Bordeaux INP
ENSEIRB
MATMECA

Projet Système
Threads en espace utilisateur
Rapport intermédiaire

Équipe M1-E4
Avril 2021

Simon BULLOT
Antoine CHARTRON
Rémi DEHENNE
Sébastien DELPEUCH
Aymeric FERRON

Résumé Ce document présente l'implémentation d'une bibliothèque de threads côté utilisateur. Nous y représentons les threads utilisateur comme des structures, alloués lors d'une création de thread et chaînés au sein d'une file d'attente. Dans cette version, un seul thread utilisateur est exécuté à un instant donné, et l'ensemble des threads utilisateurs sont exécutés sur le même thread système (modèle $M : 1$). Chaque thread doit donner explicitement la main au thread suivant, sans préemption, et peut attendre la terminaison d'un autre thread. Même si les performances obtenues sont satisfaisantes compte tenu de l'exécution séquentielle des threads, nous souhaitons dans la suite du projet adopter un modèle de thread hybride $M : N$, afin de tirer parti des architectures de processeurs multi-cœurs.

Introduction

Dans le cadre du projet système de semestre 8 à l'ENSEIRB-MATMECA, nous sommes amenés à implémenter une bibliothèque de threads en espace utilisateur. L'objectif de ce document est de présenter, à mi-parcours, l'état d'avancement de la bibliothèque. Nous présentons pour cela les mécanismes mis en place afin d'obtenir une bibliothèque fonctionnelle et relativement performante.

Nous détaillons d'abord le fonctionnement de la création des threads (`thread_create`) et de leur ordonnancement (`thread_yield`), ainsi que les structures de données associées. Par la suite, nous discutons des mécanismes de synchronisation de threads avec `thread_join` et `thread_exit`. Enfin, nous comparons les performances de la bibliothèque de threads en espace utilisateur développée avec la bibliothèque de threads système `pthread`, et détaillons nos prochains objectifs sur la seconde partie du projet.

1 Structures de données et ordonnancement de threads

Dans un premier temps, nous détaillons les structures de données et fonctions nécessaires à la création et à l'ordonnancement de threads. Ces fonctionnalités sont assurées par les fonctions de l'interface `thread_create` et `thread_yield`.

1.1 Création de threads

Tout d'abord, la fonction `thread_create` se charge de créer un nouveau thread. Pour ce faire, le programme alloue dynamiquement de la mémoire à l'aide de la fonction `malloc`, afin d'y stocker le contenu d'une structure `thread_entry`. Cette structure, présentée en Figure 1, contient les champs nécessaires à la gestion de chaque thread par la bibliothèque : elle possède notamment un champ permettant de savoir si un thread a terminé ou non (`exited`), de stocker sa valeur de retour (`ret_val`) ou encore de sauvegarder le contexte d'exécution du thread (`context`).

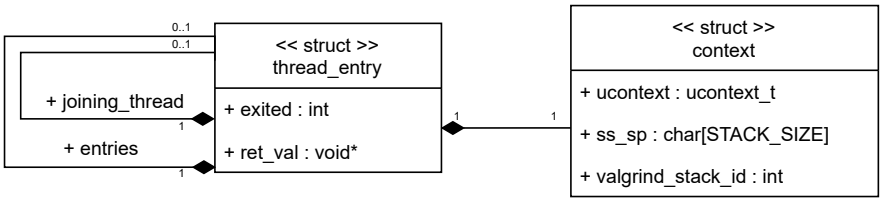

FIGURE 1: Diagramme UML de la structure `thread_entry`

La gestion de ces contextes d'exécution, telle que proposée par la bibliothèque `ucontext`, nécessite une structure stockant les informations de configuration (de type `ucontext_t`), ainsi qu'un pointeur vers un *buffer* dans lequel stocker les variables locales et la pile d'appels. Pour limiter le surcoût lié à un appel à `malloc`, les champs sont stockés directement dans la structure `context`. Par ailleurs, la structure `context` est incluse

dans la structure `thread_entry`, ce qui permet d'allouer de manière contiguë, et en un seul appel à `malloc`, l'ensemble des données d'un thread : ces deux structures sont définies indépendamment simplement pour des questions de factorisation de code et de maintenabilité, mais n'entraînent pas de surcoût à l'exécution.

1.2 Ordonnancement des threads

Une fois les threads de notre bibliothèque créés, il est nécessaire de les ordonnancer. Ces threads sont définis *en espace utilisateur* : dans cette première version, ils s'exécutent de manière purement séquentielle, sur un seul thread *système*.

Ainsi, à un instant donné, un unique thread s'exécute. Afin de conserver les threads en attente d'exécution, il est nécessaire de les sauvegarder dans une collection. Pour limiter au maximum le temps d'exécution, nous cherchons la structure de données avec la complexité en temps la plus faible sur chacune des opérations utilisées. En particulier, nous souhaitons que tout thread soit inséré à la fin de la collection, et qu'il ne soit accédé ou supprimé que lorsqu'il arrive en tête de la collection : ces opérations correspondent à celle d'une file, aussi appelée « FIFO[1] ».

La complexité de ces opérations pour différentes implémentations est comparée dans le Tableau 1. Nous constatons que parmi les trois implémentations proposées (tableaux, listes et files), seule la file ne possède que des opérations en temps constant. Comme nous ne souhaitons pas pouvoir supprimer un élément de la file en temps constant à partir de son adresse, nous optons pour une implémentation de file simplement chaînée, appelée *simply-linked tail queue* ou `STAILQ` dans les bibliothèques système BSD. Nous limitons ainsi le coût lié au chaînage inverse entre les éléments d'une file doublement chaînée (`TAILQ`), pour gérer de manière efficace la file d'exécution de threads.

Opération	**Tableau**	**Listes** `SLIST` `LIST`	**Files** `STAILQ` `TAILQ`
Accès en tête	$O(1)$	$O(1)$	$O(1)$
Suppression en tête	$O(n)$	$O(1)$	$O(1)$
Insertion en queue	$O(n)$	$O(n)$	$O(1)$

TABLEAU 1: Complexité des opérations de file (FIFO) selon la structure de données utilisée (n : nombre d'éléments dans la collection)

Dès lors, la fonction `thread_yield`, qui permet à un thread de passer la main à un autre thread de la file d'exécution, se contente simplement d'ajouter le thread courant en bout de file. Puis, elle récupère le thread en haut de file, le supprime de la file et le stocke dans la variable globale `current_thread`. La Figure 2 schématise cette situation : le thread courant `thread_0` appelle `thread_yield` et est enfilé dans la file d'attente, pour laisser `thread_1` s'exécuter.

Enfin, une fonction `thread_self` est proposée à l'utilisateur de la bibliothèque. Elle renvoie simplement l'identifiant du thread qui l'appelle, en utilisant l'adresse du thread courant, stocké dans `current_thread`. Une fois ces fonctions de base de création et d'ordonnancement de threads implémentées, il reste encore à permettre à des threads de terminer et de se synchroniser entre eux.

2 Attente et terminaison de threads

Un utilisateur d'une bibliothèque de threads s'attend généralement à pouvoir bloquer un thread en attendant qu'un autre thread termine. Pour cela, nous implémentons la fonction `thread_exit`, qui permet de terminer le thread qui l'appelle, et la fonction `thread_join`, qui fait attendre le thread qui l'appelle jusqu'à ce que le thread passé en paramètre lui redonne la main.

1. *First In, First Out*

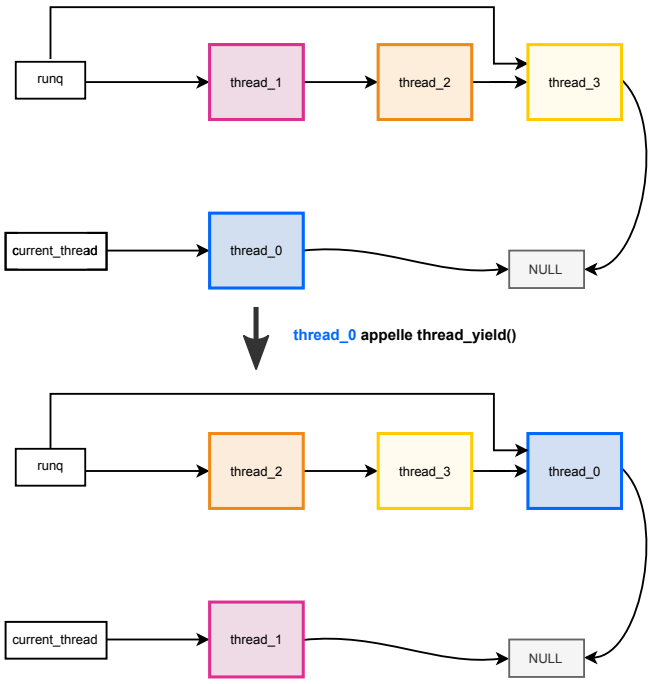

FIGURE 2: Organisation de la file d'exécution et du pointeur `current_thread` avant et après un `thread_yield`

2.1 Synchronisation des threads selon l'ordre d'exécution

Les fonctions `thread_exit` et `thread_join` nécessitent de prendre en compte l'ordre d'exécution des threads. En effet, dans la version actuelle, le thread attendu peut terminer avec `thread_exit` avant que l'autre thread ne l'attende avec `thread_join`, ou inversement. Dans une version préemptive ou multi-cœurs, ces fonctions pourraient même être exécutées en concurrence ou en parallèle.

Nous devons ainsi permettre à ces deux threads de « communiquer » entre eux : un thread qui réalise un `thread_join` doit pouvoir déterminer si le thread qu'il attend a déjà terminé, et un fils doit connaître l'identité du thread qui l'attend afin de le réveiller lors d'un `thread_exit`. Le coût de ce mécanisme de communication devrait être constant en temps et en espace, indépendamment du nombre de threads endormis ou en attente d'exécution.

D'une part, il se peut qu'un thread y termine avant que le thread x l'attende en appelant `thread_join`, comme montré en Figure 3. Dans ce cas, y met son champ `exited` à `1` et écrit sa valeur de retour dans son champ `ret_val`. Puis, il détermine qu'aucun thread ne l'attend en lisant `NULL` dans son champ `joining_thread`. Enfin, il laisse la main au thread suivant, sans se réinsérer dans la file d'exécution.

Par la suite, le thread x est réveillé et appelle `thread_join`. Il consulte alors le statut de y, dont il connaît l'adresse : le champ `exited` de y vaut `1`, y a donc terminé. De la même manière, x lit la valeur de retour de y `ret_val` et recopie la valeur de par x à l'adresse donnée par l'utilisateur de `thread_join`. Enfin, x libère les ressources occupées par y et continue à s'exécuter.

D'autre part, le thread x peut attendre avec `thread_join` le thread y qui n'a encore appelé `thread_exit`. Cette situation est représentée en Figure 4. De façon contraire, il lit une valeur `exited` à `0` dans la structure du thread y. Le thread x doit donc dormir jusqu'à ce qu'il soit réveillé par y : il stocke son adresse dans le champ `jonining_thread` de y et laisse la main au thread suivant, sans se réinsérer dans la file d'exécution. Cette méthode permet de ne pas polluer la file d'exécution avec des threads endormis, et permet de réveiller les threads endormis en temps et en espace constants. Cette manière de procéder est satisfaisante dans la mesure où un thread ne peut être attendu que par au plus un thread.

Puis, le thread y sait qu'il est attendu lors de son `thread_exit` en lisant l'adresse de x dans `joining_thread`. Après avoir écrit sa valeur de retour et son statut dans ses champs, il redonne alors la main à x, qui recopie la valeur de retour à l'adresse souhaitée par le client, libère les ressources de y, et continue à s'exécuter.

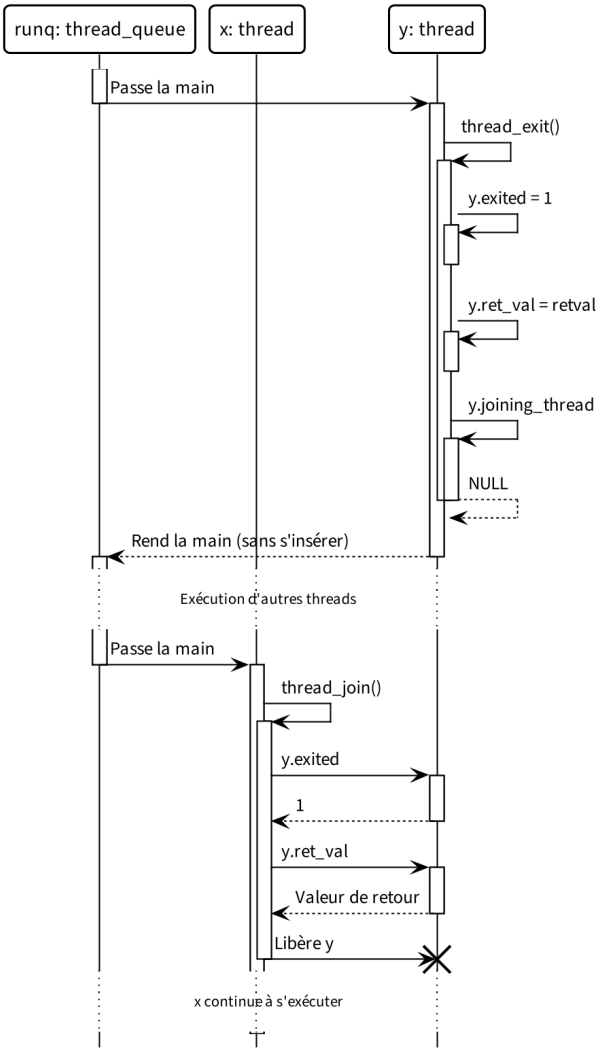

FIGURE 3: Diagramme de séquence de terminaison d'un thread y avec `thread_exit` *avant* son attente par x dans `thread_join`

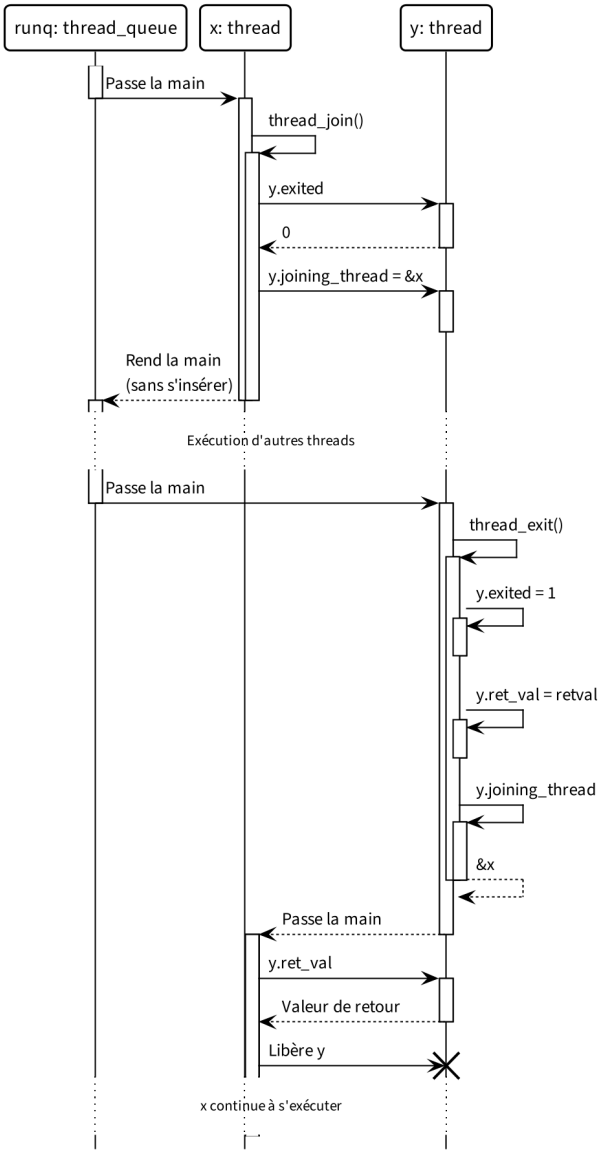

FIGURE 4: Diagramme de séquence de terminaison d'un thread y avec `thread_exit` *après* son attente par x dans `thread_join`

2.2 Terminaison implicite de threads

Finalement, il est également possible qu'un thread n'appelle pas explicitement la fonction `thread_exit`. Dans un tel cas, nous souhaiterions réutiliser la valeur de retour de la fonction comme valeur de retour écrite par `thread_exit`, comme si l'utilisateur avait véritablement appelé `thread_exit`. Pour cela, la fonction réelle de l'utilisateur passée lors d'un `thread_create` est « emballée » dans une autre fonction `thread_function_wrapper`, qui prend en argument un pointeur vers une fonction de l'utilisateur, et un pointeur générique vers les arguments à passer à la fonction de l'utilisateur. Cette fonction « d'emballage » appelle la fonction de l'utilisateur sur ses arguments, récupère sa valeur de retour et appelle `thread_exit` en lui passant la valeur de retour. Si la fonction de l'utilisateur appelle déjà `thread_exit`, alors c'est le `thread_exit` de l'utilisateur qui est prioritaire (puisque le premier `thread_exit` ne retourne pas et stoppe l'exécution du thread). Ainsi, si la fonction, à l'origine, utilisait déjà `thread_exit`, le comportement du thread ne change pas.

Néanmoins, nous avons été contraints de ne pas utiliser le mécanisme « d'emballage » pour le thread associé au `main`, la fonction `main` étant appelée automatiquement à l'exécution du programme. Cela oblige l'utilisateur à utiliser `thread_exit` si un autre thread attend le thread principal.

Conclusion

À la moitié du projet, l'ensemble des fonctions de base de la bibliothèques sont fonctionnelles et s'exécutent en un temps satisfaisant, ce qui tend à confirmer la pertinence de nos choix de structures de données et d'algorithmes. Sur certaines expériences, la bibliothèque de threads côté utilisateur apparaît même plus performante que `pthread` qui crée des threads système. Par exemple, la Figure 5 montre que pour le cas où de nombreux threads sont créés, et qu'ils réalisent de nombreuses fois l'opération `thread_yield`, notre bibliothèque termine en moyenne en 5 millisecondes, alors que la bibliothèque `pthread` dépasse les 25 millisecondes au-delà de 700 threads créés. En revanche, dans l'expérience où chaque thread en crée un autre puis l'attend, notre bibliothèque est équivalente à `pthread`, comme présenté en Figure 6.

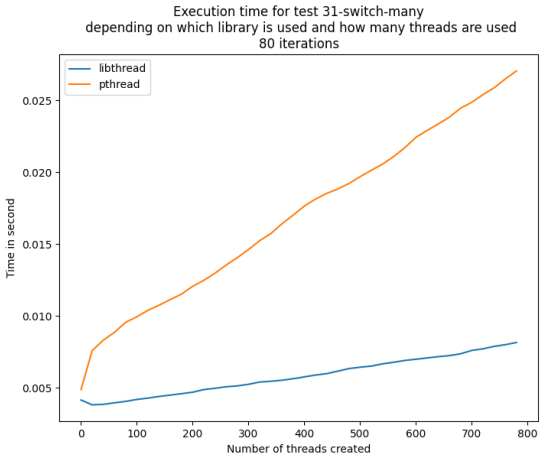

FIGURE 5: Comparaison du temps d'exécution du test `31-switch-many` entre `pthread` et `libthread`

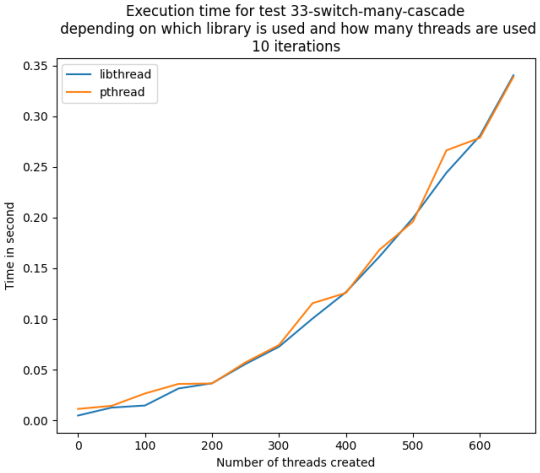

FIGURE 6: Comparaison du temps d'exécution du test `33-switch-many-cascade` entre `pthread` et `libthread`

Nous supposons que la bibliothèque `pthread` est parfois ralentie par le coût de création et de changement de contexte entre les threads système, tandis que notre bibliothèque est pénalisée par son exécution purement séquentielle. En utilisant un modèle hybride de threads dit « *many-to-many* » $(M : N)$, et en conservant les threads système dans une *thread pool*, nous pourrions exécuter plusieurs threads utilisateur en parallèle sur plusieurs threads système, de manière à limiter le coût de création et de changement de contexte des threads système. Il s'agit de l'un des objectifs avancés que nous souhaiterions réaliser. Néanmoins, le modèle de threads *many-to-many* ne semble pas nécessairement plus performant que les threads système, notamment lorsque des mécanismes de priorité entre threads sont implémentés. Certaines de ces limitations sont notamment expliquées dans [Mcewan *et al.*(2007)Mcewan, Schneider, Ifill, Welch, et Brown].

Enfin, nous souhaiterions également améliorer la génération des graphes de performance, et implémenter des mécanismes d'envoi et de réception de signaux, ainsi qu'une détection des *deadlocks* lorsque des threads s'attendent en cycle avec `thread_join`.

Références

[Mcewan *et al.*(2007)Mcewan, Schneider, Ifill, Welch, et Brown] Alistair A. MCEWAN, Steve SCHNEIDER, Wilson IFILL, Peter WELCH et Neil BROWN : C++csp2 : A many-to-many threading model for multicore architectures, 2007.