Bordeaux INP
ENSEIRB
MATMECA

Projet Système
Threads en espace utilisateur
Rapport final

Équipe M1-E4
Mai 2021

Simon BULLOT
Antoine CHARTRON
Rémi DEHENNE
Sébastien DELPEUCH
Aymeric FERRON

Résumé Ce document présente l'implémentation d'une bibliothèque de threads côté utilisateur, puis d'une bibliothèque de threads hybride $N : M$. 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. Une première version séquentielle exécute un seul thread utilisateur à un instant donné, et l'ensemble des threads utilisateurs sont exécutés sur le même thread système (modèle $M : 1$). En plus des fonctions de base, cette version est enrichie pour détecter les interblocages lors de la synchronisation de threads, et permet également l'envoi de signaux entre threads. Par ailleurs, une bibliothèque de threads hybride $M : N$ est implémentée, qui exécute plusieurs threads utilisateur en parallèle sur plusieurs threads système. Celle-ci fournit uniquement les fonctions de base, sans les signaux et les interblocages. Enfin, les performances des deux implémentations sont satisfaisantes : la version séquentielle est particulièrement performante sur des codes séquentiels et peu coûteux en terme de calculs, tandis que la version hybride est adaptée aux problèmes parallélisables, qui nécessitent de créer un nombre important de threads utilisateur pour effectuer des calculs.

Introduction

Dans le cadre du projet système du 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 l'état final de la bibliothèque à l'issue du projet. 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`. Puis, nous présentons les fonctionnalités avancées qui ont été implémentées, telles que la détection complète des interblocages lors des opérations `thread_join`, ainsi que l'envoi, la réception, le traitement et l'attente de signaux (`thread_kill`, `thread_signal` et `thread_sigwait`). Enfin, nous décrivons les modifications apportées afin de paralléliser la bibliothèque, et permettre à plusieurs threads utilisateur de s'exécuter sur plusieurs threads noyau.

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`).

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.

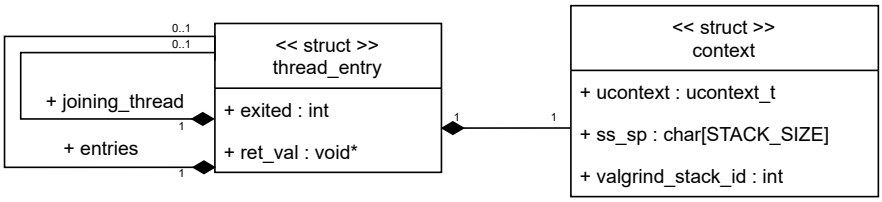

FIGURE 1: Diagramme UML de la structure `thread_entry`

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

1. *First In, First Out*

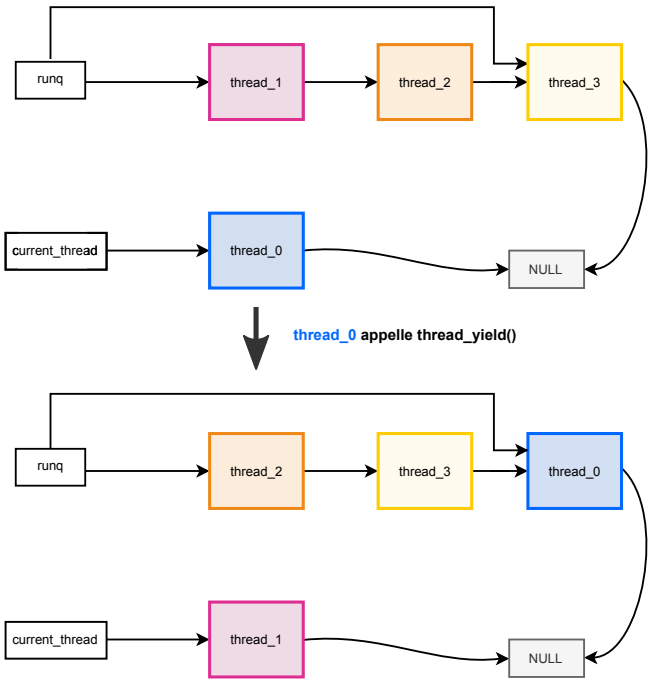

FIGURE 2: Organisation de la file d'exécution et du pointeur `current_thread` avant et après un `thread_yield`

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.

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 pas 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. Même si y pourrait être exécuté directement par x pour limiter le temps d'attente des threads, notamment lors d'appels récursifs, cette solution pourrait provoquer des famines de threads dans certains programmes, c'est-à-dire que certains threads ne seraient jamais exécutés, ou très rarement, ce qui n'est pas souhaitable. Cette méthode permet également 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,

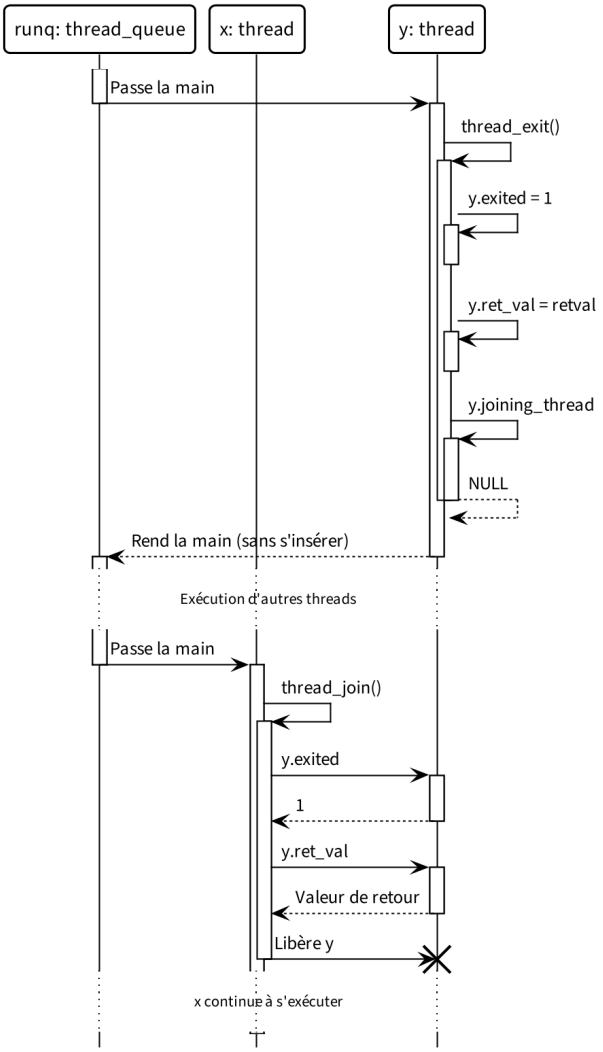

FIGURE 3: Diagramme de séquence de terminaison d'un thread y avec `thread_exit` *avant* son attente par x dans `thread_join`

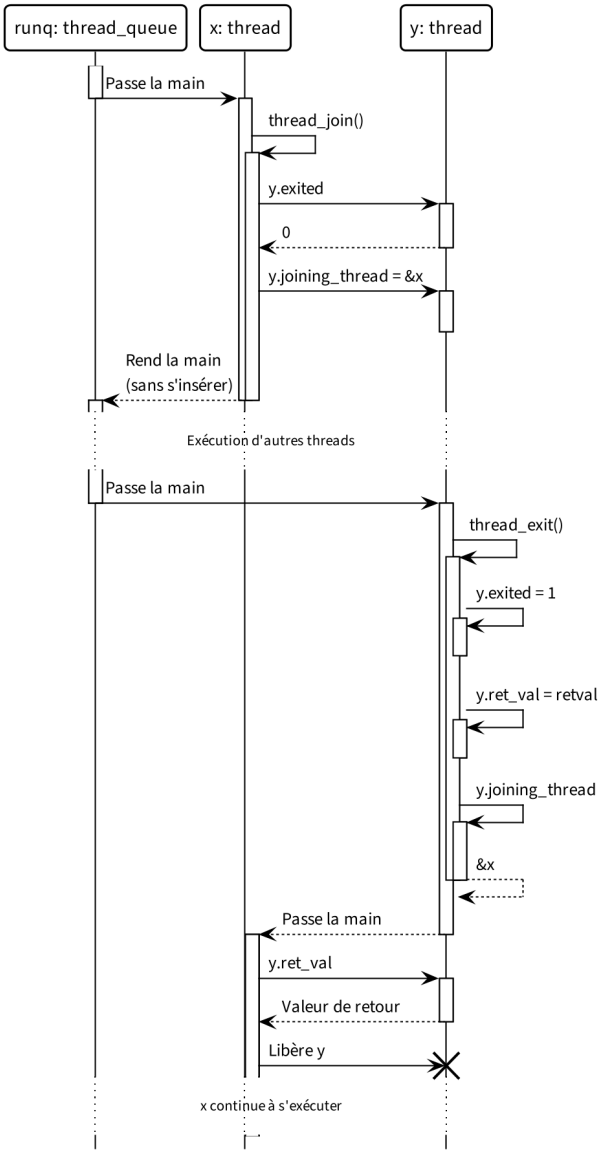

FIGURE 4: Diagramme de séquence de terminaison d'un thread y avec `thread_exit` *après* son attente par x dans `thread_join`

qui recopie la valeur de retour à l'adresse souhaitée par le client, libère les ressources de y, et continue à s'exécuter.

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.

2.3 Comparaison entre `pthread` et `libthread`

La bibliothèque de threads de base est donc fonctionnelle, et s'exécute 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 sur une machine dotée de 24 cœurs physiques, dans 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 un peu plus de 15 millisecondes, alors que la bibliothèque `pthread` dépasse les 50 millisecondes au-delà de 1500 threads créés : les coûts d'exécution sont linéaires en fonction du nombre de threads, mais le coefficient directeur est plus important pour `pthread`, notamment à cause des coûts de création et de changement de contextes des threads système. En revanche, dans l'expérience où chaque thread en crée un autre puis l'attend, notre bibliothèque est relativement similaire à `pthread`, comme présenté en Figure 6.

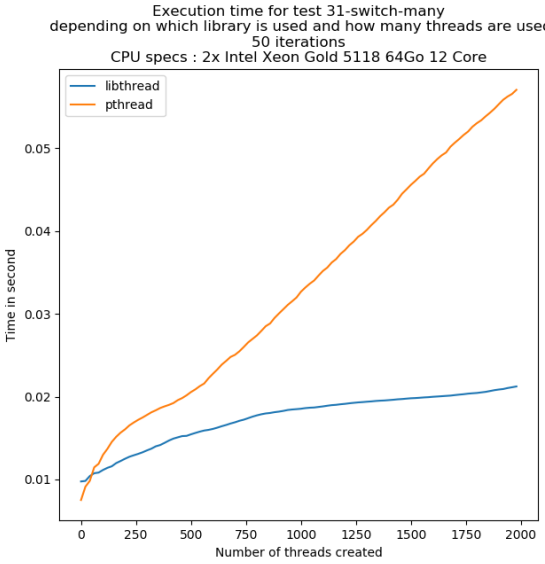

FIGURE 5: Comparaison du temps d'exécution du test `31-switch-many` entre `pthread` et `libthread`

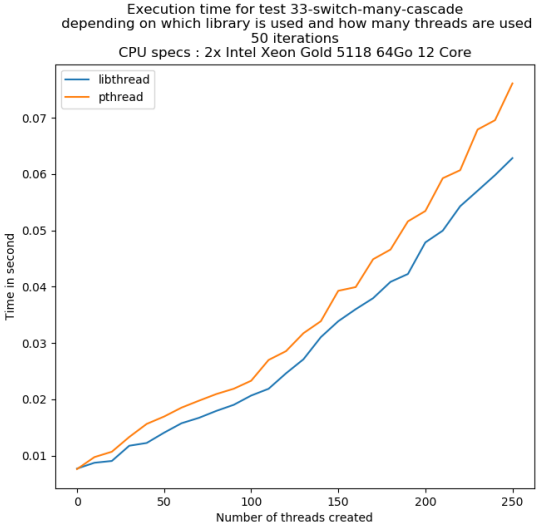

FIGURE 6: Comparaison du temps d'exécution du test `33-switch-many-cascade` entre `pthread` et `libthread`

3 Envoi et réception de signaux

Une fois la version de base de `libthread` implémentée, nous avons enrichi la bibliothèque de mécanismes avancés, notamment l'envoi et la réception de signaux entre threads. Nous rappelons que les signaux sont des messages envoyés entre threads qui permettent de déclencher un comportement particulier chez le thread receveur, en associant un signal à une fonction de rappel chargée de traiter ledit signal.

3.1 Envois de signaux et paramétrage de threads

Pour qu'un thread réagisse après avoir reçu un signal, nous ajoutons des champs à la structure de thread existante. En particulier, un champ `sigmask` de type `int` sert de masque des signaux reçus : celui-ci permet de gérer `sizeof(int)` signaux différents, soit un total de 32 signaux (de 1 à 32, le signal 0 étant ignoré par convention). Un thread peut alors envoyer un signal à un autre thread à l'aide de la fonction `thread_kill`, qui met simplement à jour le masque du destinataire du signal en mettant à 1 le $i-1^{\text{ème}}$ bit de poids faible lorsque le signal i est reçu. Tout comme la fonction ANSI C `signal`, désormais dépréciée, le masque n'encode que l'information qu'un signal a été reçu ou non, pas le nombre de fois qu'un signal a été reçu. Cette implémentation permet de limiter l'espace mémoire utilisé, et de s'assurer que les opérations seront réalisées en temps constant.

Afin de définir une fonction de rappel à exécuter lorsque le thread recevra un signal, l'utilisateur utilise la fonction `thread_signal`. Pour un signal i et une fonction f, un appel à `thread_signal` consiste simplement à stocker l'adresse de f dans la $i-1^{\text{ème}}$ case d'un tableau de pointeurs de fonction de 32 éléments, stocké dans la structure thread.

3.2 Traitement des signaux reçus

Par ailleurs, il est nécessaire qu'un thread vérifie régulièrement les signaux reçus. À chaque appel à une fonction de la bibliothèque, telle que `thread_join` ou `thread_yield`, une fonction `inline thread_handle_`

signals vérifie si le sigmask du thread courant a un bit levé. Si c'est le cas, elle traite itérativement tous les signaux levés, en exécutant dans un nouveau contexte la fonction de rappel associée au signal traité. Ce contexte est lié au contexte actuel du thread, afin de poursuivre l'exécution lorsque le contexte de la fonction de rappel est terminé.

3.3 Attente de signaux

Une fonction thread_sigwait est également proposé à l'utilisateur de la bibliothèque. Celle-ci est chargée d'endormir un thread tant qu'il ne reçoit pas un signal parmi un ensemble de signaux, en imitant la fonction sigwait de la bibliothèque C. Ainsi, l'utilisateur peut définir un masque de signaux pour lesquels le thread doit se réveiller, puis appeler la fonction thread_sigwait ce qui l'endort. Pour cela, nous avons enrichi la structure des threads avec un masque sigwait_mask, ainsi qu'un pointeur sigwait_awaken_signal où stocker la valeur du signal ayant causé le réveil du thread.

Deux cas de figure peuvent se produire lors d'un sigwait. Tout d'abord, dans une version purement séquentielle sans préemption, l'utilisateur applique un masque au thread et le fait s'endormir. Pour cela, thread _sigwait retire le thread de la runq et exécute le thread suivant, en tête de file. Plus tard dans l'exécution du code, un signal est envoyé au thread endormi avec un thread_kill. Au moment où la fonction met à jour le sigmask du thread, elle vérifie si ce signal fait partie de son sigwait_mask. Si c'est le cas, elle le réveille en exécutant l'éventuelle fonction de rappel associée. Puis, le thread est placé à la fin de la runq, pour ne pas prioriser les threads réveillés vis-à-vis des autres et d'assurer l'équité entre les threads exécutés.

Par ailleurs, dans certains cas, des signaux pourraient être reçus concurremment à la mise en attente d'un thread. Dans ce cas, un thread B qui devait réveiller un thread A pourrait très bien envoyer le signal de réveil avant même que A n'ait le temps de s'endormir. Cela pourrait créer des problèmes dans la mesure où le thread A pourrait être perdu, car jamais réveillé par la suite. Afin de gérer ce cas, nous faisons des vérifications supplémentaires au moment du sigwait : s'il a déjà reçu un signal inclus dans son sigwait_mask, on ne le retire pas de la runq et on exécute simplement sa fonction de rappel.

Ainsi, il n'est pas nécessaire de stocker les threads en attente de signaux dans une file, puisque l'émetteur d'un signal doit directement spécifier l'adresse du thread destinataire : il peut donc le réveiller directement, en temps constant. Néanmoins, puisque la bibliothèque ne garde pas de trace des signaux endormis, c'est l'utilisateur de la bibliothèque qui est chargé de réveiller tout thread endormi pour ne pas s'exposer à des fuites mémoire.

4 Détection des cycles de threads

La bibliothèque pthread permet de détecter un interblocage entre deux threads qui s'attendent l'un l'autre lors de l'opération thread_join, ou lorsqu'un thread s'attend lui-même. Dans notre implémentation, nous souhaitons vérifier toute source d'interblocage, y compris des cas plus complexes, en évitant néanmoins que la complexité devienne linéaire en nombre de threads. Pour éviter cela, nous considérons chaque chaîne de threads qui s'attendent comme des ensembles, et implémentons une structure de données « Union-Find ».

4.1 Détection d'interblocages

Détecter un interblocage revient à détecter si un graphe possède un cycle ou non. Si cela se résout trivialement en complexité linéaire, ce n'est pas satisfaisant. En dérivant les fonctions associées à la structure « Union-Find », il est possible de détecter la présence d'un cycle tout en ayant une complexité amortie quasi-constante.

Cette structure permet de représenter la partition d'un ensemble. Un élément d'une partition sera représenté par un thread. Ces derniers possèdent notamment un pointeur vers un thread parent de la partition et un rang, qui permet d'estimer la profondeur de l'arbre. Par défaut, toutes les partitions sont constituées d'un thread qui est son propre parent. La structure est également associée à deux fonctions qui, après une simple modification, vont permettre de détecter un interblocage.

La première fonction liée à « Union-Find » est Find. Elle permet de trouver le représentant d'une partition, soit la racine de l'arbre. Il suffit donc de remonter de parent en parent pour retrouver la racine, le nœud dont le parent se référence lui-même. Find permet également de diminuer la complexité moyenne en aplanissant l'arbre : lorsque Find trouve la racine, les parents de tous les enfants précédemment parcourus sont remplacés par celle-ci. Ainsi, lors de future utilisation de la fonction, il y aura moins de nœuds à parcourir avant de retrouver la racine.

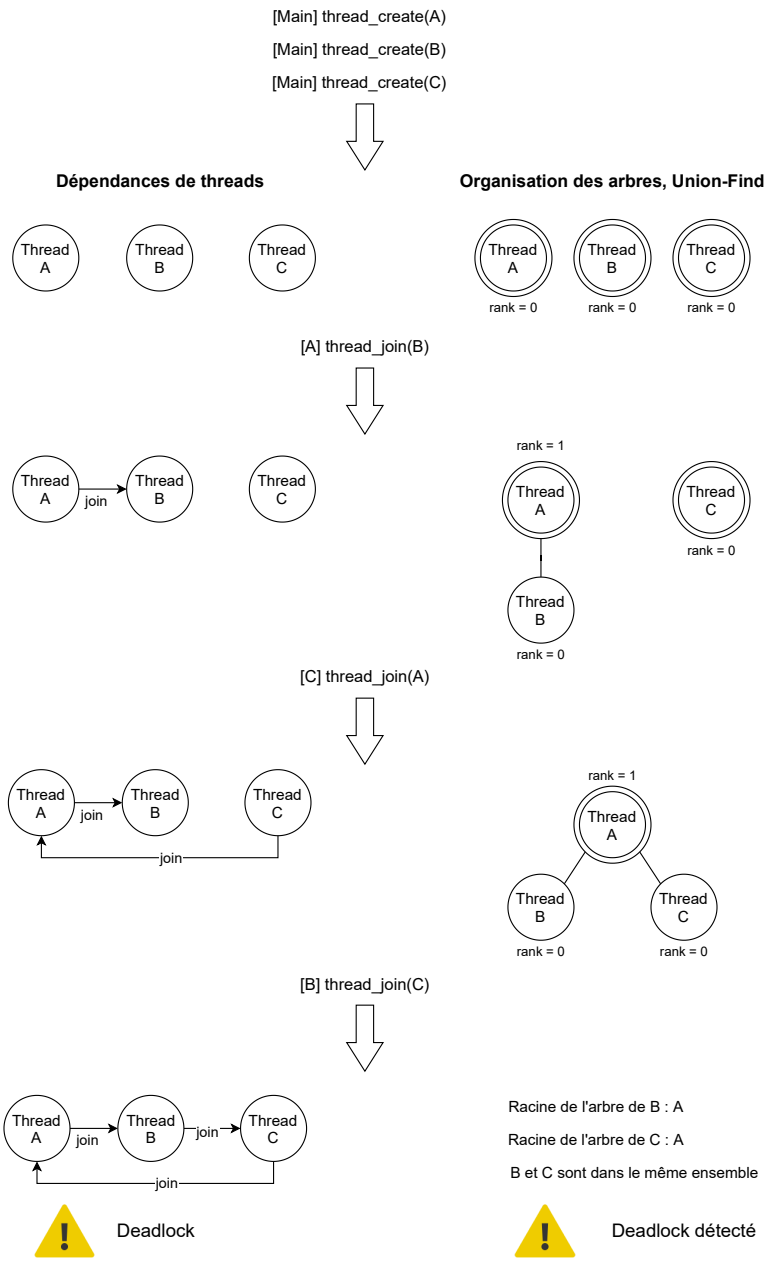

FIGURE 7: Exemple de détection d'un interblocage avec la structure de données « Union-Find »

La seconde fonction est l'`Union`. Elle permet de réunir deux partitions en une. La première étape est de trouver le représentant de chacune des partitions. On utilise leur rang pour savoir qui devient le parent de l'autre. Lorsque le rang est égal, on tranche arbitrairement et on augmente le rang de la nouvelle racine. Le rang qui permet d'estimer la taille d'un arbre, assure que lors du prochain `Find`, le nombre de nœuds qui doivent changer de parent sera minimal. La racine de l'arbre ayant le rang le moins élevé devient donc un fils de l'autre racine.

Pour détecter la présence d'un cycle entre deux partitions, nous modifions légèrement la fonction `Union`. Lorsqu'on recherche les deux racines des partitions, nous vérifions que les racines ne sont pas égales. Si elles le sont, alors l'opération va entraîner un interblocage. La Figure 7 résume les mécanismes amenant la détection d'un interblocage.

Les systèmes de rang et d'aplanissement de l'arbre apportés par l'utilisation de cette structure permettent de limiter la complexité amortie en temps à $O(\alpha(n))$, avec n le nombre de threads et α la fonction inverse d'Ackermann, comme montré dans [TL84]. Cette dernière est une fonction qui croît très lentement et qui correspond à une complexité inférieure à linéaire, quasiment constante. En effet, la réciproque de la fonction de Ackermann est un logarithme itéré. Les figures 8 et 9, permettent de visualiser l'impact de la détection d'interblocages sur les performances. On remarque que le temps d'exécution reste similaire entre la version séquentielle et la version qui détecte les interblocages. Nous avons bien atteint l'objectif de détecter tous les

interblocages possibles, tout en évitant de rendre `thread_join` linéaire en le nombre de threads de la chaîne de dépendance de threads.

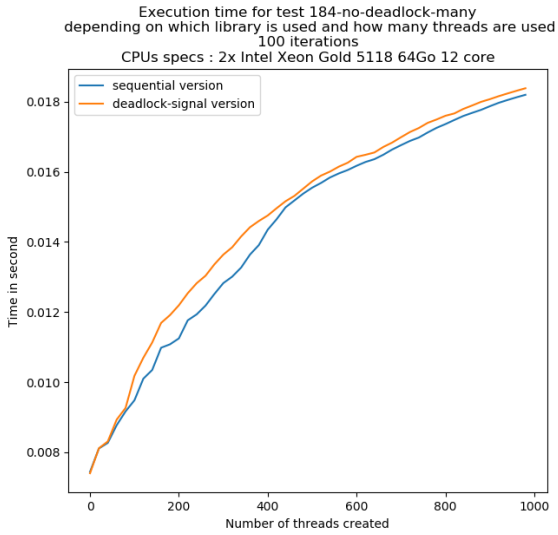

FIGURE 8: Mesure du temps pour réaliser une chaîne de threads avec la fonction `thread_join`, avec et sans détection d'interblocages, en fonction du nombre de threads

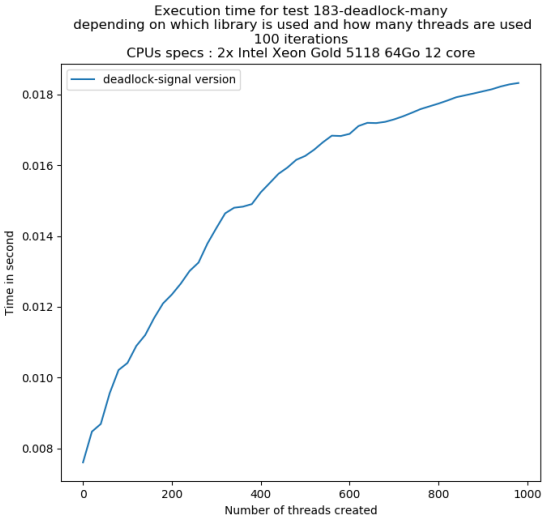

FIGURE 9: Mesure du temps pour réaliser une chaîne de threads avec la fonction `thread_join`, avec le dernier thread qui produit un interblocage

4.2 Gestion de la mémoire

Certes, nous sommes dorénavant capables de détecter des interblocages. Cependant, il est possible que le représentant d'une partition termine avant les enfants qui le référencent : il y a un risque d'utilisation de mémoire désallouée (Use-After-Free).

Pour empêcher cela, nous modifions la bibliothèque afin que chaque thread possède un attribut qui compte le nombre d'enfants qu'il possède. Cet attribut est mis à jour à chaque opération `Union`, car il n'y a en réalité que le nombre d'enfants de la racine qui est modifié. Un thread sera détruit seulement s'il ne possède plus d'enfant. Sinon, ce sera le dernier fils qui détruira son parent, nous assurant ainsi d'avoir libéré correctement la mémoire utilisé par les `threads` de cette partition. La Figure 10 résume le fonctionnement qui assure une bonne libération de la mémoire.

5 Modèle de threads hybride $M : N$

Enfin, la bibliothèque de threads a été retravaillée de manière à tirer partie des architectures multicœurs. Pour cela, nous implémentons une bibliothèque de threads « *many-to-many* » ($M : N$), dans laquelle plusieurs threads utilisateur s'exécutent en parallèle sur plusieurs threads système. Cette approche a pour objectifs de permettre l'exécution de plusieurs threads utilisateur en parallèle, tout en limitant le coût important de création et de changement de contexte des threads système.

Pour ce faire, au lancement d'un programme utilisant notre bibliothèque `libthread`, le thread principal du processus crée des threads système fils, à l'aide de la bibliothèque de threads système `pthread`. Les différents threads système se chargent alors d'exécuter en parallèle les threads utilisateur, créés par l'utilisateur de `libthread`.

Contrairement à `pthread`, la bibliothèque fournit à l'utilisateur une *couche d'abstraction* sur les threads système. En effet, le développeur d'un programme tiers ne crée que des threads utilisateur, dont les coûts de création et de destruction sont peu coûteux. `libthread` ordonnance alors les threads utilisateur sur les threads noyau, de manière opaque pour le programmeur. Cette approche permet de ne créer qu'un nombre fixe de threads noyau par exécution d'un programme, afin de limiter les coûts induits par les appels système utilisés par `pthread` pour les créer, les supprimer ou les préempter.

Ainsi, même lorsqu'un thread système n'a plus de thread utilisateur à exécuter, il reste en attente, au cas où un nouveau thread utilisateur serait disponible. Tout au long du programme, la bibliothèque conserve alors une *thread pool*, qui stocke l'ensemble des threads système créés, actifs ou non. À l'issue du programme,

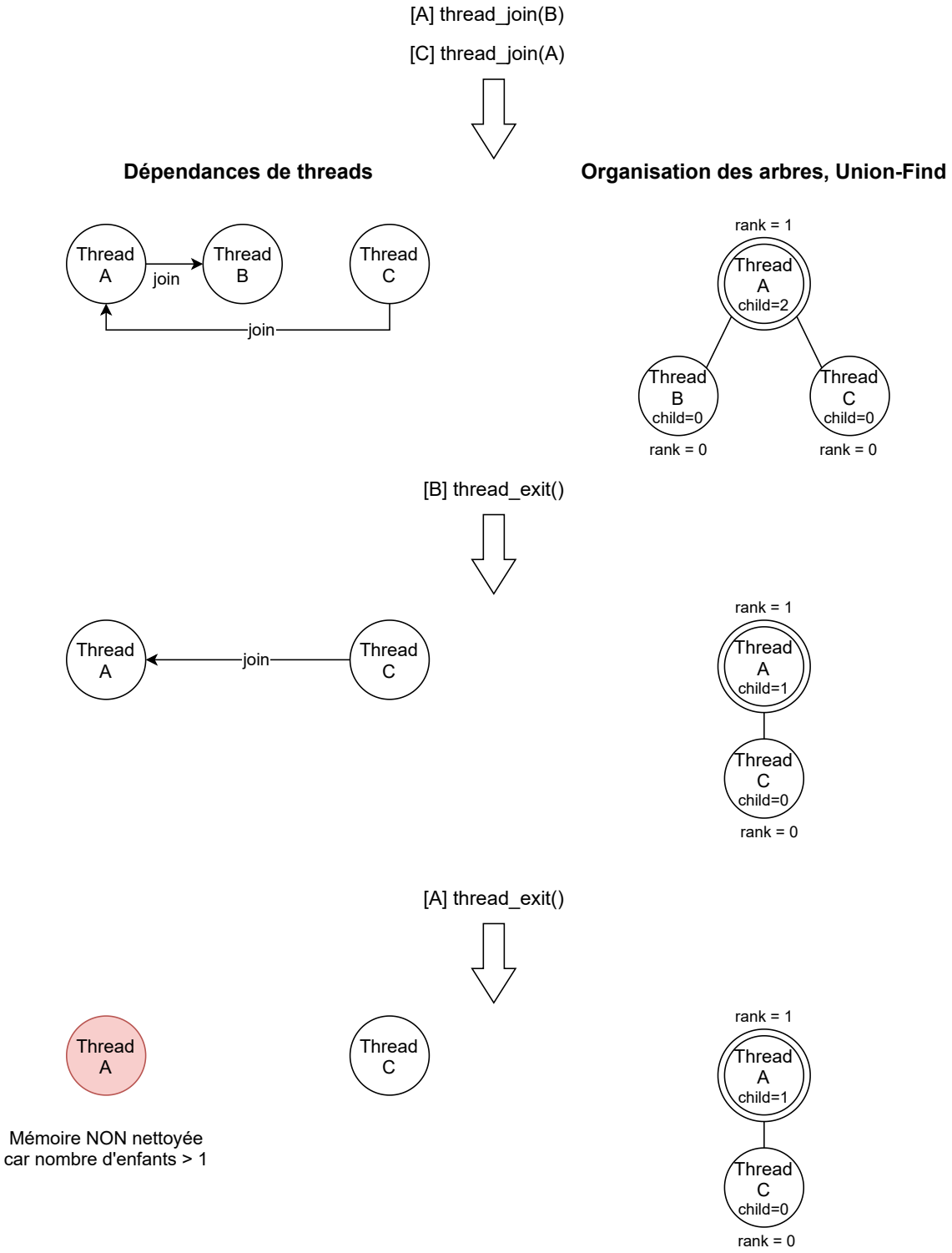

FIGURE 10: Exemple du fonctionnement si un représentant termine avant un de ses enfants.

lorsqu'aucun thread noyau n'a de thread utilisateur à exécuter, les threads noyau « fils » terminent et le thread noyau « initial » les attend, via des appels à `pthread_exit` et `pthread_join`, respectivement.

5.1 Verrouillage des ressources

Toutefois, la traduction d'une bibliothèque de threads séquentielle en parallèle ne se limite pas à exécuter le même code séquentiel sur différents threads noyau. En effet, il est impératif de s'assurer que l'exécution parallèle n'aboutit pas à des incohérences, notamment en terme de libération de la mémoire ou de gestion de la file d'exécution.

En particulier, certaines séquences d'instructions doivent s'exécuter de manière atomique, en empêchant qu'un autre thread noyau puisse accéder à un même emplacement mémoire concurremment. Par exemple, lorsqu'un thread noyau enlève un thread utilisateur de la file d'attente pour l'exécuter, les autres threads ne doivent pas pouvoir y insérer ou supprimer un autre thread en concurrence, auquel cas des erreurs pourraient se produire : deux threads système pourraient notamment récupérer le même thread utilisateur en tête de la file, puis supprimer les deux premiers éléments de la liste, comme montré en algorithme 2. Ainsi, le premier thread utilisateur de la file serait exécuté deux fois, et le deuxième serait supprimé de la file, sans jamais être exécuté. L'algorithme 1 correspond en revanche à une exécution valide.

Données : 2 threads noyau, `kth1` et `kth2`
[kth1] `kth1.threadCourant` ← `runq.tête()`
[kth1] `runq.défiler()`
[kth2] `kth2.threadCourant` ← `runq.tête()`
[kth2] `runq.défiler()`
[kth2] `kth2.exécuter(kth2.threadCourant)`
[kth1] `kth1.exécuter(kth1.threadCourant)`

Algorithme 1 : Exemple d'ordonnancement d'instructions valide lors de suppressions dans la file d'exécution

Données : 2 threads noyau, `kth1` et `kth2`
[kth1] `kth1.threadCourant` ← `runq.tête()`
[kth2] `kth2.threadCourant` ← `runq.tête()`
`/* 2 threads noyau vont exécuter le même thread utilisateur */`
[kth1] `runq.défiler()`
[kth2] `runq.défiler()`
`/* Le deuxième thread utilisateur est supprimé sans être exécuté */`
[kth2] `kth2.exécuter(kth2.threadCourant)`
[kth1] `kth1.exécuter(kth1.threadCourant)`

Algorithme 2 : Exemple d'ordonnancement d'instructions invalide lors de suppressions dans la file d'exécution

Pour empêcher les exécutions incorrectes, nous utilisons un système de verrous : un thread noyau peut acquérir, puis libérer un verrou, qui garantit que toute séquence d'instructions entre ces deux opérations ne peut être exécutée que par un thread à la fois. Ainsi, un autre thread qui souhaite acquérir un verrou est bloqué jusqu'à ce que la ressource soit disponible, ce qui garantit que les instructions protégées sont exécutées de manière *atomique* d'un point des vue des autres threads.

Dans la bibliothèque développée, les threads noyau doivent notamment acquérir un verrou afin de lire ou écrire de manière cohérente dans la file d'attente (`runq`). Pour ce faire, nous utilisons l'instruction processeur atomique `compare_and_swap`, telle que `compare_and_swap(r, old, new)` écrit la valeur `new` dans `r` si et seulement si `r` vaut actuellement `old`, et qui renvoie toujours l'ancienne valeur `old`.

Nous définissons alors l'acquisition d'un verrou par un appel à `compare_and_swap(verrou, 0, 1)` tant que la valeur retournée par l'instruction ne vaut pas `0` : en initialisant le verrou à `0`, même si plusieurs threads sont exécutés en parallèle, le processeur garantit que seul un thread pourra écrire la valeur `1` dans le registre `verrou`, et retournera `0`. Les autres threads ne parviendront pas à écrire `1` dans `verrou`, puisque le registre atomique ne vaudra plus `0` mais `1`, et l'instruction renverra la valeur `1`. De ce fait, seul le thread pour lequel l'instruction `compare_and_swap` renvoie `0` est autorisé à poursuivre pour exécuter sa section critique, tandis que les autres threads restent en attente active afin d'acquérir le verrou à l'aide de `compare_and_swap`. Lorsqu'un thread termine d'exécuter les instructions « critiques », il libère alors le verrou, en écrivant la valeur `0` pour permettre à un nouveau thread de prendre la main sur le verrou.

Cette implémentation permet ainsi de disposer d'un mécanisme de verrouillage fiable, sans subir de surcoût en terme de performances et de dépendre d'une bibliothèque de mutexes tierce. Afin de limiter les erreurs de manipulation lors du `compare_and_swap` et de factoriser le code, des fonctions de verrouillage et de déverrouillage sont néanmoins définies et utilisées par la bibliothèque, en utilisant de l'`inlining` pour éviter de générer des instructions d'appel de fonctions.

Au sein des fonctions de la bibliothèque, telles que `thread_create`, `thread_yield`, `thread_join` et `thread_exit`, ces verrous sont donc utilisés pour réaliser des opérations atomiques dans la file d'exécution ou sur un thread utilisateur.

5.2 Verrouillage de threads utilisateur

Même si l'implémentation générale de la bibliothèque reste similaire à la version séquentielle, certaines garanties doivent être vérifiées afin de permettre des exécutions valides. Ainsi, toute écriture dans la file d'exécution, ainsi que la majorité des lectures, doivent être réalisées de façon atomique.

Par ailleurs, lors de l'exécution du programme, tout thread noyau doit, soit exécuter un thread utilisateur lorsque cela est possible, soit attendre qu'un nouveau thread utilisateur soit ajouté à la file d'exécution pour le lancer. En effet, contrairement à la version séquentielle, lorsque le thread utilisateur courant termine et que la file d'exécution est vide, il ne s'agit pas nécessairement de la fin du programme : les autres threads peuvent toujours exécuter des threads utilisateur en parallèle, et potentiellement créer de nouveaux threads par la suite.

Dans la bibliothèque réalisée, chaque thread noyau exécute à sa création une fonction `kernel_thread_next_user_thread`, qui est chargée d'obtenir un nouveau thread utilisateur dès que cela est possible. Elle est également appelée, dans certains cas, lors d'un `thread_join` ou d'un `thread_exit`. De plus, un appel à cette fonction doit nécessairement avoir lieu dans un contexte statique propre au thread noyau, indépendamment des contextes des threads utilisateur. En effet, une fois qu'un thread utilisateur a terminé, ses ressources sont libérées par le thread utilisateur qui l'attend, potentiellement exécuté dans un autre thread noyau. Dans un tel cas, le contexte d'exécution serait supprimé par un autre thread noyau, tout en étant toujours utilisé par le thread noyau qui attend un nouveau thread utilisateur, ce qui provoquerait des erreurs mémoire.

De même, la terminaison d'un thread utilisateur doit être *atomique* d'un point de vue du thread qui l'attend. Dans le cas contraire, un thread utilisateur m qui attend un thread n pourrait libérer la mémoire de n, alors même que m n'a pas fini complètement son appel à `thread_exit` et n'a pas pu basculer dans le contexte d'attente de nouveau thread (`kernel_thread_next_user_thread`). Par conséquent, pour éviter des incohérences lors d'un `thread_join` et d'un `thread_exit` concurrents sur le même thread, celui-ci est verrouillé à l'aide d'un verrou présent dans la structure `thread_entry`. Cela permet de s'assurer que les opérations sensibles d'un `thread_join` ou d'un `thread_exit` sur un thread sont réalisées séquentiellement, et non concurremment. En particulier, lors d'un `thread_exit` ou d'un `thread_join`, les verrous acquis ne doivent être libérés qu'*après* changement de contexte. Par exemple, lorsqu'un thread utilisateur termine, il faut nécessairement libérer son verrou après avoir changé de contexte, sinon le contexte actuel du thread utilisateur pourrait être libéré alors même que ce contexte est encore exécuté par un thread noyau. La Figure 11 illustre une telle exécution.

5.3 Répartition des threads en attente et fin de programme

Lorsqu'un thread termine ou devient en attente d'un événement, il cède alors la main à la fonction `kernel_thread_next_user_thread`, dont le pseudo-code est présenté dans l'algorithme 3, afin de trouver un nouveau thread utilisateur à exécuter. Cette fonction est également chargée de détecter la fin du programme, lorsque plus aucun thread système n'a de thread utilisateur à exécuter. Ainsi, un compteur global `nbThreadsSysAttente` garde une trace du nombre de threads système qui n'ont pas réussi à obtenir un thread utilisateur à l'issue d'au moins une itération.

Pour récupérer un thread utilisateur, le thread système se contente d'attendre que le sémaphore `longueurRunq` soit positif et puisse le décrémenter, de manière atomique (l. 6). Ce sémaphore provient de la bibliothèque `semaphore.h`, qui fournit une telle implémentation : contrairement aux mutexes implémentés à l'aide de `compare_and_swap`, cette attente est *passive* : cela permet d'endormir le thread système tant que le sémaphore n'a pas une valeur strictement positive, et de le réveiller lorsque c'est le cas. Même si, en théorie, cette solution n'est pas aussi performante qu'une attente active avec un `compare_and_swap`, cela permet d'éviter une forte contention sur un registre atomique. D'autre part, lorsque le nombre de threads système est supérieur au nombre de cœurs, le système d'exploitation ne passe pas la main inutilement à des threads système en attente.

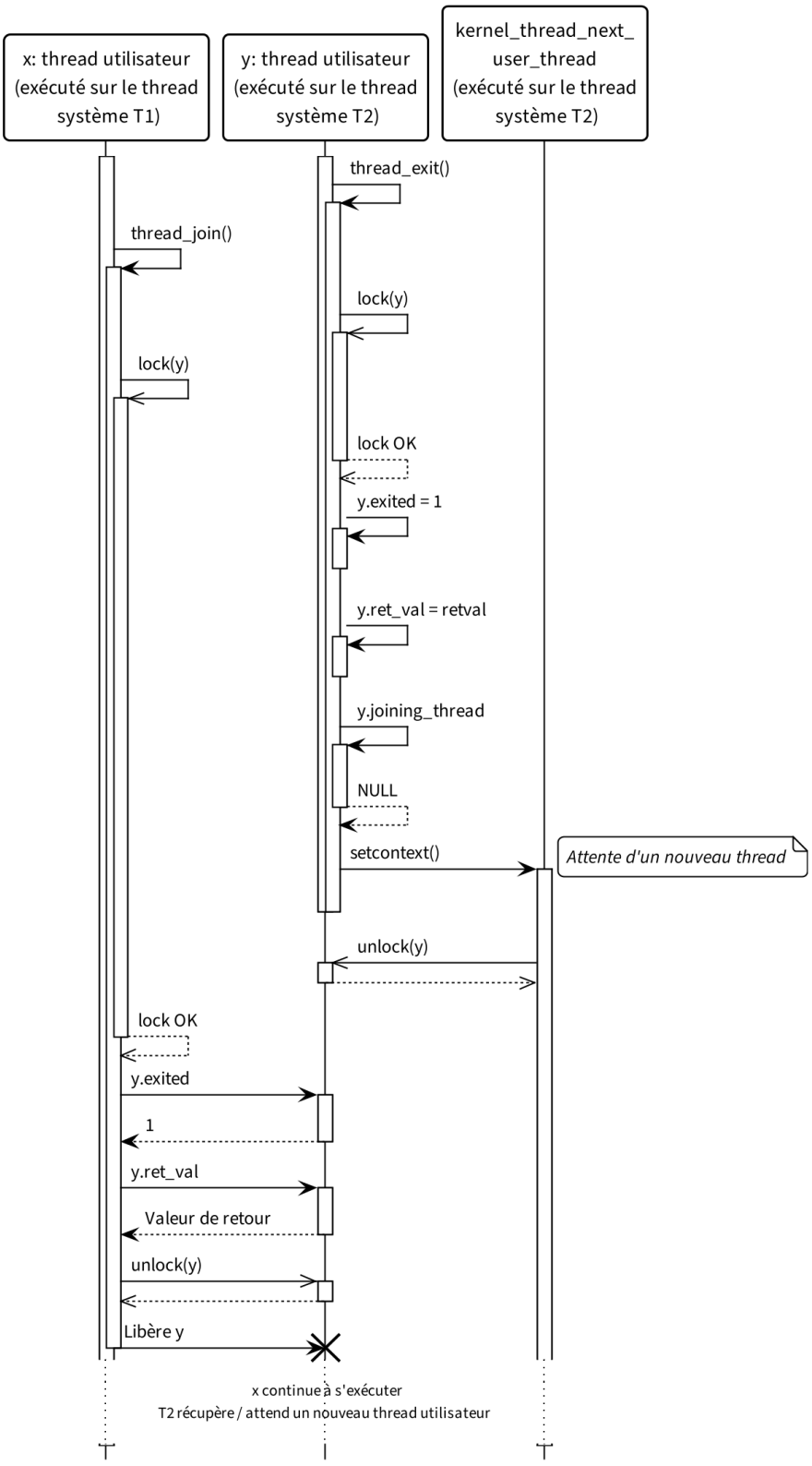

FIGURE 11: Diagramme de séquence de terminaison d'un thread y avec `thread_exit` *avant* son attente par x dans `thread_join` en parallèle

```
   threadCourant : Thread courant, local au thread
   runq : File d'attente de threads utilisateur
   verrouRunq : Verrou
   longueurRunq : Sémaphore
   nbThreadsSysAttente : Compteur protégé par verrou

   fonction kernel_thread_next_user_thread() :
       // ... Libération des verrous sur les threads acquis ...
1
2      timeout ← 1 ns
3      premiereIteration ← Vrai
4
5      tant que nbThreadsSysAttente.compteur < nbThreadsSys faire
6          si longueurRunq.attendre(timeout) ≠ TIMEOUT alors
7              // Un thread est disponible
8              verrouRunq.verrouiller ()
9              threadCourant ← runq.tête()
10             runq.défiler()
11             verrouRunq.déverrouiller ()
12
13             si non premiereIteration alors
14                 // Le thread a terminé son attente
15                 nbThreadsSysAttente.verrouiller ()
16                 nbThreadsSysAttente.compteur ← nbThreadsSysAttente.compteur − 1
17                 nbThreadsSysAttente.déverrouiller ()
18             threadCourant.exécuter () // Ne retourne pas
19         sinon
20             // Aucun thread disponible actuellement
21             si premiereIteration alors
22                 // On se déclare en attente
23                 nbThreadsSysAttente.verrouiller ()
24                 nbThreadsSysAttente.compteur ← nbThreadsSysAttente.compteur + 1
25                 nbThreadsSysAttente.déverrouiller ()
26                 premiereIteration ← Faux
27
28     // Plus de thread utilisateur dans la runq, tous les threads système en
          attente
29     synchroniserEtTerminer () // Ne retourne pas
```

Algorithme 3 : Gestion des threads système en attente et récupération des nouveaux threads utilisateur

Ici, le sémaphore est utilisé pour enregistrer le nombre de threads actuellement dans la file d'exécution. Lors d'un `thread_create`, le sémaphore est incrémenté de manière atomique, ce qui donne la possibilité à un thread système en attente de l'exécuter. En revanche, il n'est pas nécessaire de modifier le sémaphore lors d'un `thread_yield`, puisque le nombre de threads dans la file d'exécution ne varie pas.

Une fois que le thread système parvient à décrémenter le sémaphore, il attend activement que le verrou sur la `runq` soit disponible, et l'acquiert (l. 8). Il récupère alors le thread utilisateur en tête de la file, le supprime de la file et déverrouille la file (l. 9–11). Il décrémente alors le compteur de threads système en attente si nécessaire (l. 13–17), et exécute le contexte du thread obtenu (l. 18).

Néanmoins, l'attente sur le sémaphore doit être bornée dans le temps. En effet, si aucun thread utilisateur n'est disponible, il est impératif de vérifier que le nombre de threads système en attente est toujours inférieur au nombre de threads système total : dans le cas contraire, l'attente sur le sémaphore serait infinie, puisque tous les threads système seraient bloqués et attendraient une incrémentation du sémaphore. Pour ce faire, l'attente sur le sémaphore (l. 6) est limitée arbitrairement à 1 nanoseconde. Si le thread système n'est pas parvenu à décrémenter le sémaphore durant ce laps de temps, la méthode[2] `attendre` renvoie `TIMEOUT` et exécute le bloc `sinon` (l. 20–26). Le cas échéant, le thread système incrémente le nombre de threads système en attente, s'il ne l'a pas déjà fait.

Enfin, lorsque tous les threads système sont bloqués dans la boucle `tant que`, les threads système sont synchronisés, les threads « non-initiaux » terminent, et les ressources du dernier thread utilisateur sont libérées.

2. Pour des raisons de simplification, le pseudo-code est rédigé selon une syntaxe orientée objet, même si cela ne reflète pas l'implémentation réelle.

5.4 Performances obtenues

Finalement, notre implémentation du modèle hybride de threads dit « *many-to-many* » $(M : N)$ n'est pas plus performante dans l'ensemble des situations. Dans les graphiques suivants, nous comparons les performances de la version parallèle à celles de `pthread` et de la version séquentielle (sans détection de *deadlocks* et gestion de signaux).

Ainsi, la version séquentielle est plus rapide sur certains tests fournis par le sujet, notamment le test `21-create-many.c`, dont les performances sont illustrées en Figure 12. Ces différences s'expliquent par le fait que le test se contente de créer dans le thread principal, à chaque itération, un seul thread utilisateur, qu'il attend directement après. D'une part, les fonctions sont plus coûteuses en parallèle qu'en séquentiel, notamment à cause des sections critiques et des changements de contexte plus nombreux. D'autre part, le code de test est quasiment séquentiel, puisqu'il n'est possible d'exécuter que deux threads en parallèle, le thread principal et un thread fils. Or, le premier thread ne fait qu'attendre son fils, qui lui-même ne fait que quitter directement : même si ces deux threads peuvent s'exécuter en parallèle, les threads ne réalisent pas de suites d'instructions *coûteuse*, qui pourraient être réparties entre deux cœurs de la machine dans une version parallèle. Avec la bibliothèque `pthread`, cela est d'autant plus visible que chaque création, suppression, ou attente de thread conduit à un appel système, qui sont d'autant plus coûteux.

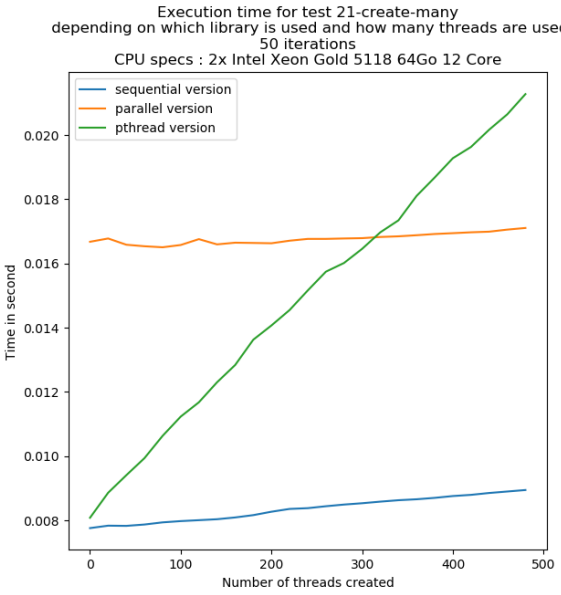

FIGURE 12: Comparaison du temps d'exécution du test `31-switch-many` entre `pthread`, `libthread` séquentiel et `libthread` parallèle

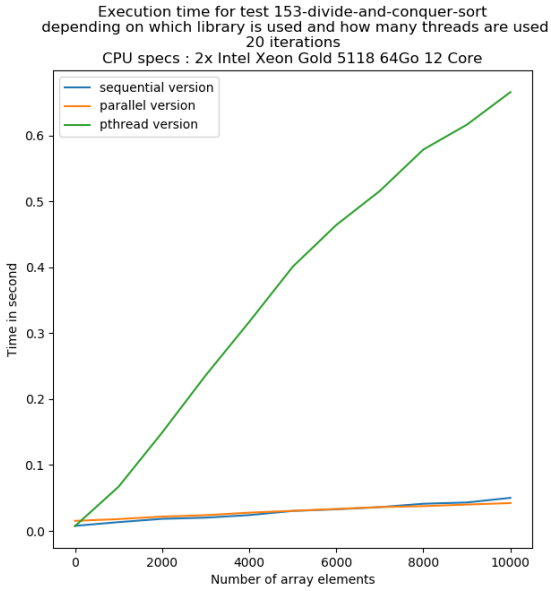

FIGURE 13: Comparaison du temps d'exécution d'un tri fusion entre `pthread`, `libthread` séquentiel et `libthread` parallèle

En revanche, sur des tests davantage parallélisables, le modèle de threads $N : M$ est plus performant que la bibliothèque séquentielle lorsque de nombreux threads utilisateur sont créés. Nous avons ainsi implémenté des tests de tri fusion et de somme d'entiers d'un tableau en diviser-pour-régner, pour lesquels chaque partie de tableau est gérée par un nouveau thread. La Figure 13 présente le temps d'exécution du tri fusion d'un tableau de valeurs aléatoires, selon la taille du tableau. Même si la version séquentielle est plus performante sur des tableaux de petite taille en raison du coût de création des 24 threads système de la version parallèle, qui n'est pas amorti par le faible nombre d'instructions réalisées en parallèle, le modèle hybride devient plus rapide pour des grandes tailles de tableaux. Ainsi, pour un tri d'un tableau de taille 10001 moyenné sur 20 exécutions sur une machine possédant 2 processeurs Intel Xeon Gold 5118 64Go de 12 cœurs physiques chacun, le modèle de threads hybride avec 24 threads noyau a un facteur d'accélération par rapport à la version séquentielle de :

$$\frac{\Delta t_{\text{seq}}}{\Delta t_{\text{par}}} \approx \frac{0,05016}{0,04222} = 1,188$$

La bibliothèque `pthread` est quant à elle fortement pénalisée par le coût de création, de suppression, d'attente de nombreux threads système, ainsi que le changement de contexte de threads au niveau du système d'exploitation : même si le temps d'exécution reste linéaire en fonction du nombre de threads, le coefficient directeur est bien plus important.

Conclusion

À l'issue du projet, l'ensemble des fonctions de base de la bibliothèque séquentielle, les fonctions d'envoi et réception de signaux, ainsi que la détection des interblocages 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.

Par ailleurs, comme anticipé lors du rapport intermédiaire, la version de la bibliothèque parallèle $M : N$ est parfois moins performante que la version séquentielle, notamment lorsque le code de l'utilisateur n'est pas suffisamment parallélisable. Par ailleurs, nous n'avons pas souhaité intégrer la détection d'interblocages dans cette version parallèle, puisque cela aurait nécessité de verrouiller de nombreux threads à cause de la structure `Union-Find`, limitant d'autant plus nos performances, tout en risquant des interblocages entre les différents verrous. Par manque de temps, nous n'avons pas non plus pu intégrer les signaux à la version parallèle.

Enfin, même si la version parallèle a été testée sur un certain nombre d'exemples, de valeurs et de configurations, le caractère non-déterministe des exécutions rend la détection d'erreurs complexe. Il est donc encore probable que de nombreux bogues, entraînant des erreurs mémoire ou des interblocages, subsistent dans la bibliothèque actuelle.

Références

[TL84] R. TARJAN et J. V. LEEUWEN. « Worst-case Analysis of Set Union Algorithms ». In : *J. ACM* 31 (1984), p. 245-281.