2e année, Filière Informatique
ENSEIRB-MATMECA

— **Projet de Système** —

Rapport Final

Équipe :
Tanguy PEMEJA
Sonia OUEDRAOGO
Julien MIRANDE

Encadrant :
M. Brice GOGLIN
M. Mathieu FAVERGE

Semestre 8

Année 2019/2020

1 Introduction

Dans le cadre d'un projet système nous avions à construire une bibliothèque de gestion threads en espace utilisateur imitant le comportement de la lib pthread. Cela a permis notamment de gérer l'ordonnancement et à des coups plus faible. Dans ce rapport nous présentons notre approche d'implémentation, nous décrivons et expliquons nos choix dans la réalisation.

2 Choix d'implémentation

Dans cette partie, nous allons discuter des différents choix que nous avons faits lors de l'implémentation du projet.

2.1 Structure de nos threads

Le bon choix de la structure de thread était très important car c'est l'élément de base sur lequel repose toute l'implémentation. Nous sommes revenus à plusieurs reprises afin d'ajouter des éléments dans cette structure ou d'en retirer. Voici nos choix de structures dans la version finale du projet :

```
struct my_thread{
  ucontext_t*  uc;
  void * retval;
  int valgrind_stackid;

  TAILQ_ENTRY(my_thread) field;
  int status_exit;
  int status_join;
  struct my_thread* before;
  int priority;

};

struct threads_list{
  my_thread_t * main;
  my_thread_t * activ;
  TAILQ_HEAD(queue, my_thread) queue;
};
```

Notre structure **my_thread** contient une partie inhérente aux threads avec :
- **uc** : le pointeur du contexte de notre thread
- **retval** : un pointeur sur la valeur de retour
- **valgrind_stackid** : permet à valgrind de savoir où se situe la pile de notre thread

Mais notre structure contient aussi une autre partie spécifique à notre implémentation :
- **field** : une entrée sur notre file contenant nos différents threads
- **status_exit** : un entier qui fait office de booléen afin de savoir si notre thread à fini son exécution
- **status_join** : un entier qui fait office de booléen afin de savoir si notre thread attend dans un join un autre thread
- **before** : un pointeur qui nous permet de revenir sur le thread précédent lorsque l'on termine
- **priority** : un entier indiquant la valeur de priorité de notre thread

Concernant la structure **threads_list**, nous n'avons que 3 attributs, **main** qui correspond au pointeur sur notre thread main, **activ** qui correspond à un pointeur sur le thread actuel ainsi que **queue** qui représente notre liste de threads.

2.2 Gestion du thread principal

Au début nous traitions notre main comme les autes threads, on l'allouait et on le libérait de la même manière. Mais il nous a très vite été nécessaire d'avoir un traitement spécifique pour le thread principal qui correspond au *main*. En effet ce dernier avait déjà une pile dédiée. Il ne fallait donc pas, contrairement aux autres threads, lui allouer une pile sur le tas. On a choisi d'initialiser notre thread main dans le constructeur, ainsi nous récupérons juste le contexte de départ. Pareillement pour la libération des zones mémoires des threads, il fallait là aussi veiller à bien gérer le thread principal. Contrairement aux autres threads, son contenu alloué dynamiquement n'est pas libéré lorsqu'il fini son exécution mais plutôt dans le destructeur.

La raison est que le main devait être le dernier thread restant : sa pile n'étant pas allouée de façon dynamique nous n'avons pas besoin de la libérer. Le thread main va alors nous permettre de libérer le dernier thread de notre liste de thread. En effet, il n'est pas possible de libérer le thread sur lequel on est, puisqu'il faut aussi libérer sa pile. Ainsi redonner la main au thread main à la fin est notre seul moyen de libérer tous les threads que l'on a créés.

Contrairement à l'allocation et à la libération, le main est manipulé comme les autres threads. Il est placé dans la liste des threads, ceci est nécessaire pour qu'il puisse agir comme les autres c'est-à-dire, récupérer la main et exécuter son code.

2.3 Algorithmique des fonctions

Dans ce projet, nous devons implémenter 5 fonctions propres aux threads. Pour chacune d'entre elle, nous avons fait des choix afin de les implémenter.

2.3.1 thread_self

Cette fonction retourne l'adresse du thread actif contenu dans notre liste de thread. Elle se fait en temps constant car nous retournons juste l'adresse de *activ* de notre *threads_list*.

2.3.2 thread_create

`thread_create` permet de créer un thread et l'ajouter à la fin de notre file en temps constant. Ce choix permet de faciliter l'exécution des derniers threads créées. En effet les threads qui suivent dans la file et qui sont susceptibles d'être sélectionner lors de `yield`, sont plus récent et ont de plus grandes chances de ne pas être bloqués par un `join`. Toutefois nous avons remarqué qu'une insertion en tête de liste n'impacte pas les performances et les améliores même lors de l'exécution de `fibonacci` comme on pourra le voir dans la section 4.3.4.
De plus dans le makecontext nous avons fait le choix de ne pas fournir la fonction thfunc qui nous est donnée en argument du thread_create. Nous avons préféré plûtot l'encapsuler dans une autre fonction *exec* qu'on appelle dans le makecontext. Ce choix a été motivé par le fait nous avons eu besoin de faire d'autres traitements pour certains threads en plus d'exécuter la fonction thfunc. En effet pour certains threads, la fonction thfunc se termine sans appel à la fonction thread_exit. Cela est problématique notamment si on fait un join après sur ces threads. Ainsi afin de s'assurer que nos threads passent dans tous dans la fonction *thread_exit*, nous l'appelons dans la fonction *exec* avec le retour de thfunc.

2.3.3 thread_yield

Notre fonction `thread_yield` parcours la liste de nos threads jusqu'à trouver un thread n'attendant pas l'éxécution d'un thread comme on peut l'observer sur la figure 1.

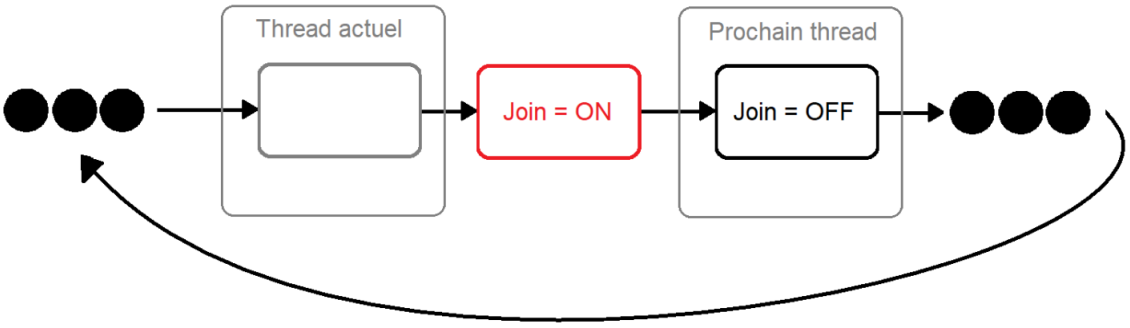

FIGURE (1) Fonctionnement de notre fonction thread_yield

Si jamais, on ne trouve pas de thread capable de reprendre la main, autrement dit lorsque le thread retombe sur lui-même alors celui-ci poursuit son exécution comme si de rien n'était.

La complexité de cette fonction est linéaire, en effet comme on le verra par la suite notre fonction `thread_join` ne retire pas les threads en attente de notre file. Ainsi la file contient aussi des threads en attente, et notre fonction va devoir parcours cette file jusqu'à trouver un thread capable de reprendre la main. On passe donc d'un temps constant à un temps linéaire par rapport à la longueur de notre file.

2.3.4 thread_join

Dans un premier temps notre fonction `thread_join` vérifie si le thread fourni en paramètre n'a pas déjà terminé, grâce à notre attribut `status_exit`. Si c'est le cas alors le thread passe en mode attente en mettant `status_join` sur ON, on définit dans la variable `before` du thread "fils" le thread actuel puis on donne la main à ce "fils" avec un `swapcontext` comme indiqué sur le schéma de la figure 2.

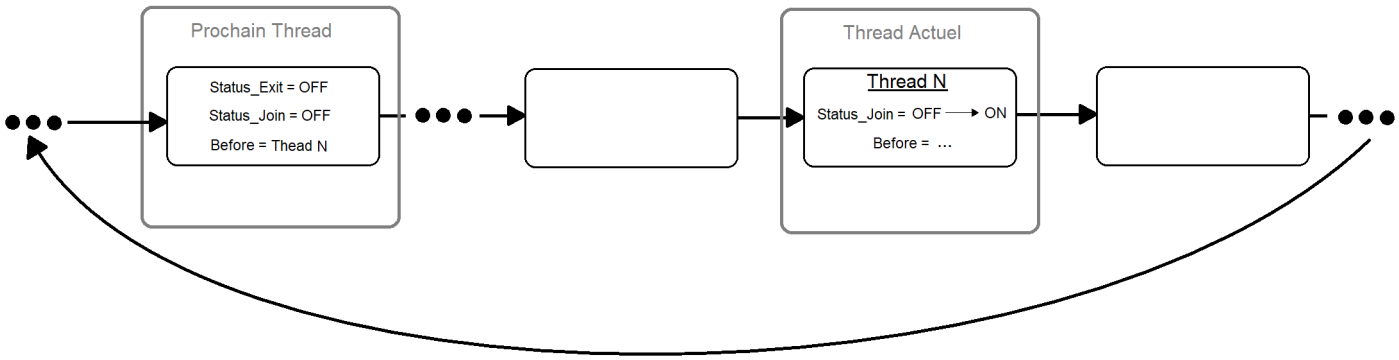

FIGURE (2) Fonctionnement de notre fonction thread_join

Notre attente est une attente passive du fait qu'on ne retourne sur le thread uniquement quand son "fils" a terminé. Toutefois, notre attente n'est pas complément passive étant donné que nous sommes obligés de faire notre `swapcontext` dans une boucle while. En effet, il est possible de revenir sur notre thread si jamais on retourne sur notre thread avec un join. Lorsque notre thread "fils" termine, on écrit la valeur de sortie du thread dans la valeur de retour donné en argument et on libère le thread.

Si on ne considère pas le temps qu'il faut pour que le thread "fils" termine, le temps d'exécution de notre fonction est linéaire. De plus, la complexité supplémentaire de notre fonction `thread_yield` est en partie compensé par une diminution de celle de notre `join`. En effet, une autre possibilité d'implémentation aurai été d'enlever le thread courant, puis de l'ajouter dans la liste une fois le "fils" terminé. Ainsi on aurait alors diminué la complexité de `yield`. Mais le fait de ne pas modifier notre liste de thread, permet de passer d'un temps linéaire à un temps constant.

Notre implémentation est donc plus efficace pour une utilisation de `thread_join` tandis que l'autre est plus efficace pour l'utilisation de `thread_yield`.

2.3.5 thread_exit

La fonction `thread_exit` permet d'enlever le thread de la file, de modifier son statut `status_exit` et de changer la valeur de retour. De plus, dans le cas où le thread n'est pas le thread main, on va alors faire un `setcontext` avec le thread défini par la variable `before` de notre thread si celui-ci existe comme sur la figure 3 suivante.

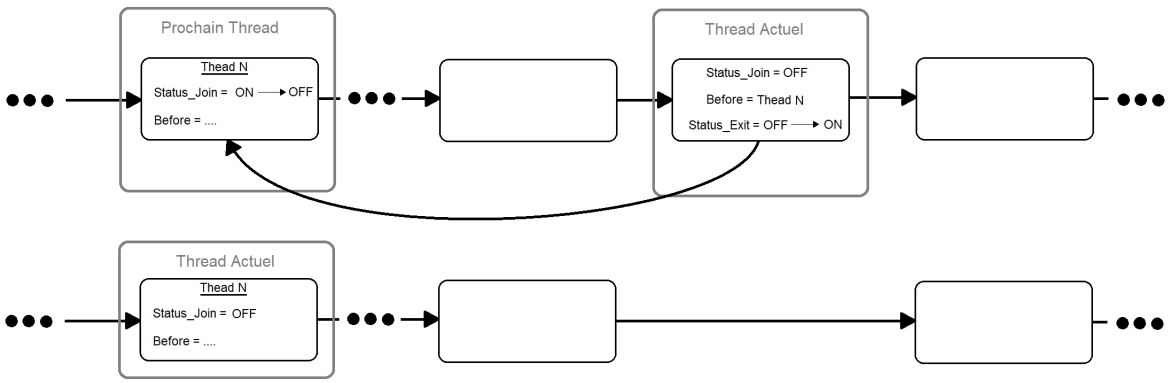

FIGURE (3) Fonctionnement de notre fonction thread_exit

Sinon, on donne la main au prochain thread de la file. Mais si jamais la file est vide alors on redonne la main au thread main.

Un autre cas de figure est que notre thread courant est le thread main et que la file n'est pas vide alors on fait un `swapcontext` avec le prochain thread à exécuter. Et on ne retournera sur le thread main uniquement à la fin de l'exécution de tous nos threads car celui-ci n'est plus dans la file. Un tel traitement du thread main est fait de façon à satisfaire la gestion de la mémoire décrite dans la section 2.2.

3 Implémentation des options avancées

Nous nous sommes penchés sur la réalisation de trois options en particulier.

3.1 La préemption

Le but était de fournir une politique de temps d'exécution uniforme pour nos threads en empêchant que certains threads bloquent les ressources.

Pour mettre en place la préemption nous avons utilisé l'envoi de signal SIGALRM. L'idée est que nous programmons un envoi de ce signal à chaque intervalle de 10ms. Le choix de 10ms est fait de sorte à laisser au thread le temps de s'exécuter, la fréquence du processeur étant de l'ordre du GHz. Dès lors que le signal est reçu, il est traité à travers un gestionnaire de signal que nous avons mis en place par l'appel système sigaction. L'instruction a exécuté étant simplement un yield, pour passer la main à un autre thread.

L'idée principal de la préeemption était donc simple, mais il a fallu bien gérer son intégration dans les fonctions de la lib. Premièrement il fallait prévoir le fait que le signal peut être réçu alors qu'on est en pleine manipulation de la liste de threads notamment lorsqu'on retire un élément. Si on fait un swap_context à ce moment, cela crée des incohérences dans l'exécution voir même des fuites mémoires si on perd le pointeur sur l'élément retiré. Nous avons donc choisi de bloquer la préemption dans ces cas. Pour cela nous utilisons un entier qui passe de 1 à 0 au lieu de l'appel système sigprocmask plus coûteux.
Une deuxième chose était de veiller à réinitialiser l'intervalle avant la prochaine préemption lorsqu'on fait un yield, sans quoi le thread qui prend la main est pénalisé avec le temps d'exécution du précédent.

3.2 Gestion des mutex

Pour permettre à nos threads de manipuler correctement des variables partagées, nous avons implémenté un système de verrou avec les mutex.

```
struct thread_mutex {
    int available;
    thread_t owner;
};
```

Notre structure de mutex contient un entier *available* qui nous permet de savoir si le verrou en question est pris ou pas par un thread, et si oui, le deuxième paramètre, *owner* nous donne l'identité de ce thread.

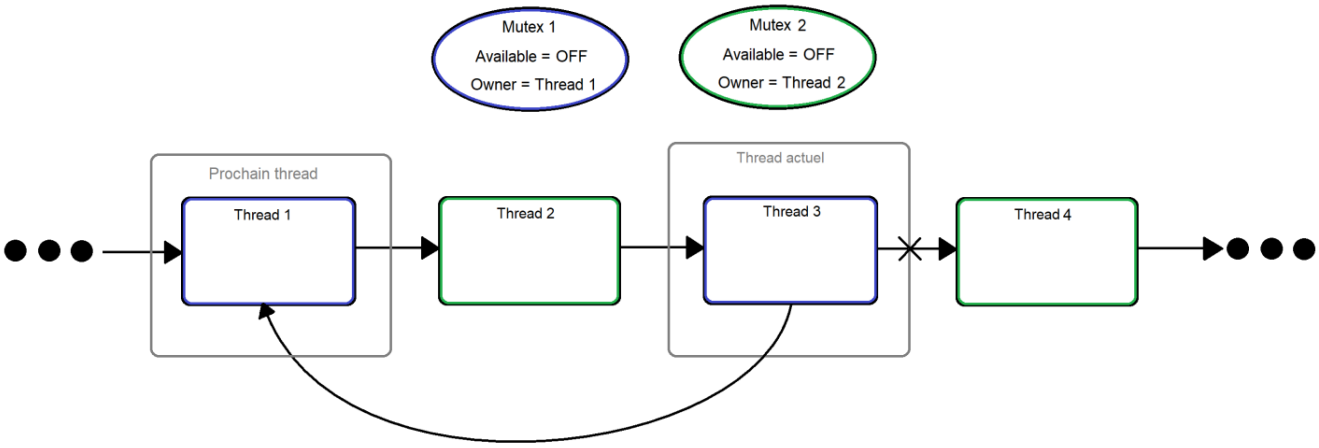

FIGURE (4) Fonctionnement de notre fonction thread_mutex_lock

L'idée principale de la fonction de verrouillage est la suivante. Lorsqu'un thread veut prendre la main sur un verrou, il vérifie d'abord la disponibilité du verrou par l'entier *available*. Dans le cas positif, il change le status du verrou et se place comme propriétaire courant à travers la variable *owner*. Mais là où il nous fallait faire attention c'est le cas où le verrou était déjà pris. Dans un premier temps, nous faisions juste un yield() dans une attente active pour passer la main au thread suivant. Cela générait un temps d'exécution très long (de l'ordre de la seconde) dans certains cas comme celui illustré sur la figure 4. Dans cet exemple nous avons deux verrous et chaque verrou est accessible par un groupe de threads donnés (ici ceux qui ont la même couleur que le verrou). Si l'on suppose que tous les verrous sont déjà pris par les premiers threads, à partir du thread 3 aucun thread ne pourra encore prendre le verrou sans que les threads 1 et 2 les aient lâchés. Dans notre première version nous faisions une série de yield inutile ce qui augmentait le temps d'exécution. Pour améliorer cela, on utilise le paramètre *owner* afin de passer la main au thread qui possède actuellement le verrou. Toutefois cela n'enlève pas l'attente active que nous avons dans cette fonction.
Concernant la fonction de déverrouillage, nous avons choisi d'y intégrer un yield a la fin. Autrement dit lorsque un thread lâche un verrou, il réinitialise les paramètres du mutex puis passe la main au thread suivant dans la liste. C'est aussi la structure des test 61_*mutex* et 62_*mutex* qui nous a incités dans ce sens. En effet tout comme dans ces codes, il peut arriver que le thread ne fasse de yield que quand il possède le verrou. Ainsi aucun autre thread voulant accéder au même verrou ne pourra le faire avant que le *owner* ne finisse entièrement toutes ces modifications de la variable partagée. En ajoutant le yield dans la fonction de déverrouillage, notre objectif est de laisser réellement l'opportunité à un autre de thread de prendre le verrou et ainsi illustrer que nos threads peuvent alterner correctement des modifications sur la variable partagée.

3.3 Gestion des priorités

Pour gérer la priorité, nous avons ajouté un champ *priority* à notre structure *my_thread_t* qui est initialisé à -1. Ainsi, tous les threads ayant une priorité fixée à -1 sont traités comme si ils n'avaient pas de priorité. Pour pouvoir laisser plus de liberté à l'utilisateur, nous utilisons une fonction pour modifier la priorité d'un thread déjà créé.

```
extern int thread_setschedprio(thread_t *thread, int priority);
```

Cette fonction prend en paramètres un pointeur vers un thread pour pouvoir modifier son ancienne priorité par la nouvelle. La priorité est fixée entre 1 et 99 (où 1 est la priorité la plus faible donc une probabilité plus faible d'être exécuté et 99 la priorité la plus forte). Notre réalisation pour la gestion des priorité est incomplète car actuellement les threads sont replacés dans la file de threads en fonction de leur priorité mais uniquement après l'appel de cette fonction et non pas après chaque exécution du thread. Cependant, nous avons pensé à deux manières d'implémenter la priorité pendant l'exécution :

- Une première solution est de faire évoluer le placement de chaque thread après son exécution (donc à chaque yield) c'est à dire que plus le thread à une priorité élevée, plus ce thread est replacé à l'avant de la file avec une forte probabilité (et donc à l'arrière avec une priorité faible). Après chaque yield, il ne faudrait plus prendre le prochain thread libre mais le premier dans la file.
- La deuxième solution est de faire une fonction qui nous retourne le prochain thread. Chaque thread possède son propre intervalle de valeur suivant sa priorité, plus la priorité est grande et plus notre intervalle l'est aussi. Ainsi en juxtaposant les intervalles des threads de notre file et en générant un entier aléatoirement dans cet intervalle, il nous serait possible d'obtenir le prochain thread à exécuter suivant l'emplacement de notre entier dans l'intervalle. Une plus grande

priorité entraînerait alors une plus grande taille d'intervalle et donc une plus grande probabilité d'être le prochain thread.

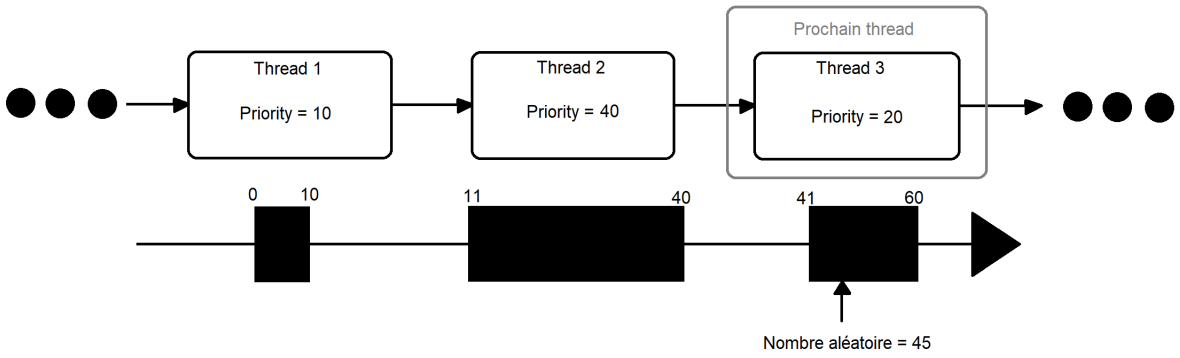

FIGURE (5) Idée de réalisation de la priorité à l'aide d'intervalle

Sur la figure 5, le prochain thread qui va être exécuté est le `Thread 3`. En effet, le nombre généré aléatoirement 45, appartient à l'intervalle 41-60 qui correspond au thread 3 ayant une priorité de 20.

Ainsi, si on lance plusieurs threads avec des priorités min et max, les threads avec la priorité max devraient quasiment monopoliser le fil d'exécution avec les yields et terminer en premier. C'est ce qu'on a essayé de faire à travers notre test. Cependant, il est difficile de faire des stats sur l'ordre d'exécution des threads pour vérifier notre implémentation.

4 Analyses des résultats

4.1 Cohérence avec la bibliothèque pthread

Plusieurs tests ont été réalisés afin de valider d'abord visuellement le comportement de nos threads par comparaison avec le biliothèque pthread. A la suite de ces tests nous avons pu constater la validité de notre librairie. Certains cas particuliers ont pu être traités afin de correspondre totalement. Par exemple :

— Le *join* sur un thread est fait correctement peu importe que ce dernier fasse appel ou pas à la fonction exit durant son exécution. En effet comme expliqué dans la section 2.3 nous forçons nos threads à passer dans la fonction exit. Ainsi s'il arrive que le join est fait avant le exit, nous récupérons juste le pointeur en argument du join afin d'y stocker la valeur de retour de thread une fois qu'il fera le exit. Et dans le cas contraire si le exit est fait avant le join, nous stockons la valeur de retour (en argument du exit) dans le retval du thread afin de le récupérer dans le join.
Nous avons également géré dans le join le cas où le main, thread principal, termine avant ses fils. En effet en ayant un traitement particulier pour le thread main (section 2.2), nous nous assurons qu'il soit le dernier à être libéré même s'il termine effectivement son exécution avant ses fils.

— Concernant la fonction *yield*, le comportement attendu est le bon et l'ordre d'exécution dans la file est respecté. Par ailleurs nous validons le fait qu'un thread puisse faire des *yield* même s'il est seul dans la liste sans produire de message d'erreur.

4.2 Validation valgrind

En plus de valider le comportement des threads, nous avons veillé également à assurer une bonne gestion mémoire dans notre implémentation en libérant correctement nos ressources allouées. Une règle **make valgrind** permet de lancer tous les tests les uns à la suite des autres. Pour éviter la verbosité de valgrind tout en gardant les résultats, nous avons mis en place deux solutions :

— Dans le terminal, nous affichons seulement le nom du test, les commandes de compilation, son exécution ainsi que le résultat des *free* et *errors* comme on peut le voir sur la figure 6.

```
gcc -g -Wall -Wextra -c src/test/51-fibonacci.c -o install/bin/51-fibonacci.o -Isrc/thr
ead
gcc -g -Wall -Wextra -o install/bin/51-fibonacci install/bin/51-fibonacci.o -lthread -L
install/lib -lrt
fibo de 20 = 6765 in 6.193940e+00 s
/************* 51-FIBONACCI ****************/
Test free valgrind : SUCCESS
Test error valgrind : SUCCESS
```

FIGURE (6) Affichage des résultats des tests valgrind

— Dans un deuxième temps, nous créons un fichier de log placé dans la dossier **bin** contenant l'ensemble de la sortie de valgrind.

Nous avons créé une règle pour chaque test ainsi il est possible de les tester individuellement. De plus, tous les tests sont au verts sur la forge et valgrind ne détecte aucune erreur de contexte ou de libération de mémoire allouée dans aucun de nos tests.

4.3 Graphes de performance

Pour comparer les performances entre la bibliothèque *pthread* et nos implémentations, nous avons créé 3 scripts python dont on peut spécifier des **MODES** lors de nos `make` afin de compiler les tests avec les bonnes librairies et afficher différentes courbes. Les modes `head` et `premp` correspondent respectivement au fonctionnement avec une insertion de nos threads en début de file et l'utilisation de la préemption. De plus, pour un meilleur visuel, les axes de nos graphes sont en base logarithmique. Ainsi lorsqu'on parle de complexité linéaire et de coefficient directeur, il s'agit en réalité d'une complexité exponentiel et de l'exposant de celle-ci. De plus, voici la configuration avec laquelle les tests ont été réalisés :

— Système virtuel : VM VirtualBox, Ubuntu, 2 coeurs, 5245 Mo de RAM

— Système hôte : Intel Core I7 2.60GHz, Windows 10, 4 coeurs, 8Go de RAM

4.3.1 Résultat global

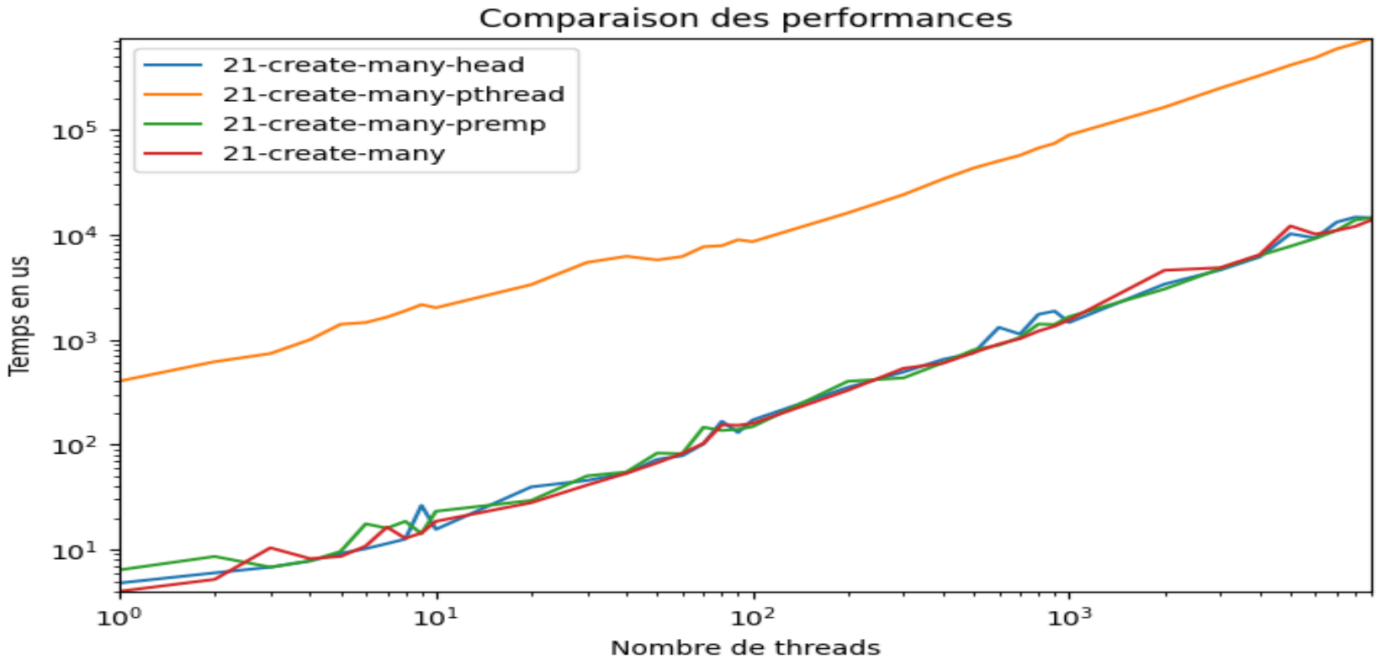

FIGURE (7) Performances pour le test 21-create-many

Lors d'une utilisation simple de nos bibliothèques, nos implémentations sont plus efficaces que la bibliothèque `pthread` comme le montre la figure 7. Ce facteur 10 que l'on gagne provient du temps d'initialisation de `pthread`. En effet, `pthread` est peu efficace lorsque l'on utilise que peu de threads par comparaison à nos bibliothèques. Mais ce retard est compensé lorsqu'on tend vers un nombre de thread plus élevé, la bibliothèque `pthread` a alors un coefficient directeur égale à celui de

nos implémentations. Les différentes bibliothèques tendent alors vers la même allure linéaire en échelle logarithmique (et donc exponentielle) mais gardent toujours le décalage dû au surcoût d'initialisation de `pthread`.

4.3.2 Problème de performance avec la préemption

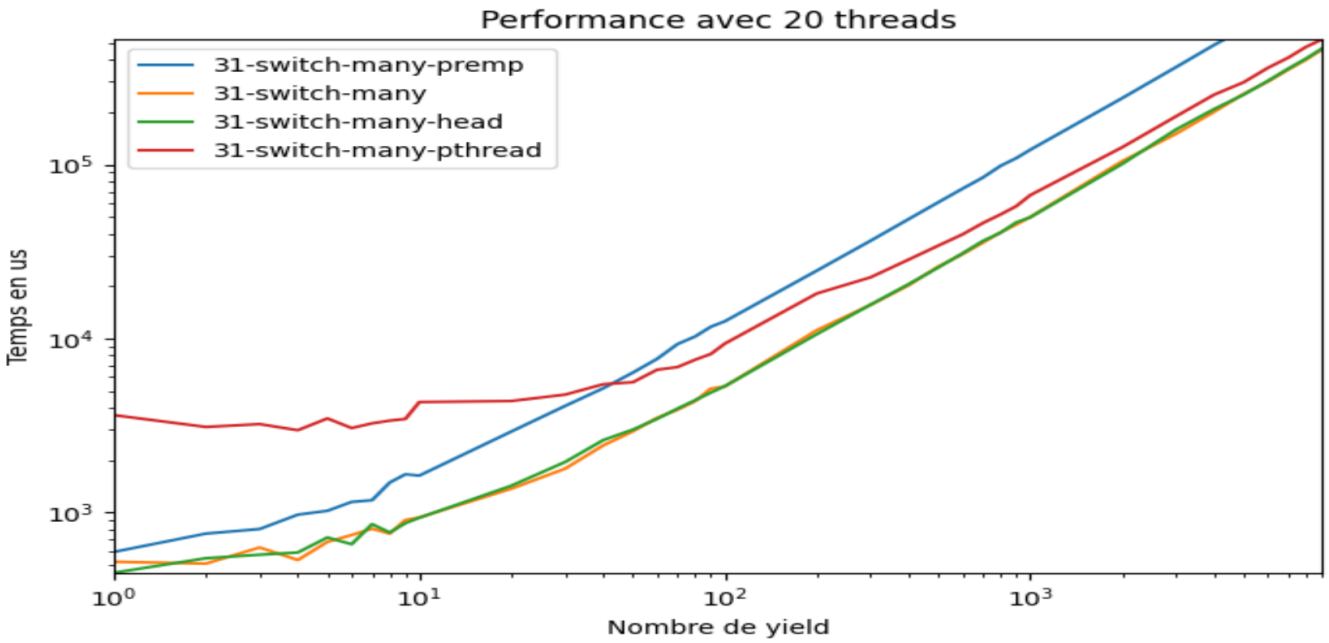

FIGURE (8) Performances pour le test 31-switch-many en variant le nombre de yield

Dans la figure 8, on constate que notre bibliothèque est environ 10 fois plus rapide à l'initialisation que pthread mais les courbes convergent l'une vers l'autre vers un même coefficient directeur pour un très grand nombre de yields. En revanche, on voit clairement que notre bibliothèque avec la préemption est légèrement plus lente que pthread. Nous pensons que cet écart de performance est dû au nombre d'appels systèmes à travers la fonction *update_timer()* qui réinitialise le timer après chaque yield.

4.3.3 Mise en évidence du problème de complexité de nos yields

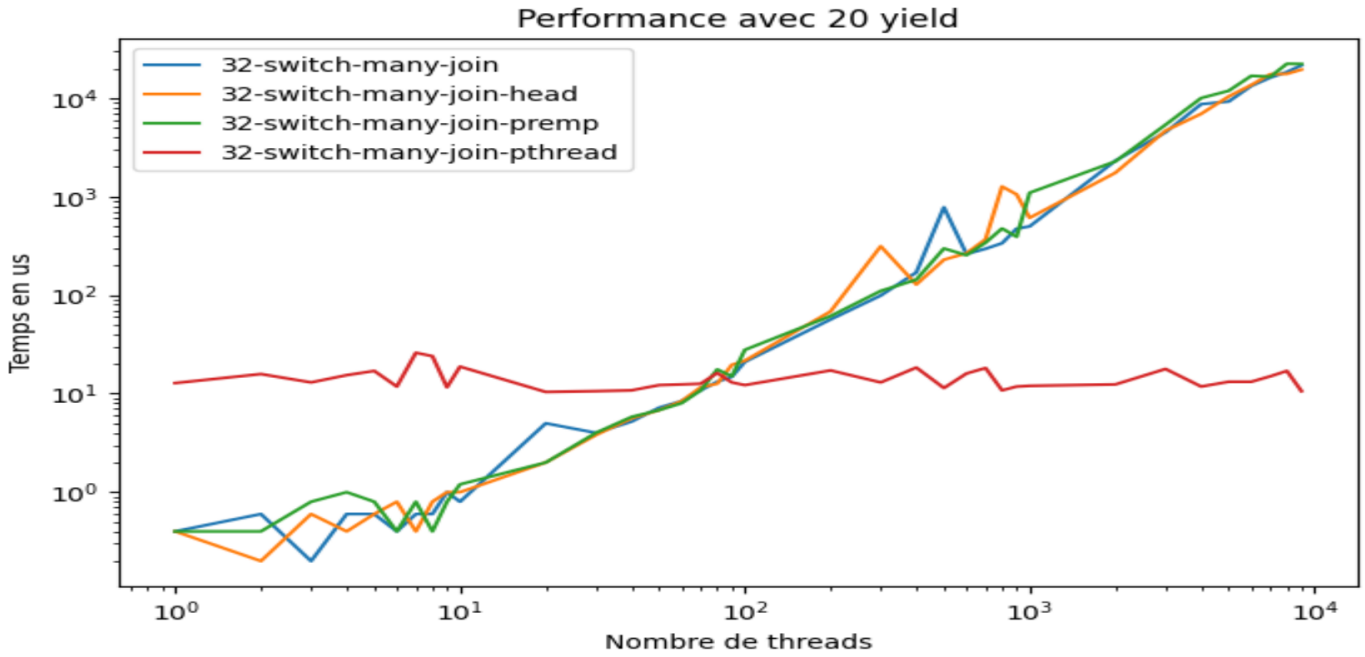

FIGURE (9) Performances pour le test 32-switch-many-join en variant le nombre de thread

Même si les courbes sont assez peu lissées par manque de points, la figure 9 est intéressante car c'est le seul cas où notre bibliothèque est plus lente que *pthread* pour un grand nombre de threads. En effet, *pthread* garde une complexité plus ou moins constante selon le nombre de threads alors que notre bibliothèque croit linéairement en fonction de nombre de threads. Les résultats sont tout à fait cohérents avec notre implémentation car nous ne supprimons jamais les threads en attente dans la file. Ainsi, à chaque yield on doit parcourir une liste de plus en plus longue (ici 10^4 threads) ce qui ralentit les performances. Une autre politique aurait été de supprimer les threads en attente et de les remettre lorsque le thread "fils" termine. Dans ce cas, nous gagnons du temps pour trouver le prochain thread dans le yield mais nous en perdons pour le supprimer et le remettre dans le join.

4.3.4 Gain de performance de nos bibliothèques

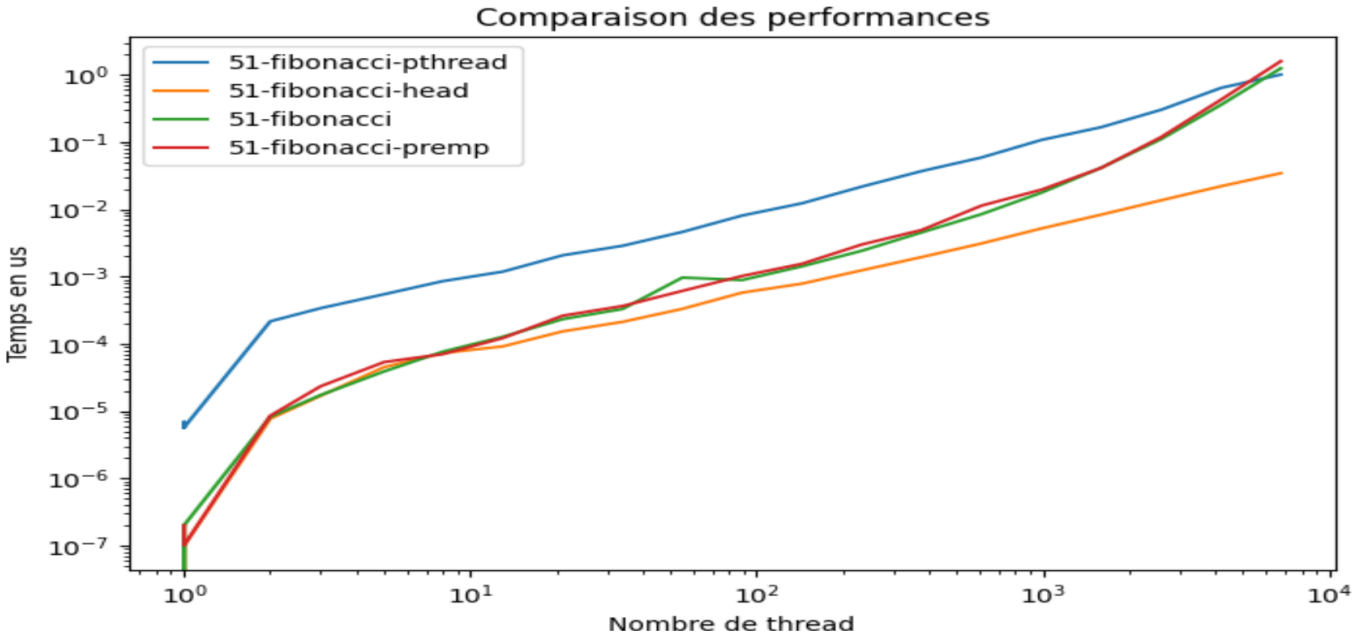

FIGURE (10) Performances pour le test 51-fibonacci

Pour terminer, analysons le calcul de fibonacci de 20 avec les différentes bibliothèques. On constate que **pthread** croit linéairement et est plus lente que nos implémentations pour la bibliothèque de base et celle avec la préemption, jusqu'à fibonacci de 20. Ces 2 implémentations ajoutent les nouveaux threads en fin de liste. Or, en ajoutant à la fin de la liste, l'ordonnancement est moins bon pour nos yields et on tombe plus souvent sur des threads bloqués par des join. C'est donc pour cette raison que nous avons essayé en insérant en tête les nouveaux threads. Dans le cas de fibonacci, nos yield sont alors plus rapide car l'ordonnancement est plus favorable. Il est normal de voir que notre bibliothèque est au final plus rapide (d'un facteur 10) que *pthread*. Au final, nous arrivons à un fibonacci de 20 en envrion 0.03 s.

5 Conclusion

Ce projet a été très stimulant pour nous. Nous avons pu implémenter des threads en espace utilisateur et analyser les performances d'une telle implémentation mais aussi faire face à certains problèmes. Cela nous a aussi permis d'expérimenter pour la première un changement de contexte. Nous nous sommes efforcés de répondre au mieux aux attentes du sujets tout en améliorant notre algorithmique.