Semestre 8

Avril 2022

Rapport final du projet de systèmes d'exploitation

Filière Informatique - ENSEIRB-MATMECA

Auteurs :
DO Nicolas
Khattab Omar
Kunow Johannes
NASDAMI Quatadah

Encadrant :
SWARTVAGHER Philippe

1 Introduction

Ce projet a pour but de programmer en C une bibliothèque de threads en espace utilisateur afin de nous permettre de mieux appréhender le fonctionnement de l'ordonnancement des threads mais également de mieux comprendre les changements de contextes et les différences de performances que nous pouvons obtenir entre notre bibliothèque qui n'utilise que des changements de contextes et la bibliothèque pthread qui utilise des appels systèmes.

2 La version de base

Une première version de cette bibliothèque de gestion de threads, s'appuyant sur un ordonnancement coopératif, c'est à dire sans préemption et uniquement basé sur les appels à *thread_ yield*, et à politique *First In First Out* a été implémentée.

2.1 La structure d'un thread

La structure des threads que nous utilisons est la suivante :

```
struct thread_t{
    int id;
    STAILQ_ENTRY(thread_t) next_thread;
    ucontext_t *context;
    enum state etat;
    void * retval;
    struct thread_t* waiting_thread;
    int valgrind_stackid;
    STAILQ_ENTRY(thread_t) next;
};
```

où *id* permet à identifier le thread (plutôt utile pour le débogage), *context* permet de sauvegarder le contexte du thread, *etat* donne l'état actuel du thread (est ce que le thread est fini ou pas), *retval* permet de stocker la valeur de retour du thread, *waiting_ thread* correspond au thread qui est en train d'attendre le thread en question (dans le cadre de la fonction *thread_ join*) et *valgrind_ stackid* qui renseigne où se trouve la pile du thread et qui est à libérer lorsque le thread termine.

2.2 L'ordonnancement

L'ordonnancement se repose sur une politique *First In First Out*. Pour se faire, nous avons mis une place une file d'attente à liaison unique (STAILQ) où placer les threads à exécuter. Selon la fonction appelée, un thread peut être ajouté ou retiré de cette file d'attente. Par exemple, lors de l'appel à *thread_ create*, nous allons ajouté le thread fraîchement créé à la fin de la file d'attente, dans le cas d'un appel à *thread_ yield*, nous allons retiré le thread en tête de la file pour le replacer en fin de file.

2.3 Le thread de la fonction main

La fonction *main* du programme principale doit se comporter comme un thread et doit être traitée de manière identique que les autres threads. Afin d'associer un thread à cette dernière, il a fallu initialiser ce thread avant même l'exécution de la fonction *main*. Pour

remédier à ce problème, nous avons implémenté la fonction *init_ var* qui possède __attribute__((constructor)), ce qui permet à cette fonction d'être exécuté lors du chargement de la bibliothèque partagée donc avant même l'exécution de la fonction *main*. Cette fonction a pour but d'initialiser le thread du programme principale, qui se comportera alors de la même façon que les autres threads, et de l'ajouter à la file d'attente.

2.4 La fonction wrapper

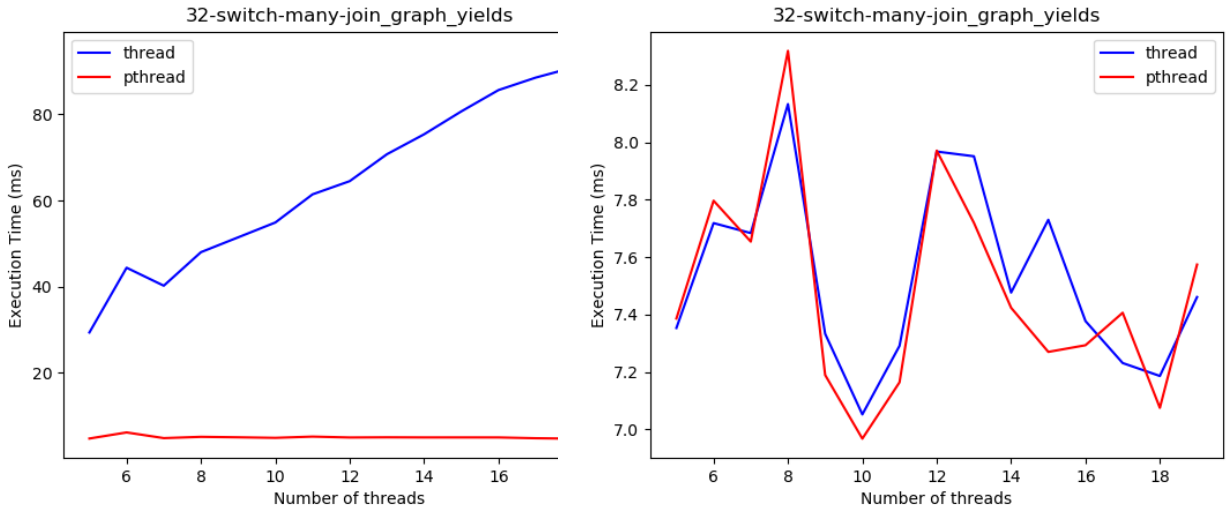
Dans la fonction *thread_ create*, une particularité a dû être mis en place : il s'agit de la fonction *wrapper* qui est passé en paramètre de la fonction *makecontext*. Cette fonction prend en paramètre une fonction et ses arguments (qui sont ceux passés à la fonction *thread_ create*). Elle permet de faire un unique appel à la fonction *thread_ exit* à la fin de la fonction qui est associée au thread, que l'utilisateur ait lui même fait appel à *thread_ exit* à la fin de sa fonction ou non. Sans cela et dans le cas où l'utilisateur n'aurait pas fait appel à thread_exit à la fin de sa fonction il aurait été impossible de savoir si un thread avait terminé.

2.5 La fonction thread_join

L'implémentation de la fonction *thread_ join*, initialement envisagée, consistait en une attente active, c'est à dire à vérifier que le thread passé en paramètre de la fonction était terminé et si ce n'était pas le cas, on appelait la fonction *thread_ yield* pour passer la main, et cela jusqu'à que le thread en paramètre soit marqué comme terminé.

```
while (thread->etat == VIVANT){
        thread_yield();
}
```

Néanmoins, cette implémentation nous a rapidement paru peu performante car étant donné que dans *thread_ join*, nous n'attendons qu'un seul thread, il s'avérait dommage de devoir systématiquement revenir sur le thread appelant *thread_ join*, tout cela pour immédiatement repasser la main. La solution que nous avons opté a été de sauvegarder le thread appelant *thread_ join* dans le champ *waiting_ thread* du thread qui est attendu. En faisant cela, on peut simplement enlever le thread appelant de la file d'attente et le re-introduire dans la file seulement lorsque le thread attendu ait terminé (appel à *thread_ exit*). La différence de performances entre ces deux implémentations est visible Figure 1. On remarque très clairement que l'implémentation sans attente active est plus performante que celle avec attente active (en moyenne 10 fois plus rapide).



(a) 20 threads avec 10000 yields avec attente active (b) 20 threads avec 10000 yields sans attente active

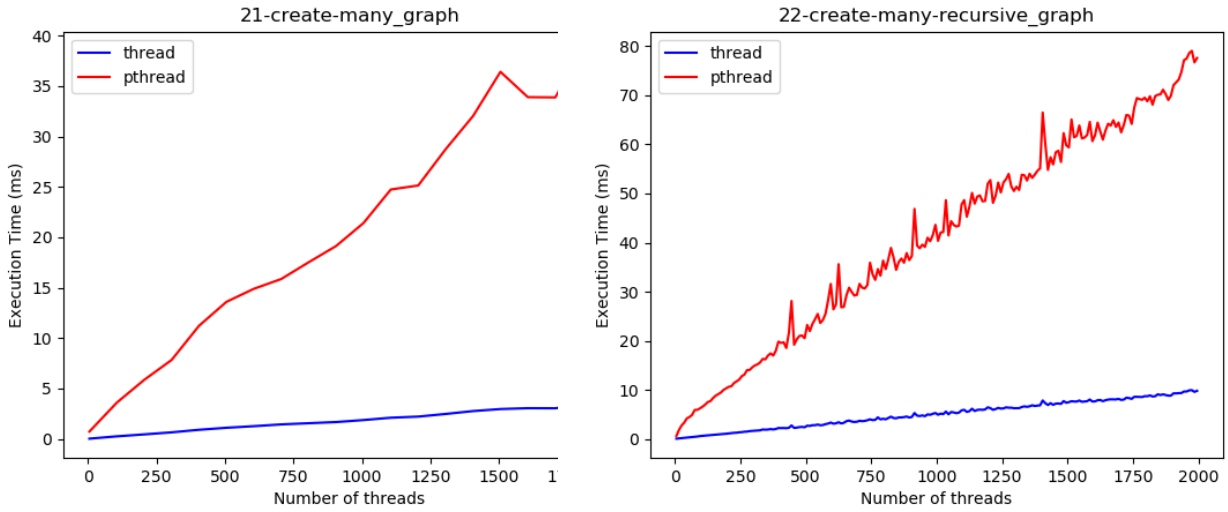
FIGURE 1 – Performances selon l'implémentation de thread_join

2.6 La libération de la mémoire

La libération de la mémoire se fait en toute fin de la fonction *thread_join*. Cette dernière libère la mémoire alloué du thread passé en paramètre qui est alors terminé pour de bon. Il est à remarquer que la libération de la mémoire ne se fait pas lors de l'appel à *thread_exit* car nous avons besoin de faire remonter certaines informations comme la valeur de retour du thread et son état.
Une fonction *destroy_var* possèdant __attribute__((destructor)) permet de libérer les threads qui n'ont pas été libérés par le biais de la fonction *thread_join* en fin de programme.
Le plus gros problème que nous avons rencontré en rapport avec la libération de la mémoire a été le cas où un thread fait un appel à *thread_join* sur le thread de la fonction *main*, autrement dit, le cas où le thread terminant le programme n'est pas le thread de la fonction *main*. En effet, lorsque l'on essayait de libérer la pile allouée au thread depuis son propre contexte, nous avions des erreurs valgrind. Pour résoudre ce problème, il a fallu faire en sorte de revenir sur le contexte du programme principale afin de libérer les ressources allouées par le dernier thread. Pour ce faire, une fois le thread identifié comme celui terminant le programme, nous faisons un appel à *setcontext* pour revenir sur le contexte du *main* stocké en variable globale. Nous libérons ensuite les ressources allouées par le thread et nous terminons le programme.

2.7 Threads utilisateurs contre pthreads

L'utilisation de notre bibliothèques de threads en espace utilisateur comporte, pour certains aspects, un avantage en termes de performances par rapport à l'utilisation des pthreads. Notamment, comme le montre la Figure 2, la création de threads en espace utilisateur est beaucoup moins coûteuse en temps que la création de pthreads. En effet, contrairement à pthreads qui doit faire des appels systèmes (donc très coûteux), la création de nos threads en espace utilisateur ne fait qu'une allocation dynamique.

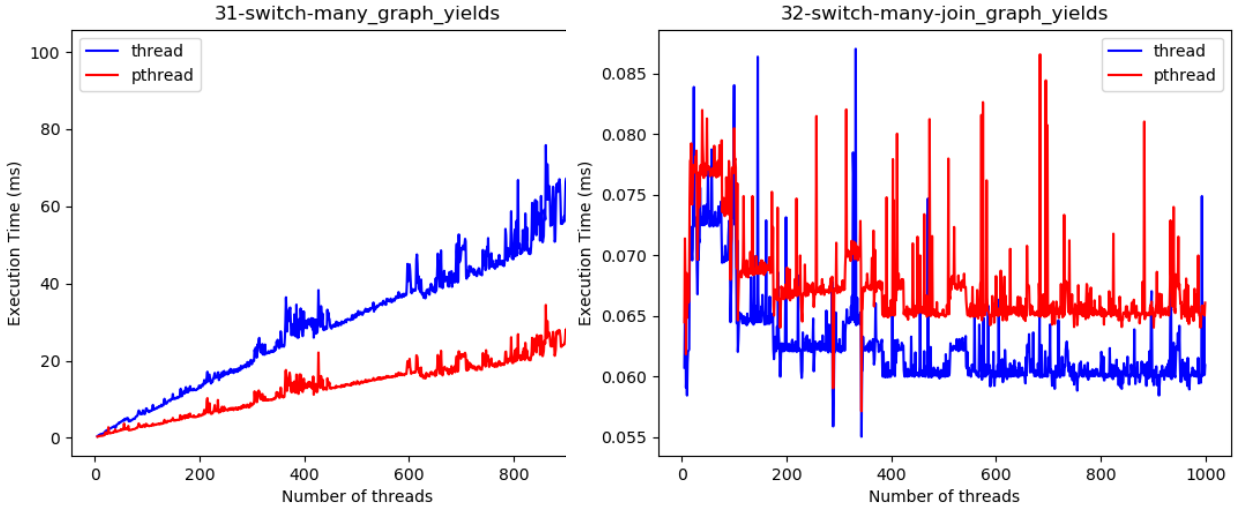


(a) Graphe de performances sur la création de 1 à 2000 threads

(b) Graphe de performances sur la création récursive de 1 à 2000 threads

FIGURE 2 – Performances sur le création de threads

Ensuite, avec la Figure 3a, pthread est largement meilleure pour passer la main à un autre thread par rapport à notre bibliothèque. Le coût de notre bibliothèque vient du fait que nous devons faire le changement de contexte à chaque fois qu'on passe la main alors que pthread non. Néanmoins comme le montre la Figure 3b, lorsque l'on passe la main alors que les autres threads sont bloqués dans *join*, les performances de notre bibliothèque sont presque identiques à celle de pthread car nous n'avons plus de changement de contexte à faire étant donné qu'on a un seul thread dans la *runqueue* (grâce à l'attente passive de *thread_join*).



(a) Graphe de performances sur 20 yields par 1000 threads

(b) Graphe de performances sur 20 yields en cascade par 1000 threads

FIGURE 3 – Performances sur le thread_yield

3 Les objectifs avancés traités

Une fois la version de base fonctionnelle et les divers contraintes imposées respectées (Makefile avec toutes les règles du sujet, la génération de graphes de performances, les tests fournis et les tests supplémentaires qui sont fonctionnels sans erreurs valgrind), nous avons pu ajouté à notre implémentation de nouvelles fonctionnalités. Nous avons notamment traité l'ajout de mutex permettant aux threads de manipuler des données partagées de manière sécurisée et la détection d'un cycle de threads qui joinent leur suivant et donc sont bloqués en deadlock. En plus de cela, nous avons tenté de traité différente méthodes d'ordonnancement comme l'ordonnancement avec priorité ou la préemption mais n'avons pas abouti à des implémentations totalement fonctionnelles par manque de temps.

3.1 Les mutex

Les mutex sont des éléments permettant d'éviter que des ressources partagées ne soient utilisées en même temps.

La structure des mutex que nous utilisons est la suivante :

```
typedef struct thread_mutex{
    enum lock_state etat;
    STAILQ_HEAD(, thread_t) wqueue;
} thread_mutex_t;
```

où *etat* donne l'état actuel du mutex (LOCKED, UNLOCKED ou DESTROYED) et *wqueue* correspond à une file d'attente contenant les différents threads qui attendent le verrou.

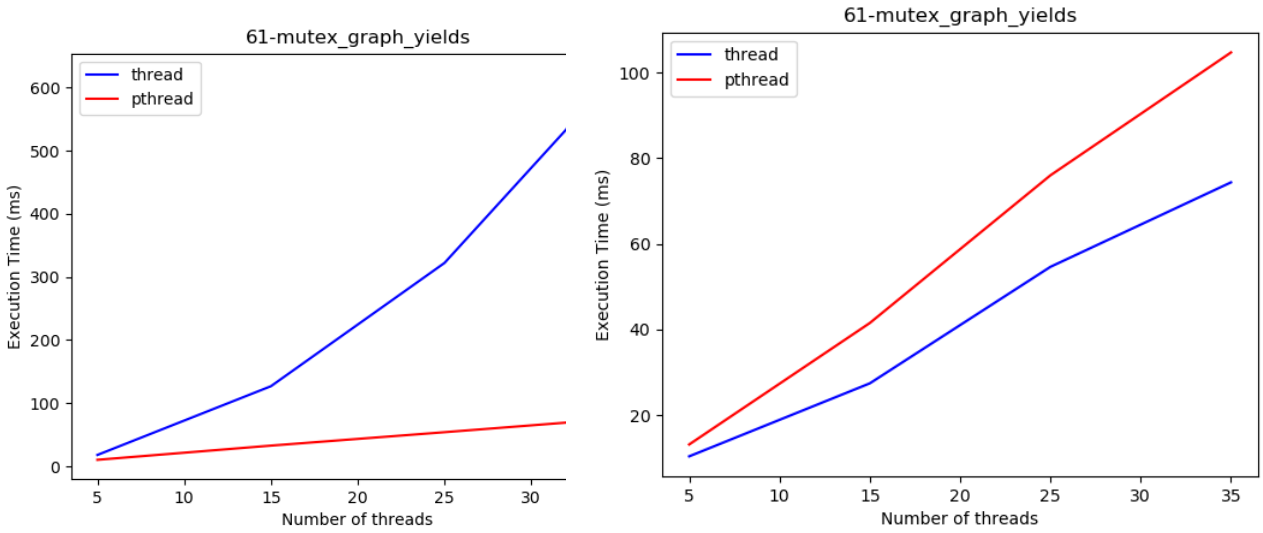
A l'instar de la fonction *thread_join*, nous avons dans un premier temps implémenté la fonction *thread_mutex_lock* avec une attente active

```
while (mutex->etat == LOCKED){
        thread_yield();
}
```

puis, dans un second temps, nous avons mis en place une attente passive avec la même méthode que pour *thread_join*, c'est à dire que nous stockons dans l'attribut du mutex tous les threads qui attendent le mutex et nous retirons ces threads en question de la *runqueue*. Dès que le mutex est débloqué, on réintroduit un des threads qui attendaient en à la fin de la *runqueue*.

La différence de performance entre ces deux implémentations est visible Figure 4. On remarque qu'avec une attente active, notre bibliothèque aurait été beaucoup moins performante que celle de pthread alors que sans attente active, elle est légèrement mieux que celle de pthread. On remarque qu'on réduit le temps d'exécution d'un facteur d'environ 10 entre les deux implémentations.

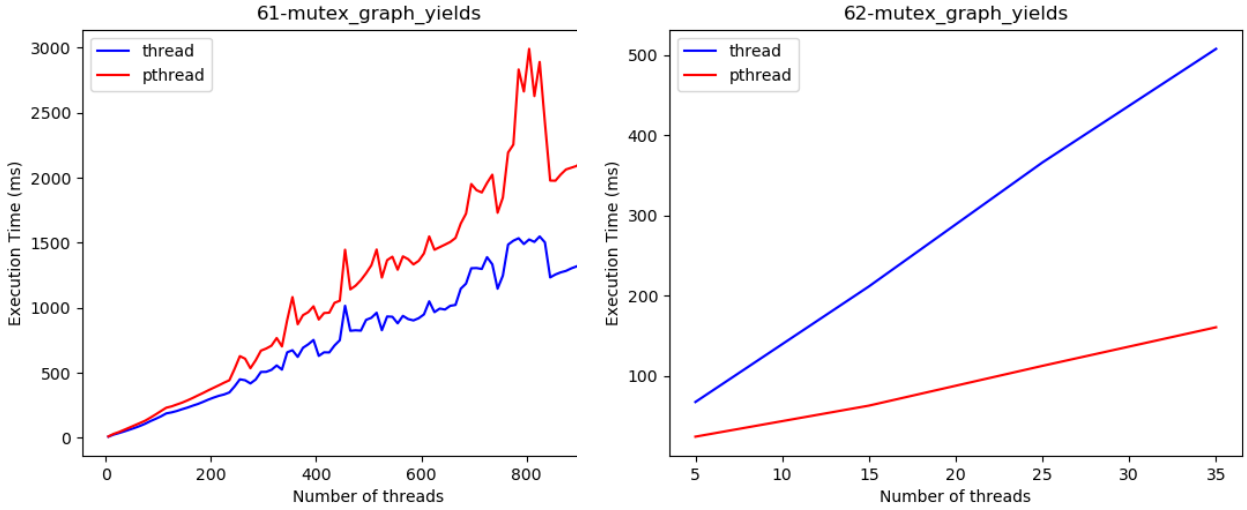
Les différences de performances entre notre bibliothèque et celle de pthread pour le test 61-mutex et 62-mutex avec un plus grand nombre de threads visibles Figure 5. On remarque que pour la gestion d'un mutex, notre bibliothèque est plus performante mais lorsqu'il s'agit d'en gérer plusieurs, pthread est meilleure.



(a) avec attente active

(b) sans attente active

FIGURE 4 – Performances selon l'implémentation des mutex



(a) Test 61-mutex

(b) Test 62-mutex

FIGURE 5 – Performances sur les tests 6*-mutex

3.2 Deadlock de join()

Le problème du Deadlock de join est schématisé comme dans la figure 6. Le **thread 0** attend la terminaison du **thread 1**, le **thread 1** à son tour attend la terminaison du **thread 2**, et puis le **thread 2** attend le **thread 0**. Cette attente circulaire est un problème pour nous, puisqu'aucun des threads ne terminera à la fin.

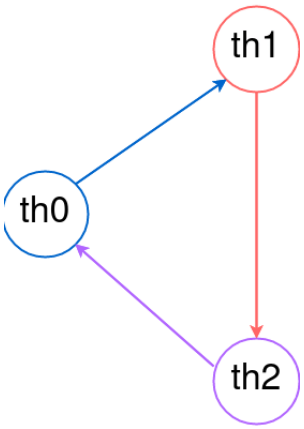


FIGURE 6 – Deadlock

Nous avons réussi à résoudre ce problème. Notre solution consiste à rajouter une vérification au début de la fonction *thead_join()*, qui permet de voir si le thread attendu fait déjà parti de la chaîne de threads qui attendent leur suivant. Si c'est le cas, nous avons la présence d'un cycle, c'est à dire une situation où les threads serait bloqués en deadlock. Le code de cette vérification est visible Figure 7.

```
1   thread_t t = current_thread->waiting_thread;
2     while(t != NULL){
3         if (t == thread){
4             return -1;
5         }
6         t = t->waiting_thread;
7     }
```

FIGURE 7 – Résolution de deadlock

.

3.3 Priorités

Dans la littérature, plusieurs algorithme proposent des méthodes pour ordonnancer l'exécution des threads. Chacun a ses avantages et ses défauts. On peut citer par exemple, l'algorithme le plus naïf "**Premier arrivé premier servi**". L'algorithme de **Round-Robin**, qui exécute un thread pendant un laps de temps constant (**quantum**), puis le retourne à la fin de la file

d'attente. Chaque thread reçoit dans ce cas une part égale des cœurs. Ou encore l'algorithme **SJF** qui exécute sans interruption le thread qui se terminera le plus rapidement,

Dans une version qui ne s'est pas achevée au moment d'écrire ces lignes, nous avons tenté d'implémenter un ordonnancement des threads se basant sur des priorités. Nous avons définis trois niveaux de priorité, **élevée**, **moyenne**, **faible** comme le montre la figure 8.

```
typedef enum {
  LOW,
  NORMAL,
  HIGH
} priority;
```

FIGURE 8 – Niveaux de priorités

Chaque niveau de priorité donnant droit à une queue de threads utilisateurs,fonctionnant selon le principe FIFO, selon leur ladite priorité.

En principe, la priorité élevée est donnée aux threads qui doivent prendre la main dans les plus brefs délais possibles, sans trop attendre. La priorité moyenne est donnée aux threads qui peuvent attendre la fin d'exécution d'autres threads plus prioritaires. Et enfin, la priorité faible est fourni au threads qui doivent à fortiori attendre l'exécution des autres. Le principe d'ordonnancement est assez intuitif. Quand un thread est créé, il est de priorité moyenne puisque l'on connaît aucun à priori sur lui. Dès que la fontion thread_join est appelée sur un thread, ce dernier prend automatiquement une priorité élevée, le thread appelant cette fonction prend une priorité faible, et la main passe à un autre thread de priorité élevée choisit par la fonction thread_yield. Quand le thread attendu termine son exécution, le thread en attente reprend une priorité élevé pour qu'il puisse s'exécuter dans les plus brèves délais.

La fonction thread_yield, qui choisit le thread à exécuter, commence par la queue de priorité élevée. Si cette queue est vide, le choix est fait sur la queue de priorité moyenne et ensuite sur celle de priorité faible.

On peut imaginer dans une version plus élaborée, qui peut être implémentée ultérieurement, un ensemble de n priorités fonctionnant selon le principe suivant :
Initialement, un quantum est fixé, et l'ensemble des threads sont exécutés pendant ce quantum. Ensuite, une priorité discrète est affectée progressivement aux threads proportionnellement à leur temps d'exécution.

Un nouveau thread qui n'a pas eu encore la main aura une priorité élevé de l'ordre de $\frac{1}{Quantum}$ et s'éxecutera pendant un quantum. Un thread qui a déjà eu une fois la main aura une priorité de l'ordre de $\frac{1}{2*Quantum}$ et s'exécutera pendant 2 Quantums lorsqu'il sera choisit. Ou encore, un thread qui a déjà eu k fois la main aura une priorité de l'ordre de $\frac{1}{k*Quantum}$ et s'exécutera pendant k Quantums lorsqu'il sera choisit. De cette manière, tous les threads

prendront la main au moins une seule fois dès leur arrivée et les threads qui nécessitent un temps d'exécution relativement important prendront une priorité faible à fur et à mesure. Notons qu'on peut bien jouer avec l'aléa pour choisir des threads de priorité faible même en présence d'autres plus prioritaire, par exemple selon une loi de Bernoulli (90% des cas suivre cet ordonnancement prioritaire et 10% des cas choisir aléatoirement parmi tous les queue). Aussi, cela nous invite à penser au moments où l'on a envie d'utiliser l'aléa pour casser cette priorisation discrète, et l'apprentissage automatique pourrai être une piste envisageable pour apprendre à distribuer les priorités aux threads.

3.4 Préemption

Nous avons également tenté d'implémenter l'ordonnancement basé sur la préemption mais par manque de temps, elle n'a pas abouti. Cette tentative peut être retrouvée dans le fichier *preemp_ thread.c* dans le répertoire *src* du dépôt. L'idée qu'on a eu a été de fixer un intervalle de temps régulier au bout de laquelle on forcerait le thread à passer la main. Pour ce faire, nous avons mis en place un timer (avec *setitimer*) qui envoie au bout du temps imparti un signal SIGALRM, signal dont le gestionnaire a été au préalable modifié avec *sigaction* afin d'appeler une fonction faisant un *thread_ yield* lors de la réception du signal. Une fois cette étape réalisée, il fallait correctement gérer la réception des signaux, chose qui était assez complexe et qui n'a pas pu être réalisé à temps. Il fallait notamment désactiver ces signaux lors que l'on réalisait des opérations délicates comme lors de la manipulation de la *runqueue* puis il fallait les réactiver au bon endroit du code.

4 Conclusion

Le projet nous a permis de mieux appréhender le fonctionnement de l'ordonnancement des threads mais également de mieux comprendre les changements de contextes. Nous avons également pu énormément expérimenter sur la façon d'implémenter nos threads et notre ordonnancement tout en comparant les différences de performances, les avantages et les inconvénients que cela pouvait provoquer. De plus, le fait de pouvoir comparer notre implémentation qui est en espace utilisateur avec la bibliothèque de thread standard pthread a été enrichissante.