativement avec le nombre de yields.

D'après la figure 3, le temps d'exécution des yields suivies des joins et plus ou moins constant, ce qui est dû au fait que la complexité temporelle de la fonction yield est négligeable devant celle du join. On remarque aussi que le temps d'exécution de la version `extra` est toujours significativement plus grand que `pthread` et `core` quand le nombre de yields augmente.

Cette remarque est aussi valable pour la figure 4. Cela est probablement dû à la complexité linéaire introduite par la détection du deadlock. En effet, dans ces tests (en particulier `33-switch-many-cascade`) la longueur des chaînes des threads liés par une relation join sont très longues, d'où l'explosion en temps de calcul dans la figure 4.

D'autre côté, Les figures 5, 6, 7, montrent que la complexité de la création des threads est bien meilleure que pthread, ce qui était attendu, vu que notre librairie est très légère en terme de fonctionnalités par rapport à pthread. En effet, même avec l'ajout des fonctionnalités supplémentaires dans la version `extra`, on remarque que le coût de création des threads est toujours très faible par rapport à `pthread`.

Enfin, La figure 8 montre que notre version `core` est plus performante que pthread quand il s'agit des calculs directes qui n'utilisent pas des fonctionnalités avancées. En même temps en remarque la performance de notre version `extra` est très proche de `pthread`, en plus elle est également limitée par un $N \sim 20$ pour le calcul de la suite fibonacci.

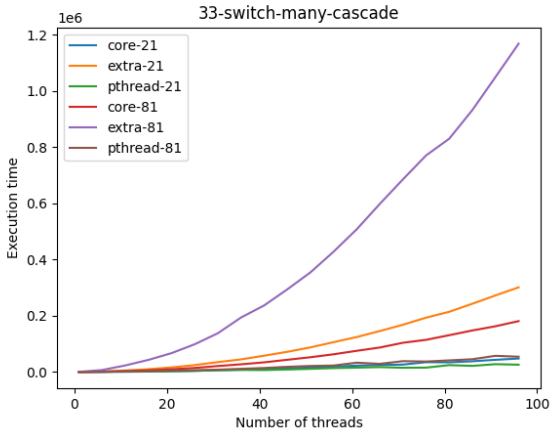

FIGURE 4 – Le test `33-switch-many-cascade`

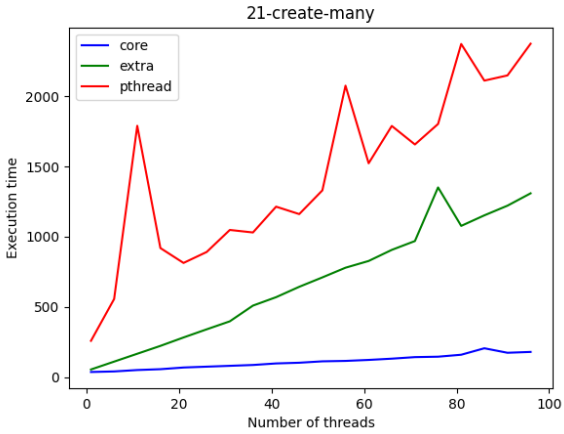

FIGURE 5 – Le test `21-create-many`

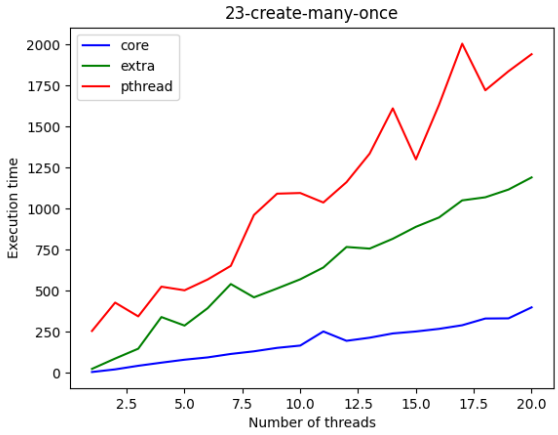

FIGURE 6 – Le test `23-create-many-once`

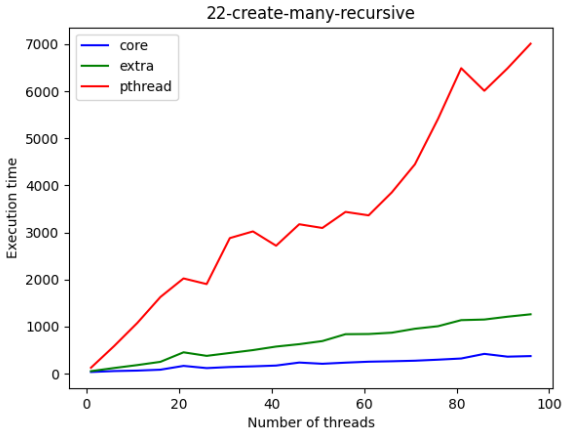

FIGURE 7 – Le test `22-create-many-recursive`

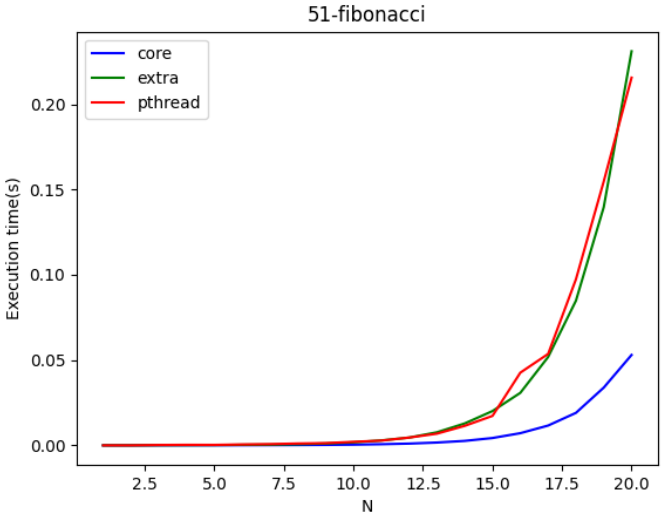

FIGURE 8 – Le test `51-fibonacci`

4 Conclusion

Au terme de ce projet nous avons pu obtenir des résultats satisfaisants de la part de notre librairie. L'ensemble des fonctionnalités de base ont été implémentées, sont opérationnelles et ont des performances correctes. Ce projet nous a permis de mettre en application de nombreuses compétences de cours vues au cours du semestre, ainsi que les compétences de programmation système du semestre précédent.