Système d'exploitation

Rapport final du projet de Système d'exploitation : Threads en espace utilisateur

Professeurs encadrants : **Philippe SWARTVAGHER**, **Mathieu FAVERGE**

Sidi ABDEL MALICK
Zineb BAROUDI
Juliette DEGUILLAUME
Kaïs-Khan HADI
Lucas TROCHERIE

ENSEIRB-MATMECA – Bordeaux INP
Talence, France

17 mai 2020

1 Introduction

Ce rapport a pour principal but de présenter l'avancement final du projet de système d'exploitation du groupe 9116.
Le projet vise à construire une bibliothèque de gestion de **threads**, pouvant remplacer une partie de la bibliothèque existante **pthread**. Un des intérêts de ce sujet est de voir les différences notables de performances entre la bibliothèque `pthread` et celle visant à la remplacer en partie, développée par le groupe, ainsi que les impacts des ajouts à la bibliothèque sur ses performances. De plus, ce projet est évidemment intéressant d'un point de vue culturel et apprentissage pour chaque membre de l'équipe.

L'organisation de ce projet s'est faite à l'aide de l'outil *Trello*, qui permet de découper le projet en tâches, répartir ces tâches à des membres de l'équipe mais aussi de suivre l'avancement de celles-ci. A cet outil, s'ajoutent des réunions vocales hebdomadaires, sur le serveur *Discord* de l'équipe, où l'avancement de chaque membre et l'avancement global dans le projet sont étudiés. Ces réunions permettent également de remplir le tableau *Trello* avec les affectations des membres à des tâches ainsi que l'ajout de nouveaux objectifs pour la réussite du projet.

2 Implémentation

Notre implémentation est passée par différentes étapes et différentes architectures avant d'arriver à une solution satisfaisante, c'est-à-dire qui permette d'avoir un comportement similaire à la bibliothèque `pthread` et qui empêche toute fuite mémoire.

Cette partie du rapport présentera le fonctionnement de notre bibliothèque, mais également des optimisations ajoutées et des problèmes rencontrés au cours du projet, ainsi qu'une analyse de cette bibliothèque.

2.1 Fonctionnement

La bibliothèque de `threads` s'articule autour d'une structure `thread` contenant toutes les informations nécessaires au bon fonctionnement de celle-ci :

```
struct thread{
  int id; // thread identifier
  int valgrind_stackid; // help for valgrind for the stack's size and pointer
  ucontext_t *context; // thread's context
  void *retval; // return value of the thread
  int is_finished; // boolean giving whether a thread has finished or not
  STAILQ_ENTRY(thread) next; // pointer toward the next thread in the queue
  STAILQ_HEAD(thread_waiting_list, thread) thread_waiting_list; // waiting list
};
```

Le type `thread_t` fourni dans l'énoncé correspond ici au type `struct thread*`. L'interface manipule une file, simplement chaînée, contenant des `thread_t`. Cette file sera appelée ici `runqueue`.

La file est une **Queue BSD** et a été choisie simplement chaînée. Cela permet d'avoir une manipulation simple de liste et cela a été un choix quel que peu par défaut, à la base, de par la richesse d'implémentations que proposent les `Queue BSD`. En avançant dans le projet, nous avons décidé de rester sur ce choix, car ajouter une file plus complexe n'aurait pas apporté de plus-value au projet, au regard de l'utilisation que notre bibliothèque fait de cette liste. Cependant, nous sommes passés d'une file simplement chaînée **SIMPLEQ** à une **STAILQ** pour un léger gain de performances sur l'insertion de `threads` à la fin de la `runqueue`.

L'interface est constituée d'un constructeur et d'un destructeur de manière à avoir le contrôle sur le code exécuté, et notamment sur le `main` de l'utilisateur de la bibliothèque. Le constructeur initialise la `runqueue` et *push* le `main` utilisateur dedans, dont le contexte a été récupéré grâce à la fonction `getcontext()`.
Le `main` utilisateur est alors exécuté grâce à un appel à `swapcontext(oldctxt,newctxt)`. Les paramètres `argc` et `argv` sont enregistrés de manière à garder le comportement attendu par l'utilisateur.

2.1.1 Les fonctionnalités `pthread` de base

Cette partie présente les fonctionnalités essentielles du projet. Elles se traduisent en cinq fonctions issues de la bibliothèque `pthread`, que nous avons implémentées dans notre bibliothèque pour obtenir une version basique.

La fonction `thread_create()` est relativement simple. Celle-ci récupère le contexte au moment où elle est appelée et crée, à partir de celui-ci, un nouveau contexte en prenant soin d'allouer une nouvelle `stack` et de stocker son pointeur et sa taille dans la table de **Valgrind**. Une nouvelle structure `thread` est ainsi instanciée et celle-ci est ajoutée à la `runqueue`. La fonction réalise ensuite un `swapcontext(oldctxt, newctxt)` et exécute la fonction appelée sur le nouveau `thread`.

Des fonctions auxiliaires sont utilisées pour la création de la nouvelle instance et pour son exécution. Il a été défini dans le sujet que la fonction d'un `thread` ne pouvait être appelée qu'avec au plus un paramètre. Une exception a du être faite pour le `main`, nécessitant deux paramètres. C'est pour cela que les variables `argc` et `argv` sont stockées en variables globales. Cette exception se trouve dans la fonction auxiliaire `thread_exec(...)`.

Ensuite, la fonction `thread_self()` est une fonction très simple à comprendre. Elle récupère le `thread` en tête de la `runqueue` et le retourne. Par définition, dans notre implémentation, le `thread` actuellement en train de s'exécuter est le `thread` en tête de file. Ceci est important pour la fonction suivante.

La fonction suivante est la fonction `thread_yield()`. Celle-ci récupère le `thread` en tête de la `runqueue`, comme `thread_self()`, mais cette fois en l'y retirant. Elle le place alors à la fin de la `runqueue`. Elle récupère ensuite le nouveau `thread` en tête de la `runqueue`, et lui passe la main à l'aide d'un `swapcontext(...)`.

En complémentarité de la fonction `thread_yield()`, la fonction `thread_join(...)` a été écrite. Celle-ci permet à un `thread` d'attendre la fin de l'exécution d'un autre `thread`. C'est ici que le booléen `is_finished` présent dans la structure `thread` prend tout son sens.

Si le `thread` passé en paramètre a déjà son booléen `is_finished` passé à `true`, le `thread` à l'origine du `thread_join(...)` continue son exécution normale. Sinon, ce dernier est sorti de la `runqueue` et est placé dans la `thread_waiting_list` du `thread` attendu. Une fois qu'un `thread` a terminé, tous les `threads` présents dans sa `thread_waiting_list` sont replacés dans la `runqueue` (cela est réalisé dans `thread_exit`).

Un `thread` qui attendait reprend son exécution lorsqu'il est à nouveau en tête de la `runqueue`, ce qui peut arriver bien plus tard que le moment où le `thread` qu'il attendait a terminé, et récupère la valeur de retour du celui-ci. Au passage, le stockage de `struct thread*` dans la `runqueue` au lieu de `struct thread` permet d'accéder directement aux informations importantes sans avoir à parcourir la file. Cela est lié à l'implémentation de `thread_exit()`, qui, lorsqu'un `thread` a terminé, retire celui-ci de la `runqueue` et le place ailleurs, comme cela est expliqué par la suite.

Enfin, pour terminer l'interface, la fonction `thread_exit(...)` a été implémentée. La fonction récupère le `thread` s'exécutant, i.e le `thread` en tête de file. Les attributs de ce `thread` sont ensuite complétés, cela inclut la valeur de retour `retval` et le booléen `is_finished`. Comme expliqué précédemment, tous les `threads` qui l'attendaient, donc présents dans sa `thread_waiting_list`, sont replacés dans la `runqueue`. La fonction le retire ensuite de la `runqueue`, et l'empile sur une file appelée `thread_deadstack`. Elle récupère ensuite le nouveau `thread` en tête de la `runqueue` et lui donne la main grâce à un appel à `setcontext(newctxt)`.

La fonction `thread_exit` ne retourne jamais. Il y a une exception à cette fonction. Celle-ci arrive lorsque le `thread` qui l'appelle est le dernier `thread` dans la `runqueue`, auquel cas son pointeur est stocké dans une variable globale, et la fonction réalise le dernier `return` de l'exécutable.

Beaucoup de zones mémoires ont été allouées dans les fonctions précédentes mais aucune n'a été libérée. Un destructeur a donc été ajouté ; celui-ci libère la zone mémoire du `thread` restant et de tous les `threads` présents sur la `thread_deadstack`.

2.1.2 Les mutex

Suite à la réalisation des fonctionnalités de base de la bibliothèque, fournissant une première version, l'équipe s'est penchée sur des objectifs avancés du sujet. Nous en avons implémenté plusieurs, dont celui correspondant à l'ajout des fonctions de synchronisation de type `mutex` au projet, présenté dans cette partie.

Comme pour les `threads`, la gestion des `mutex` dans la bibliothèque se réalise autour d'une structure `mutex`. Cependant, aucune structure de données supplémentaire n'est présente cette fois. Toute la gestion des `mutex` s'effectue directement avec la `runqueue`.

```
1 struct thread_mutex {
2     int is_initialized; // boolean to know if the mutex has been initialized
3     thread_t thread_locker; // thread who currently holds the mutex
4     STAILQ_HEAD(mutex_waiting_list, thread) mutex_waiting_list; // threads waiting
       to use the mutex
5 };
```

La fonction `thread_mutex_init(...)` s'occupe simplement de l'initialisation d'un `thread`. Elle initialise la `thread_waiting_list`, déclare le détenteur du `mutex` à *NULL* et valide l'initialisation du `mutex`.

Ensuite, en complément, la fonction `thread_mutex_destroy(...)` détruit un `mutex`. Comme aucune zone mémoire n'est allouée dans cette implémentation, cette fonction remet simplement l'initialisation du `mutex` comme non-valide.

Les fonctions suivantes réalisent les fonctions de verrouillage et déverrouillage présentes dans la bibliothèque `pthread`.

La première fonction est `thread_mutex_lock()`. Dans le cas où cette fonction est appelée et qu'aucun autre `thread` n'a la main sur le `mutex`, la fonction définit uniquement le `thread` appelant comme détenteur du `mutex` à l'aide de la fonction **`__sync_lock_test_and_set`**. Si le `mutex` est déjà utilisé, alors le détenteur du `mutex` ne change pas et le `thread` appelant est ajouté dans la `mutex_waiting_list`.

La deuxième fonction est `thread_mutex_unlock`. Deux cas de figure se présentent. Soit le `mutex` attend d'être utilisé par d'autres `threads`, soit personne ne l'attend. Dans le premier cas, la fonction récupère le `thread` en tête de sa `thread_waiting_list`, autrement dit le dernier

`thread` à avoir demandé un *lock* sur ce `mutex`, et le place dans la `runqueue`. Sinon, le détenteur du `mutex` est simplement déclaré à *NULL*.

2.1.3 La préemption

La préemption est un des objectifs avancés proposés dans le sujet qui a été implémenté dans la version finale du projet. Elle permet également de toucher à un autre objectif avancé qui concerne les signaux.

Le fonctionnement de cet ajout s'articule en deux parties. Tout d'abord, **un timer de 100ms**, qui est le temps d'exécution par défaut sous **Linux**, a été ajouté en utilisant la structure `itimerval`. Ce dernier est initialisé dans les fonctions `thread_exec(...)` et `thread_yield()`, ce qui permet de mesurer le temps d'exécution de n'importe quel `thread` à sa première exécution, mais aussi à chaque fois qu'il est potentiellement relancé suite à un `thread_yield()`. Ainsi, un `thread` ayant terminé son exécution avant la fin de *timer*, et passant la main à un autre `thread`, ne posera pas de problème, car le *timer* sera remis à 0 dans `thread_yield()`. Le *timer* utilisé **envoie un signal** `SIGALRM` une fois son temps écoulé.

Dans un deuxième temps, il a donc été nécessaire de **traiter ce signal** à sa réception pour implémenter la préemption. Pour cela, deux fonctions ont été ajoutées :

- `setting_handler()`, qui utilise la structure `sigaction` pour changer l'action à réaliser par un processus suite à la réception d'un signal spécifique (ici `SIGALRM`).
- `alarm_handler(int signum)`, qui est l'action à réaliser à la réception du signal `SIGALRM`. La réception de ce signal signifie que le temps d'exécution autorisé est écoulé. Ainsi, l'action à réaliser est de passer la main à un autre `thread` en appelant `thread_yield()`.

2.1.4 Les tests

Pour tester notre interface, de nombreux tests nous ont été fournis, nous permettant de valider notre implémentation dans un panel varié de scénarios proposés.

Nous avons également pris la liberté d'ajouter quelques tests. Ils sont disponibles dans le dossier `custom_tests/` et nous ont permis, tout au long du projet, de valider par nous-mêmes certains aspects spécifiques de l'implémentation. C'est par exemple le cas du test `31-preemption.c`, qui créé des `threads` dont l'exécution est volontairement longue, afin de tester que la préemption dans notre bibliothèque fonctionne correctement (c'est-à-dire vérifier si elle passe la main à un autre `thread` si un de ceux-ci dépasse son temps d'exécution autorisé). Quant à lui, le test `21-large-array-sum.c` somme tous les éléments de très grands tableaux en utilisant des `threads` et des `mutex` pour accélérer le calcul.

2.2 Optimisations

Après avoir eu une première version de bibliothèque stable, suite à l'implémentation des fonctionnalités de base imitant la bibliothèque `pthread`, l'équipe s'est penchée sur des optimisations possibles, permettant d'améliorer les performances de notre bibliothèque.

Ainsi, nous avons effectué deux optimisations, qui sont liées, dans les fonctions `thread_join()` et `thread_mutex_lock(...)`. En effet, initialement dans notre implémentation de `thread_join()`, tant que le `thread` attendu n'avait pas terminé son exécution, la bibliothèque appelait la fonction `thread_yield()` en boucle. Autrement dit, un `thread` qui attendait réalisait de l'attente semi-passive, le `thread` reprenait de temps en temps la main sur le processeur avant de la redonner aussitôt si le `thread` attendu n'avait toujours pas terminé. Cela entraînait le parcours de tous les `threads` (qu'ils soient prêts à être exécutés ou non, car dans l'attente d'un autre `thread`).

Pour éviter cette attente impactant les performances, nous avons ajouté un système de file d'attente pour les `threads`, avec la `Queue BSD` simplement chaînée `thread_waiting_list` (le choix de ce type de liste s'est fait de la même manière que pour la file `runqueue`). Ainsi, cela a permis de séparer les `threads` prêts à être exécutés des `threads` non exécutables, qui sont dans l'attente d'un autre `thread`. Cela a entraîné l'évitement du parcours inutile de certains `threads` avec une boucle sur des appels à `thread_yield()` et donc un gain de performances, présenté dans la partie 2.4.2.

Cet ajout de file d'attente a simplifié la fonction `thread_join()`, dont le fonctionnement final est expliqué dans la partie 2.1.1. Le principe a été le même pour l'optimisation dans la fonction `thread_mutex_lock(...)`, dont le fonctionnement final est présenté dans la partie 2.1.2.

Enfin, nous avons effectué une optimisation sur le type des files simplement chaînées utilisées dans la bibliothèque. Nous sommes passés d'une file simplement chaînée `SIMPLEQ` à une `STAILQ` pour un léger gain de performances sur l'insertion de `threads` en fin de `runqueue`.

2.3 Problèmes rencontrés

Plusieurs problèmes ont été rencontrés lors de l'écriture de notre bibliothèque de `threads`. De nombreux `Segfault` ont été rencontrés, comme certains *crashs*, et ceux-ci ont ralenti le développement de l'interface, nécessitant plusieurs heures de *débuggage* pour résoudre ces problèmes.

Des problèmes plus mystérieux ont également été rencontrés, comme un nombre d'allocations mémoire très important réalisées par le programme, pour des raisons encore difficilement compréhensibles aujourd'hui, dans le test **ficonnacci.c**. Ces allocations mémoires dépassaient largement les capacités de la `RAM` (pourtant de 8 GB), entraînant des *swaps* mémoires continus avec la mémoire de stockage de l'ordinateur, le rendant inutilisable.

De plus, une mauvaise gestion des contextes au début du projet a causé beaucoup de problèmes sous-jacents. Ces problèmes ont été résolus en repensant notre solution algorithmique dans son ensemble, ce qui a été réalisé plusieurs fois dans ce projet.

2.4 Analyse de la bibliothèque

Pour tester la bibliothèque, nous possédons une recette `Makefile` permettant de visualiser, avec des graphes, les performances de notre implémentation. Lancer cette règle sur les tests fournis par l'énoncé nous donne une bonne indication de la qualité de nos algorithmes. Les courbes ont toutes été tracées dans les mêmes conditions sur la même machine à des niveaux différents de l'avancement du projet (voir les différentes sections).

2.4.1 Analyse générale

Ces premières figures présentent les performances de notre bibliothèque de `threads` à la date du rendu du rapport intermédiaire. Des optimisations et améliorations ont été réalisées par la suite et sont présentées dans les sous-parties suivantes.

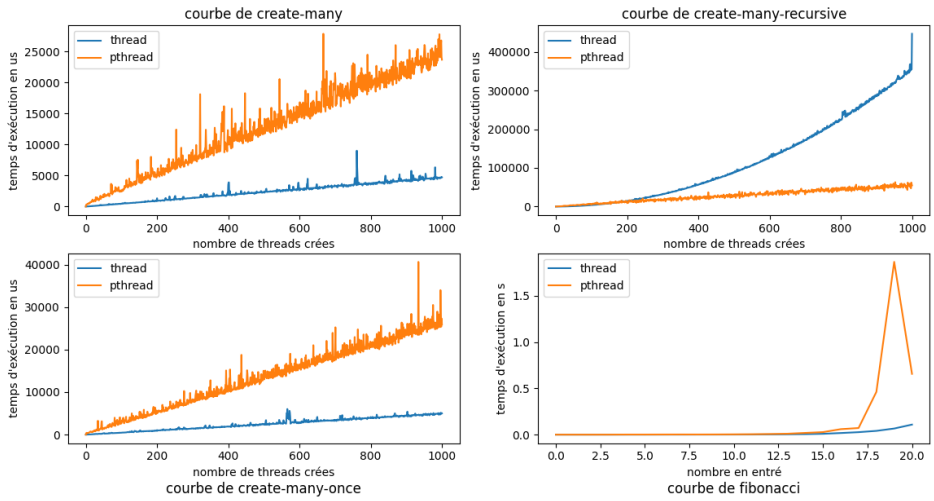

FIGURE 1 – Version intermédiaire : Tests "Create Many" et Fibonacci

Pour les tests présentés dans la figure 1, les performances de notre implémentation sont, de manière générale, meilleures que la bibliothèque `pthread`.
Les résultats pour `create_many`, `create_many_once` et `fibonacci` peuvent s'expliquer par le fait que la création et la destruction de `threads` sont beaucoup moins complexes dans notre implémentation que dans la bibliothèque `pthread`.

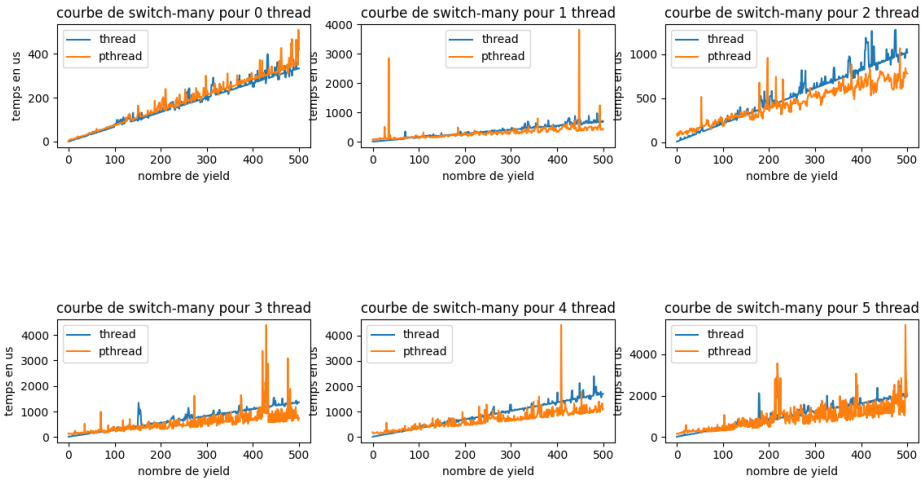

FIGURE 2 – Version intermédiaire : Tests "Switch-Many"

Nous obtenons d'aussi bons résultats que la bibliothèque `pthread` sur la batterie de tests présentés dans la figure 2, ci-dessus. Cela montre qu'au niveau d'un `yield` récursif de `threads`, notre bibliothèque a d'aussi bonnes performances.

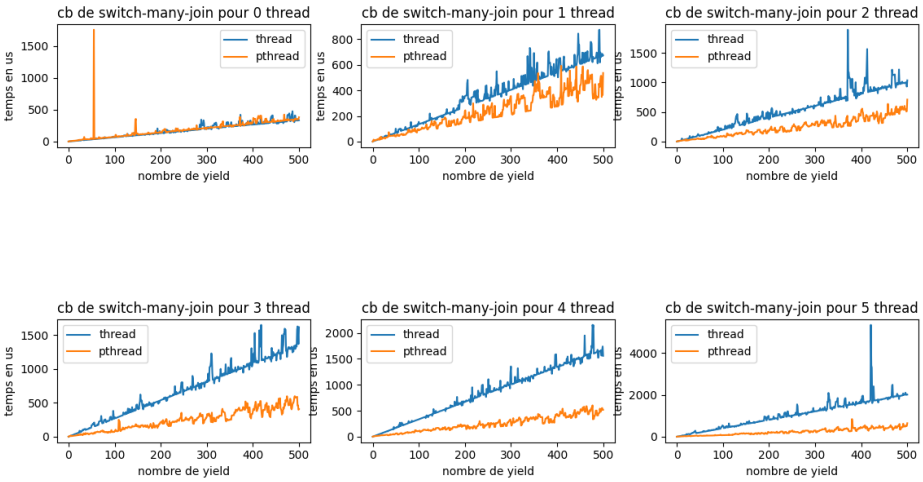

FIGURE 3 – Version intermédiaire : Tests "Switch-Many-Join"

Pour les tests de `switch-many` avec un `join` (présentés en figure 3), les résultats ne sont pas aussi positifs. Notre bibliothèque était plus lente (au moment du rendu intermédiaire) dans tous les scénarios que `pthread`, ce qui est lié à des permutations circulaires coûteuses de la `runqueue`

dues aux premières versions des fonctions `thread_join()` et `thread_yield()`, où les `threads` en attente sont laissés dans la `runqueue`.

D'une manière générale, la bibliothèque `pthread` est une bibliothèque beaucoup plus lourde et complète que l'interface que nous avons implémentée. Ce résultat est bien sûr attendu. De l'analyse que nous avons fait de `pthread`, cette bibliothèque est située très bas niveau, bien plus bas que notre code. De plus, elle réalise beaucoup de vérifications sur sa mémoire, afin de vérifier l'intégrité des informations qu'elle possède. Elle utilise également les méthodes `lll_lock` et `lll_unlock` qui manipulent des `futex` (Fast Userspace Mutex) dans le même but.

De plus, la bibliothèque `pthread` a été conçue pour être portable, et on trouve dans son code source des conditions et environnements qui changent si la bibliothèque est utilisée par différentes versions de la **libc** ou encore si la bibliothèque est utilisée par du **C++**.

On trouve aussi une différence entre `pthread` et notre bibliothèque, qui est que l'on l'utilise notre bibliothèque dans l'espace utilisateur, contrairement à `pthread` qui crée des `threads` au niveau du noyau, ce qui est plus coûteux, et qui utilise des appels système pour arriver à ses fins.

Ces éléments rendent, de manière générale, son exécution inévitablement plus lente que notre interface, mais la rendent fort heureusement plus sûre d'utilisation et plus fiable une fois en production.

2.4.2 Analyse des optimisations

Dans cette partie de l'analyse, la version de la bibliothèque prise en compte est celles où les différentes optimisations présentées dans la partie 2.2 sont implémentées.

L'attente semi-passive a donc disparu. Un `thread` qui réalise un `thread_join()` est sorti de la `runqueue` et est placé dans la file d'attente du `thread` attendu. De plus, la bibliothèque de `threads` est passée de l'utilisation de `SIMPLEQ`, file simplement chaînée avec uniquement un pointeur vers la tête de la file, à une utilisation de `STAILQ`. La seule différence est l'usage, dans la `STAILQ`, d'un pointeur supplémentaire vers la queue de la file, ce qui permet d'économiser un parcours de la `runqueue` lors de l'insertion d'un `thread` à la fin de celle-ci, comme c'est le cas dans de nombreuses fonctions de notre bibliothèque.

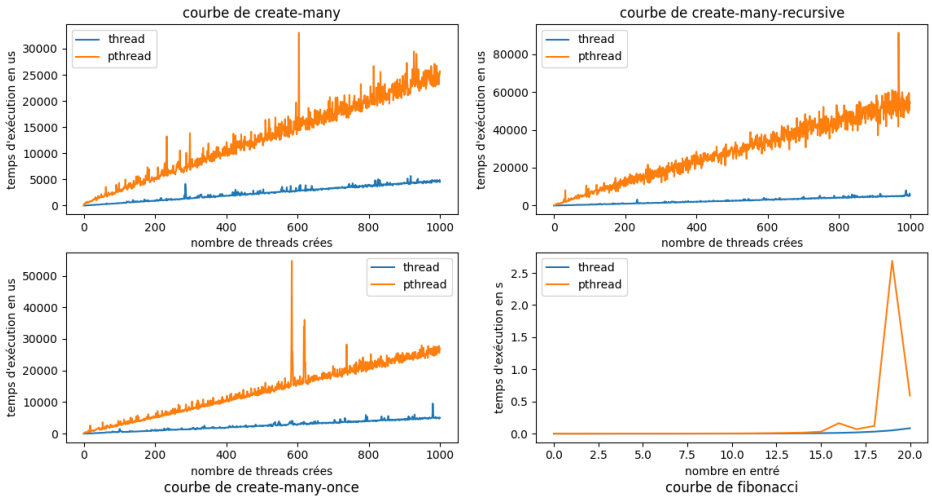

FIGURE 4 – Version avec optimisations : Tests "Create Many" et Fibonacci

Comme cela est visible sur la figure 4, les performances continuent à être meilleures que la bibliothèque `pthread`, mais cette fois dans tous les scénarios, **create-many-recursive** inclus. Cette optimisation a permis de gagner beaucoup de temps à l'exécution.

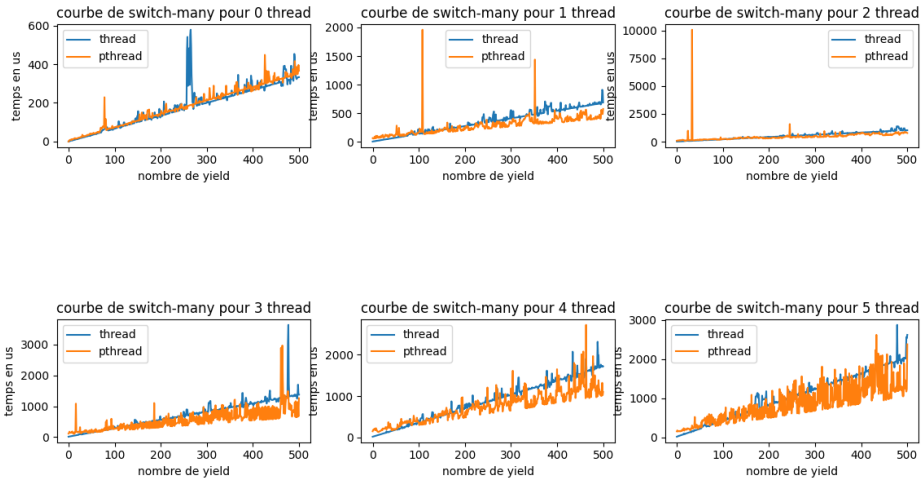

FIGURE 5 – Version avec optimisations : Tests "Switch-Many"

Cependant, cette optimisation n'a pas eu d'impact important sur le temps d'exécution du test **switch-many**, visible sur la figure 5. Ce dernier réalise peu d'appels à `thread_join`, ce qui explique cette absence de différence significative.

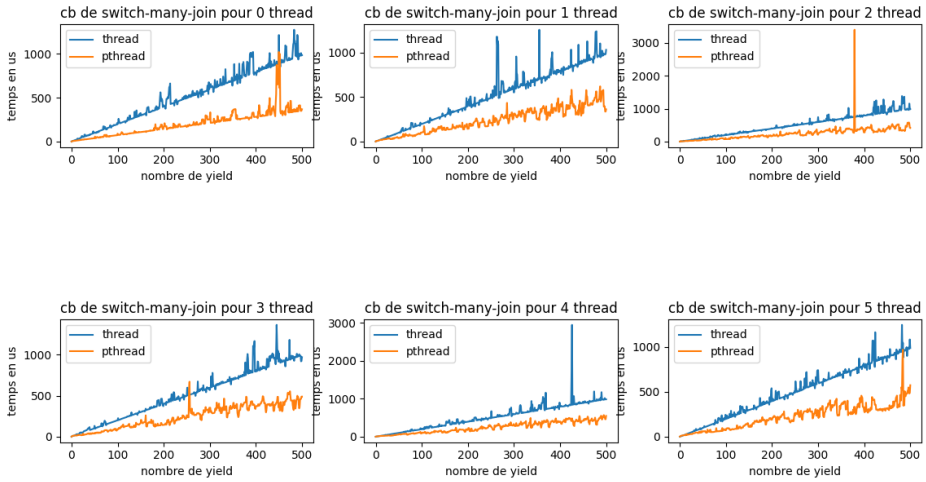

FIGURE 6 – Version avec optimisations : Tests "Switch-Many-Join"

Enfin, sur le test **switch-many-join** dont les résultats sont présentés sur la figure 6, les performances de notre bibliothèque restent malheureusement toujours en retrait de celles de `pthread`, mais une amélioration des temps d'exécution est cependant notable quand la comparaison est faite avec la première version de notre bibliothèque présentée sur la figure 3.

2.4.3 Analyse de la préemption

La préemption ne permet pas un gain de performances sur les tests disponibles. Cependant, imaginons le test suivant : un programme lançant un `thread` "parasite" dont l'exécution est anormalement longue, voire infinie, et un `thread` qui nous intéresse, réalisant une tâche lambda courte. Dans ce cas, la préemption apporte une vraie plus-value dans ce projet.

En effet, si le `thread` "parasite" est lancé avant le `thread` qui nous intéresse, alors ce dernier n'aura la main que dans un long moment, voire jamais. Cette situation serait problématique pour l'intérêt de l'utilisateur, voire même totalement bloquante.

En plus d'ajouter une notion d'équité pour chaque `thread`, la préemption permet d'éviter les `threads` "parasites" bloquant le fonctionnement des autres, tout en n'affectant que peu les performances de la bibliothèque. La différence de performances étant minime, présenter de nouveaux graphiques montrant ces résultats n'était pas intéressant selon nous.

3 Conclusion

Au rendu de ce projet, nous avons au final une bibliothèque de gestion de `threads` basique mais fonctionnelle, qui permet de remplacer en partie la bibliothèque `pthread`. Nous avons implémenté les cinq fonctions de base issues de `pthread` pour le fonctionnement de notre bibliothèque, mais aussi des objectifs avancés (les `mutex` et la préemption, qui utilise également des signaux), sans oublier des optimisations. Nous avons pu tester la bibliothèque avec de nombreux tests qui nous ont été fournis, mais également avec nos propres tests. Enfin, nous avons pu constaté les écarts de performances au cours du projet et des différentes implémentations et ajouts, grâce à des graphes comparatifs entre la bibliothèque `pthread` et la nôtre.

De manière générale, sur un petit nombre de `threads` et de `yields`, notre bibliothèque est plus performante que `pthread`. C'est ce que nous avions également pu constater dans le rapport intermédiaire. Cependant, en augmentant considérablement le nombre de `threads` et de `yields` pour les tests représentés dans les graphes, nous avons pu constater que notre bibliothèque n'était pas plus performante que `pthread` de manière générale.

Des optimisations et des ajouts pourraient évidemment être ajoutés pour compléter et améliorer les performances de notre bibliothèque. Nous avions d'ailleurs commencé à réfléchir aux objectifs avancés support des machines multiprocesseur et priorités, mais n'avons pas eu le temps de les implémenter.