Filière Informatique

Projet Système

Threads en Espace Utilisateur

Hind Boucherit
Antoine Boulangé
Julien Miens
Maxime Rosay

Encadrant :
Mathieu Faverge

Version du 17 Mai 2020

Introduction

Le projet consiste en la réalisation d'une bibliothèque permettant la création et la manipulation de threads en espace utilisateur. L'idée est d'obtenir de meilleures performances en utilisant notre librairie basée sur les contextes (espace utilisateur) plutôt que celle fournie par `pthreads`, basée sur les appels systèmes.

Ce rapport commence par présenter les fonctions de bases permettant de gérer des threads, puis il aborde également les notions de mutex, de préemption et enfin de calcul multicoeur.

1 Version de Base

Nous abordons dans cette première section les fonctionnalités de bases de la librairie, sans notion de mutex, de préemption ni de multicœur.

1.1 Initialisation de la librairie

La libraire possède plusieurs variables statiques, notamment la liste des threads ainsi qu'un pointeur vers le thread actif. Ainsi les threads ne changent pas de places dans la liste, ils sont ajoutés à la fin de celle-ci et supprimés sur place (voir détails plus loin).

La variable `active_thread`, déclarée en `static` dans le fichier `thread.c`, est un pointeur sur le thread en cours d'exécution. Elle nous permet de retrouver instantanément le thread courant. Cela nous permet de ne pas modifier la liste lorsqu'on change de contexte car on se contente de pointer ailleurs, à la différence d'une implémentation qui aurait, par exemple, toujours le thread actif en tête de file. Les seules modifications de la liste interviennent lors de la création d'un thread et de sa suppression.

La liste des threads est implémentée en utilisant une `LIST`, définie dans la librairie BSD, puisque cette structure de données nous permet d'ajouter facilement un élément en tête de liste et d'en supprimer un en temps constant (O(1)) quelle que soit sa position dans la liste. Dans la suite, on s'en sert comme une liste circulaire, cependant nous n'avons pas utilisé de liste circulaire (`CIRCLEQ`) de la librairie BSD car c'était fortement déconseillé par cette dernière.

Enfin, on initialise avant l'exécution du main un thread principal (correspondant au thread main) depuis un constructeur. Ce thread est très important puisqu'il permet de stocker le contexte actuel de l'exécution principale avant le lancement d'autres threads. Cela permet de traiter le thread principal comme tous les autres threads lors de l'ordonnancement.

1.2 Création d'un Thread

Pour créer un thread, une instance de la structure `thread_s` est allouée pour contenir le thread et sa pile. Il est alors possible de mettre en place un contexte qui exécutera la fonction fournie en paramètre.
Tous les threads ne vont pas appeler la fonction `thread_exit` à leur terminaison, c'est pourquoi nous devons forcer son appel si l'on souhaite pouvoir récupérer la valeur de retour et déclarer le thread comme terminé. Pour ce faire, nous avons une fonction `thread_runner` utilisée en tant que *wrapper*. Ainsi, le thread exécute la fonction `thread_runner` qui appelle `thread_exit` sur la fonction qu'il a reçu en paramètre lors de sa création. Cette fonction ayant été placée dans la structure du thread, cela permet de ne passer que le thread comme argument à la fonction `makecontext`. Il est nécessaire de rappeler que cette dernière permet la création d'un contexte à partir d'une fonction initiale qu'on lui donne en paramètre (ici `thread_runner`) avec ses arguments (ici le thread créé).

Enfin, le thread est inséré en tête de la liste des threads utilisateur à exécuter. Un `thread_yield` est effectué à la fin de la création, cela n'est pas nécessaire mais cela permet de donner la main à un autre thread, qui n'est pas forcément celui qui vient d'être créé, et ainsi distribuer la main de manière non linéaire pour faire progresser tous les threads utilisateurs.

1.3 Terminaison d'un Thread

Un thread se termine toujours par l'appel à `thread_exit`, soit lorsque l'utilisateur l'appelle lui-même, soit grâce au `thread_runner` défini précédemment (section 1.2). Cette fonction commence par mettre le statut de thread à `FINISHED` puis il le retire de la liste des threads en cours d'exécution. Il faut ensuite donner la main à un autre thread. Pour ce faire, la stratégie que nous utilisons est simplement de prendre le thread suivant dans la liste, ou de prendre le premier de la liste s'il était le dernier.

L'appel à `thread_exit` est géré de la même manière sur le main que sur n'importe quel autre thread, ce qui nous assure de ne pas y revenir. Cependant, si le programme se termine sur un autre thread que le main, cela signifie qu'on ne pourra pas libérer la pile de ce dernier thread directement (puisque nous n'avons pas encore changé de contexte). Il faut donc prévoir la libération des piles qui resterait en fin de programme. C'est pourquoi on a ajouté un contexte de fin. Ce contexte est appelé automatiquement par `thread_exit` lorsqu'elle détecte qu'il n'y a plus aucun thread à lancer, c'est-à-dire lorsque la liste est vide. Ce contexte ayant été au préalable alloué statiquement, il n'y a pas besoin de libérer sa pile. Son travail est alors de libérer ce qu'il reste du dernier thread actif, permettant de conclure le programme proprement.

Cette solution est plus simple que celle où on ajouterait un thread similaire à celui du main car il suffit d'avoir le contexte et il n'est pas dans la liste des threads ce qui permet de ne pas avoir à l'éviter lors d'un parcours de cette liste.

1.4 Join d'un Thread

Lorsqu'on appelle `join` sur un thread, on commence par vérifier le statut de ce thread, s'il n'est pas terminé on lui donne tout de suite la main pour qu'il puisse progresser. On ne fait rien tant qu'il n'est pas terminé (`FINISHED`). Seul un appel à `thread_exit` (par l'utilisateur, ou par le `thread_runner`) peut passer son statut à `FINISHED`, on réalise donc de l'attente active. Une fois cette attente terminée, on récupère la valeur de retour du thread, qui avait été stockée dans un champ de la structure, puis on libère toutes les données du thread. A la fin du `join`, le thread appelant continue sa propre exécution.

Remarque : tout thread non *joined* n'est pas libéré, sauf si le dernier thread actif n'est pas le main, alors la pile de ce thread uniquement sera libérée par le contexte de fin.

1.5 Passer la Main

En cas d'appel à la fonction `thread_yield()`, le thread doit permettre à un autre thread de s'exécuter à son tour. On choisit de donner la main au thread suivant dans la liste des threads, ou au premier si on est en fin de liste.

La figure 1 ci-dessous permet d'illustrer ce fonctionnement[1].

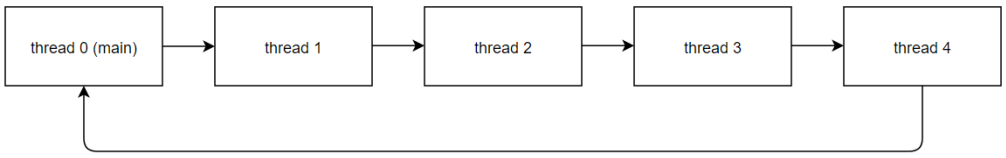

Figure 1: Ordonnancement

C'est ce même principe qui est utilisé à la fin de `thread_exit` (section 1.3) pour passer la main à la terminaison d'un thread.

[1] A noter que la flèche allant du thread 4 de la figure 1 au thread 0 n'existe pas réellement, le thread 4 n'ayant pas son champ `next` pointant sur le thread 0.

1.6 Structure de thread

Finalement, pour permettre la gestion des threads de la manière expliquée ci-avant (section 1), nous avons déduit différentes informations importantes. Voici donc la structure que nous avons choisi d'implémenter :

```
typedef struct thread_s {
  int status;
  ucontext_t uc;
  void *retval;
  int valgrind_stackid;
  void *(*func)(void *);
  void *funcarg;
  LIST_ENTRY(thread_s) next;
} thread_s;
```

- `status` peut prendre deux valeurs : `FINISHED` ou `NOT_FINISHED`. Ce champ permet de connaître le comportement à adopter lors de l'appel à la fonction `thread_join()` (cf. section 1.4). De plus, cet `int` permet de mieux gérer la terminaison des threads, pour les retirer de la liste une fois terminés par exemple.
- `retval` permet de stocker la valeur de retour du thread à sa terminaison. Ce champ est nécessaire car, lors de la libération de la pile en fin d'exécution, la valeur de retour n'est plus accessible. Cette valeur est donc placée dans ce champ afin de pouvoir y accéder ultérieurement.
- `func` correspond à la fonction à exécuter et `funcarg` à son paramètre associé. Ils sont notamment utilisés dans la fonction `thread_runner()` afin de lancer la fonction spécifiée (cf. section 1.2).
- `next` est utilisé pour la gestion de la liste chaînée et permet d'accéder à son élément suivant.
- `uc` permet de stocker la pile du thread.
- `valgrind_stackid` est l'identifiant de pile reconnu par Valgrind.

2 Mutex

Les mutex nous permettent d'éviter la concurrence sur la mémoire partagée entre plusieurs threads. Les mutex sont des verrous qui peuvent ainsi prendre deux états : disponible ou verrouillé.

Afin de mettre en oeuvre cette exclusion mutuelle, nous utilisons une structure `thread_mutex` ainsi que quatre fonctions principales permettant d'initialiser, détruire, verrouiller ou libérer un verrou.

2.1 Structure `thread_mutex`

Les informations nécessaires au bon fonctionnement d'un verrou sont au nombre de cinq :

```
typedef struct thread_mutex {
  unsigned short lock_init;
  unsigned short lock_destroyed;
  unsigned short lock_taken;
  thread_s* lock_owner;
  SIMPLEQ_HEAD(head_pt, pending_threads) lock_pendings;
} thread_mutex_t;
```

- `lock_init` est un booléen permettant de savoir si le verrou a bien été initialisé, condition nécessaire à son utilisation pour éviter tout comportement imprévisible ;
- `lock_destroyed` est un booléen permettant de savoir si le verrou a été détruit, puisqu'il n'est alors plus possible de l'utiliser ;
- `lock_taken` est également un booléen qui indique l'état dans lequel se trouve le verrou : verrouillé (true) ou disponible (false) ;
- `lock_owner`, pointeur vers un thread, sauvegarde l'identité du thread tenant le verrou ;
- `lock_pendings` est une file simple sauvegardant tous les threads en attente de la libération du verrou, c'est à dire les threads bloqués par le verrou.

2.2 Fonction de verrouillage

Une fonction `thread_mutex_lock()` est mise à disposition afin de pouvoir réserver l'accès à la mémoire partagée pendant une la zone critique. Celle-ci possède deux comportements.

Si le verrou est disponible, le thread appelant tient alors le verrou et s'exécute normalement. En revanche, si le verrou n'est pas disponible, le thread est ajouté à la fin de la file des threads bloqué par ce verrou et, tant que le verrou n'est pas disponible, la main est redonnée au thread tenant ce même verrou. L'attente est alors semi-active seulement, laissant ainsi la ressource processeur disponible au thread bloquant.

2.3 Fonction de libération

Une fonction `thread_mutex_unlock()` est également disponible pour libérer le verrou après la zone critique. Elle possède également deux comportements.

Si aucun thread n'est bloqué par ce verrou, alors le verrou est tout simplement libéré et est alors disponible. En revanche, si au moins un thread est bloqué, alors le premier thread de la file d'attente en est ôté et la main lui est donné. Cela permet notamment un certain ordonnancement des threads bloqués selon la méthode FIFO (premier arrivé, premier servi).

2.4 Performances

La conception des mutex en attente semi-active, détaillée ci-dessus, nous permet un gain non négligeable en terme de performance.

L'ancienne conception était en attente active. Lorsqu'un thread prenait la main et souhaitait accéder aux ressources partagées, celles-ci n'était parfois pas disponibles (à cause d'un verrou positionné par un autre thread). Dans ce cas là, le thread laissait la main grâce à `thread_yield()`. Si le thread suivant n'était pas non plus le propriétaire du verrou, la main était de nouveau laissée. Tous les threads bloqués s'exécutaient donc en attendant que la main soit redonnée au thread bloquant. Ce problème a été réglé entre autres en donnant la main directement au thread bloquant.

De plus, lors de la libération du verrou par un thread, la main était redonné à un autre thread grâce à `thread_yield()`. Afin d'assurer le bon fonctionnement du mutex et éviter qu'un thread reste bloqué plus que de raison, nous avons décidé de mettre en place une légère modification de l'ordonnancement. En effet, une liste ordonnée des threads bloqués est gardée afin de pouvoir donner la main au thread bloqué depuis le plus longtemps, s'il en existe un, lorsque le verrou est libéré.

3 Préemption

L'ordonnancement préemptif peut être utilisé afin d'assurer la bonne distribution du temps entre les différents threads. L'idée est donc d'interrompre une tâche lorsque l'exécution de celle-ci dépasse la durée qui lui a été accordée. Le thread en question sera donc suspendu et passera la main à d'autres threads dans la liste.
Dans notre implémentation, nous nous sommes intéressés à ce type d'ordonnancement, et nous avons mis en place une préemption à l'aide de `timers` et d'`alarmes`.

3.1 Création du timer

La création du timer dans cette section se résume à munir une variable *timer* de type `timer_t` d'une structure `itimerspec`. C'est à travers cette structure là que nous fixons les différents paramètres de démarrage et d'expiration du timer . On l'initialise comme suit :

```
1 struct itimerspec timer_spec;
2 timer_spec.it_interval.tv_nsec = WAITING_TIME;
3 timer_spec.it_value.tv_nsec = WAITING_TIME;
```

Le paramètre `WAITING_TIME` (ou `timeslice`) est la période du timer, ça sera la durée de temps écoulée entre chaque deux déclenchements d'un signal d'alarme. Le choix de ce paramètre ne doit bien sûr pas être arbitraire, il faut prendre en considération deux élément :

- Si l'intervalle de temps est trop long, on perd le principe de la préemption en laissant un thread spécifique exploiter trop longtemps le temps CPU, sans passer la main.
- Si trop court, le peu de temps alloué à un thread sera expiré suite aux changements de contexte, sans jamais que le thread puisse réellement effectuer sa tâche. Un peu comme accorder la parole à quelqu'un et l'interrompre au moment où il souhaite prononcer un mot !

Une `timeslice` raisonnable serait de l'ordre de `10-100ms`. Dans notre projet, nous avons opté pour une timeslice de 20ms.
Une fois que ces paramètres ont été fixés, nous faisons appel à la fonction `timer_setitimer` qui va nous permettre d'*armer* notre `timer` avec la structure `itimerspec`. Cela se fait comme suit :

```
timer_settime(timer, 0, &timer_spec, NULL);
```

C'est l'expiration du timer à des intervalles de temps réguliers qui nous permettra d'indiquer à notre programme que le thread actif a eu sa part de temps pour s'exécuter, et qu'il est temps de passer la main.
Lorsque le timer expire, un signal `SIGARLM` est envoyé au processus courant. Arrive donc après la gestion de ce signal.

3.2 Gestionnaire d'alarme

La fonction `sigaction` nous permet d'installer un gestionnaire de signal pour `SIGALRM`. On crée alors une structure `sigaction` que l'on nommera *action*, et qui sera un des paramètres passés à la fonction `sigaction`. L'initialisation de la structure et ses attributs se fait de la façon suivante :

```
1   struct sigaction action;
2   memset(&action, 0, sizeof(action));
3   sigemptyset(&action.sa_mask);
4   action.sa_handler = timer_handler;
```

Nous spécifions la fonction de gestion de signal à travers l'attribut `sa_handler` de la structure (ligne 4 ci-dessus). Maintenant, il faut relier notre *action* à une éventuelle réception de signal `SIGARLM`, c'est ici qu'intervient la fonction `sigaction` :

```
sigaction(SIGALRM, &action, NULL);
```

Le gestionnaire de signal `timer_handler()` est une fonction qui récupère le thread courant grâce à la fonction `thread_self()` de notre librairie et teste si ce dernier a déjà reçu le signal une fois, si c'est le cas, on fait appel à `thread_yield()` qui passe la main au thread suivant.
Afin de mieux observer ce qu'il se passe lorsqu'un thread est préempté, le test `71-preemption.c` a été mis en place. Son but est simplement de créer plusieurs thread sur une fonction `void * thfunc()` et d'afficher un message sur la sortie standard précisant le thread qui passe la main, et cela à chaque expiration du timer.

3.3 Bloquage de signaux dans le code

Puisque le signal d'alarme peut intervenir à n'importe quel moment et cela peut importe la tâche en cours, il est important de bloquer le signal d'alarme dans les sections critiques de notre code. Les sections critiques sont les blocs de code où nous modifions les principales ressources de notre programme, par exemple lorsqu'un thread est en cours de création, lorqu'il est ajouté à la `LIST` ou retiré de celle-ci : **nous ne devons pas être interrompus par un signal**.

Cela se fait en créant un masque de signaux `block` de type `siget_t` auquel nous ajoutons le signal `SIGALRM` grâce à la fonction `sigaddset`, comme suit : `sigaddset(&block, SIGALRM);`.
La fonction `sigprocmask` reçoit la structure `block` et un entier qui précise si les signaux présents dans ce masque sont à bloquer ou débloquer. La valeur de cet entier sera `SIG_BLOCK` ou `SIG_UNBLOCK` selon ce dont nous avons besoin :

```
sigprocmask(SIG_BLOCK, &block, NULL);
/* Section critique */
sigprocmask(SIG_UNBLOCK, &block, NULL);
```

Un signal bloqué n'est bien sûr pas ignoré. Le signal `SIGARLM`, si déclenché pendant qu'une section critique s'exécute, n'aura d'effet que lorsque le masque est débloqué.

4 Multicoeur

Le multicoeur est une notion clé lorsque l'on souhaite créer une librairie de threads. Elle permet d'exploiter au mieux les performances du processeur de sa machine pour paralléliser un code par exemple. Le parallélisme engendre de la concurrence et c'est ce qui sera discuté dans cette partie.

4.1 Mise en place de threads noyaux

L'idée ici est la mise en place de threads appelés *noyaux* qui seront lancés dès l'appel à notre librairie de threads, grâce à notre constructeur, et exécuteront l'entièreté des threads utilisateurs. Actuellement, ils sont au nombre de quatre[2] (le processus initial exécutant le code ainsi que trois nouveaux processus fils créés).
Ceci est résumé dans le schéma suivant (figure 2):

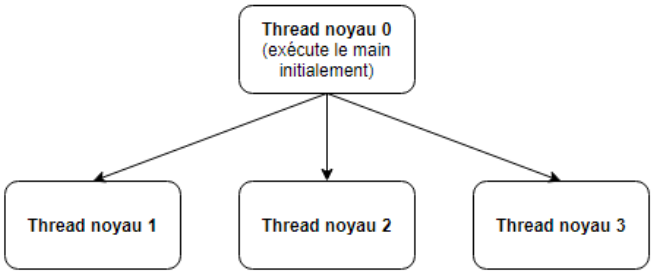

Figure 2: Architecture générale des threads noyaux

Comme précisé en début de cette sous-section, ces threads noyaux sont créés dans notre **constructeur** à l'aide de la fonction clone, ce qui donne le code suivant:

[2]Aujourd'hui la plupart des machines ont au minimum un processeur équipé de quatre cœurs

```
const int clone_flags = (CLONE_VM | CLONE_SIGHAND | CLONE_THREAD | SIGCHLD |
                         CLONE_SYSVSEM | CLONE_FILES | CLONE_FS | 0);
pid_t tid = clone(cloneFunc, stack_top, clone_flags, (void *)arg);
```

Les `flags`[3] utilisés sont semblables à ceux présents dans la libraire `pthread`, on fournit aussi un pointeur vers le haut de la pile qu'utilisera notre nouveau processus clone et à son lancement il exécutera la fonction `cloneFunc` avec pour argument `arg`.
Il faut aussi noter que les threads noyaux ont accès à de la mémoire partagée entre eux, comme par exemple la liste de threads utilisateurs non terminés: il leur faut ainsi un identifiant propre.
A leur création, on récupère les différents `tid`[4] des nouveaux processus retournés par la fonction `clone` et on associe à chaque `tid` un identifiant précis (allant de zéro à trois) dans un tableau global. Cela permet notamment le bon déroulement de la fonction swapcontext qui échange le contexte d'exécution courant du thread noyau en question avec un autre.
Enfin, avant de passer à la section suivante, rappelons les trois états principaux que peuvent prendre un processus (figure 3):

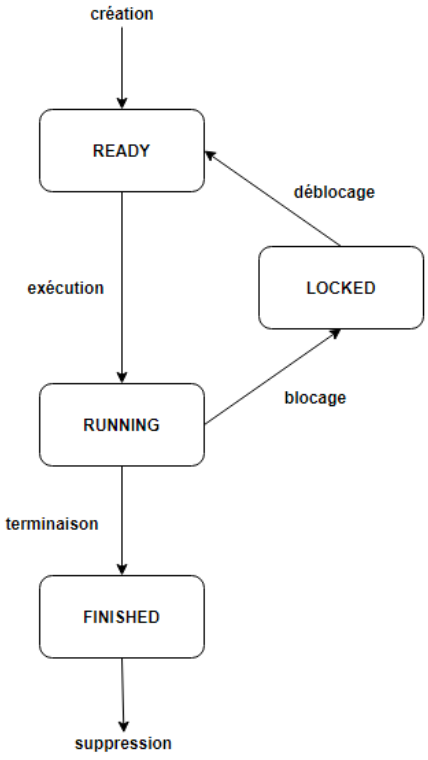

Figure 3: Diagramme d'état général d'un processus

Ainsi, chaque thread utilisateur alternera entre trois états[5] appelés `READY`, `RUNNING` et `LOCKED` (et ses variantes) dans la suite et auxquels s'ajoute l'état `FINISHED` précisant la terminaison d'un thread.

4.2 Gestion et distribution des threads utilisateurs

La gestion des threads utilisateurs est semblable à la version classique de notre librairie sans le multicœur. Cependant, c'est leur distribution qui est modifiée et nous alors voir comment dans

[3] Voir page 2 du man de clone pour les détails
[4] Thread id
[5] Contenus dans le champ status des threads utilisateurs

cette partie.
Abordons maintenant la fonction principale des threads noyaux: `cloneFunc`. Son pseudo-code est le suivant:

```
1  int cloneFunc(void * argument) {
2  /* Attente de la création de tous les threads noyaux */
3  /* Attendre tant qu'il n'y pas de thread utilisateur dans la liste */
4  /* Parcourir la liste indéfiniment jusqu'à trouver un thread utilisateur
5   * en état READY pour le faire progresser (état RUNNING dorénavant)
6   * Si la liste est vide entre temps, repartir à la ligne 3
7  */
8  }
```

Le pseudo-code ci-dessus met en évidence la manière dont les threads noyaux vont rechercher un thread utilisateur à exécuter.
Il y a tout d'abord une attente active sur la création, par le thread noyau zéro, des différents clones permettant d'assurer que les `id` de ces derniers sont bien initialisés. Une seconde attente active est effectuée sur la liste de threads utilisateur, si elle est vide, il n'y a rien à faire progresser et donc on attend.
Puis la fonction principale de recherche s'exécute de la manière suivante:

1. On récupère le thread utilisateur en tête de liste.
2. On regarde s'il est dans l'état `READY`.
   (a) Si oui, on le passe en état `RUNNING` et on actualise notre thread actif avec ce dernier.
   (b) Puis on échange son contexte courant avec celui du thread utilisateur trouvé.
   (c) Lorsque l'on arrive à cette étape, c'est que le thread noyau a terminé d'exécuter le thread utilisateur et donc il reprend sa recherche en partant du thread utilisateur suivant.
3. Sinon, on passe au thread utilisateur suivant dans la liste (s'il n'existe pas on repart en début de liste).
4. Enfin, on teste si la liste de threads utilisateurs non terminés est vide, si c'est le cas on sort de notre recherche et on repart au début de la fonction `cloneFunc`

Cela assure que chaque thread noyau possède sa variable `thread_active` correctement remplie s'il exécute un thread utilisateur et que celui-ci se trouve en état `RUNNING` et permet aussi un balayage constant de la liste de threads utilisateur.

Cependant, nous n'avons pas encore abordé les problèmes de concurrence présents lors de l'accès à la liste de threads utilisateurs ainsi que les changements de contextes. C'est ce que nous allons voir dans la partie suivante.

4.3 Techniques utilisées pour limiter les sections critiques

Cette sous-partie vise à présenter les techniques misent en place afin de limiter au mieux les problèmes de concurrence.

Le premier est lié aux accès et aux modifications de notre liste de threads utilisateur. Chaque thread noyau doit s'assurer qu'il est le seul à lire ou modifier des variables d'un thread de cette liste lors de sa recherche notamment. C'est pourquoi nous avons mis en place un système de sémaphore très simple: une sémaphore nommée `sem_list` contrôle les accès à cette liste.
Dans tout le code, tout type de modification est ainsi contrôlé par cette dernière empêchant strictement deux threads noyaux de s'attribuer le même thread utilisateur par exemple.

Un peu plus complexe, le deuxième problème relevé concerne les échanges de contextes avec la fonction `swapcontext`. Comme précisé plus haut et dans son manuel, cette fonction sauvegarde un contexte courant et met en place un autre contexte fourni en argument.

```
/* sauvegarde du contexte courant dans la variable contexte_actif
 * mise en place du prochain_contexte
*/
swapcontext(contexte_actif,prochain_contexte);
// on revient ici si on exécute à nouveau le contexte_actif !
```

Il faut impérativement assurer qu'aucun autre thread noyau ne puisse reprendre le contexte venant à peine d'être échangé, puisqu'on ne peut contrôler cette fonction à l'aide de sémaphore. Pour cela, nous avons utilisés le statut `LOCKED` décliné sous différents cas et traités directement dans la fonction `cloneFunc`.
Voici un extrait de cette dernière:

```
1  if (next_thread->status == READY) {
2       active_thread[clone_id] = next_thread;
3       active_thread[clone_id]->status = RUNNING;
4
5       /* on récupère son suivant pour notre retour*/
6       thread_after = LIST_NEXT(next_thread, next);
7       /* On relache la sémaphore puisqu'on a terminé nos manipulations */
8       assert(sem_post(&sem_queue) == 0);
9
10      swapcontext(&clone_uc[clone_id], &next_thread->uc);
11
12      /* On a terminé d'avancer ce thread, on est revenu */
13      active_thread[clone_id] = NULL;
14      /* Gestion des différents status LOCKED */
```

A partir de la ligne 8, tous les autres threads noyaux peuvent désormais accéder à la liste de threads utilisateur, mais ne peuvent exécuter le thread qui vient d'être pris.
Lorsqu'un thread noyau a terminé d'exécuter un thread utilisateur (ce dernier passe alors en état bloqué), le thread noyau se retrouve à la ligne 11, indique qu'il n'a plus de thread actif et entre, à la ligne 14, dans la gestion du statut du thread qu'il vient de bloquer.

Avant de présenter les différents cas de blocage, il faut introduire la nouvelle variable `before_thread` qui vient en complétion de `active_thread`. En effet, avant d'exécuter un changement de contexte, chaque thread noyau va bloquer puis stocker dans sa variable propre `before_thread` le thread qu'il était en train d'exécuter pour lui permettre de le débloquer ultérieurement (post changement).
Comme les changements de contexte sont présents à divers endroits du code, il existe différents cas:

- `LOCKED_BY_YIELD`: le thread utilisateur dans cet état n'est pas terminé mais a appelé `thread_yield` pour donner la main à un autre thread. Le thread noyau doit simplement le passer en statut `READY`.
- `LOCKED_BY_EXIT`: le thread utilisateur dans cet état est terminé, retiré de la liste de thread il faut donc que le thread noyau le passe en statut `FINISHED` pour qu'il soit libéré (sauf s'il s'agit du thread main).
- `LOCKED_BY_JOIN`: le thread utilisateur dans cet état a appelé la fonction `thread_join` pour attendre un autre thread utilisateur et le thread noyau a pu exécuter ce dernier. S'il termine ce thread attendu, il ré-exécutera le thread initial ayant appelé `thread_join`.

- `LOCKED_BY_ELSE_JOIN`: le thread utilisateur dans cet état a appelé la fonction `thread_join` pour attendre un autre thread utilisateur mais le thread noyau n'a pas pu l'exécuter (il est exécuté par un autre thread noyau) et donc la suite est la même que pour `LOCKED_BY_YIELD`.

En réalité, il existe un cas qui est implémenté par souci de cohérence, mais qui n'arrivera jamais en pratique: il s'agit du `LOCKED_BY_JOIN`. Tout d'abord, il est important de préciser aussi qu'avant de stocker le thread courant dans sa variable `before_thread`, le thread noyau va au préalable vérifier si elle est vide, sinon il débloque le thread stocké pour y placer le nouveau.
Ainsi, le cas du `LOCKED_BY_JOIN` et la reprise du thread ayant appelé `thread_join` ne sera pas effectuée directement par le thread noyau initial. Le thread dans cet état repassera simplement en statut `READY` et sera exécuté par un thread noyau quelconque. En effet, pour revenir au thread utilisateur ayant appelé `thread_join` il faut impérativement que le thread attendu soit passé par la fonction `thread_exit` et donc remplace la variable stockée dans `before_thread`.

4.4 Analyse des performances

Les courbes suivantes ont été générées sur une machine virtuelle exécutant Ubuntu version 19.10 avec un processeur Intel(R) Core(TM) i7-8750H hexacore.

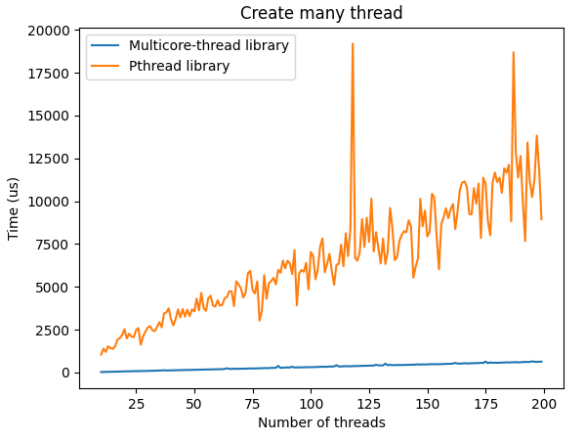

Figure 4: test create-many

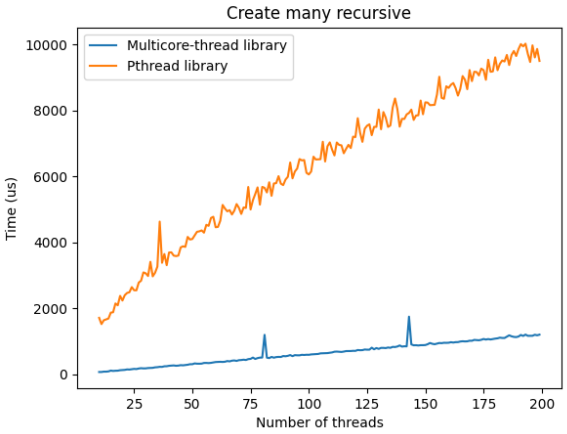

Figure 5: test create-many-recursive

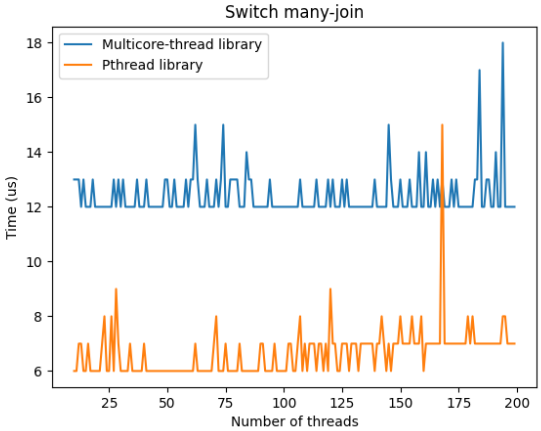

Figure 6: test switch-many-join

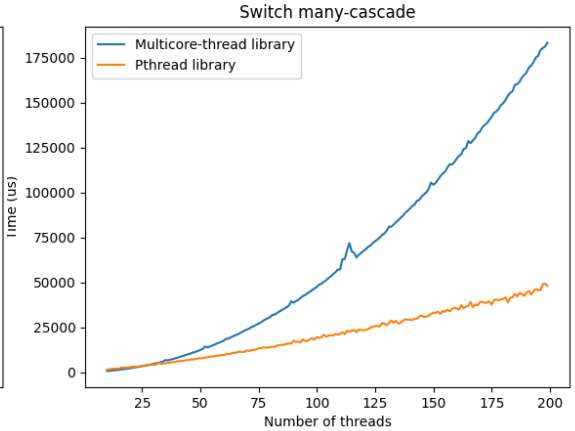

Figure 7: test switch-many-cascade

Les figures 4 et 5 montrent exactement pourquoi nous avons créé une librairie de threads utilisateurs: pour y gagner en performance. On remarque que ces deux tests sont sur la création de

beaucoup de threads de deux manières différentes: séquentielle et récursive.
Dans les deux cas, nous sommes meilleurs puisque comparé à `pthread` notre création est beaucoup moins coûteuse dû au fait qu'on ne recrée pas un processus pour chaque thread, mais que nous passons simplement par une liste. De plus, dans cette version multicœur, seuls quatre processus coexistent donc le système *joue* seulement avec ces quatre derniers et non avec un nombre linéairement dépendant du nombre de threads qu'on souhaite créer.

La figure 6 montre simplement que notre version est similaire à `pthread` pour ce test, cependant cela démontre juste qu'il y a quelques optimisations possibles.
Cependant, la figure 7 montre une des limites de notre implémentation multicoeur: la distribution des threads utilisateurs.
En effet, actuellement à chaque ajout/retrait d'un élément dans la liste des threads utilisateur, tous les threads noyaux repartent en début de liste, s'ils recherchent activement un thread à exécuter, pour minimiser les problèmes de concurrence sur cette dernière.
De plus, si un thread noyau fait une chaîne récursive de `thread_join`, il ne remontera jamais cette chaîne mais reprendra son comportement initial une fois arrivé en bout en de chaîne. Cependant, le coût est amoindri puisque chaque thread utilisateur nouveau est ajouté en début de liste (voir section 1.2).

5 Tests et Performances

L'objectif de ce projet étant non seulement d'implémenter une librairie de threads en espace utilisateur, mais aussi d'essayer de conserver une complexité des plus satisfaisantes possibles, afin de nous assurer de bonnes performances. Nous allons donc dans cette section comparer notre implémentation avec celle existante, `pthreads`.

5.1 Courbes

Les courbes dans cette section ont été générées sur une machine avec un processeur Intel(R) Core(TM) i5-8250U quadcore sous Ubuntu 16.04.

Les figures ci-dessous (Figures 2, 3, 4, 5 et 6) comparent les performances de notre librairie de threads avec la librairie *pthreads* en faisant varier les arguments passés aux tests. On peut alors observer la différence de temps d'exécution, en microsecondes, entre ces deux librairies. A noter que chacun des points des courbes correspond à une moyenne temporelle sur 10 exécutions.

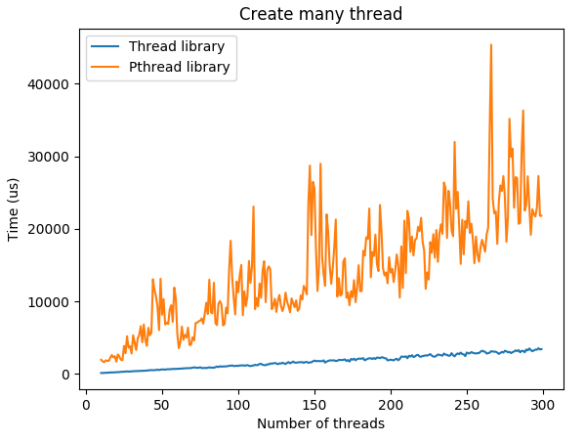

Figure 8: create-many

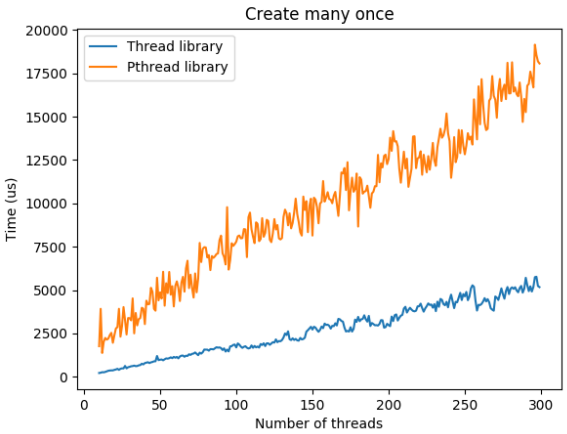

Figure 9: create-many-once

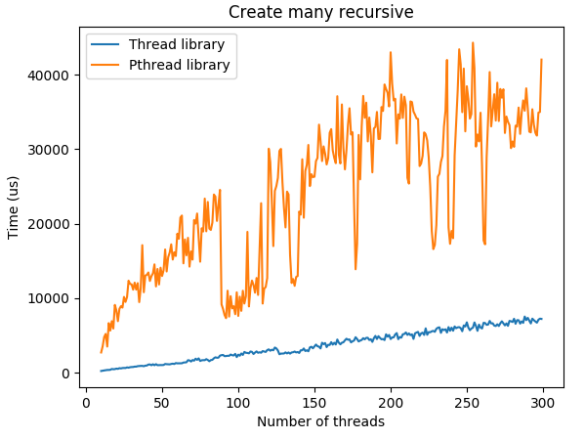

Figure 10: create-many-recursive

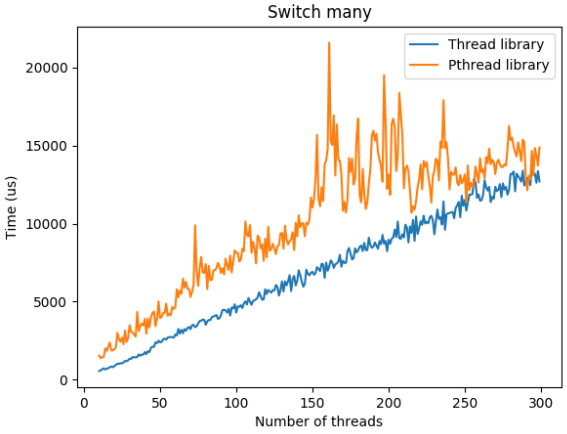

Figure 11: switch-many

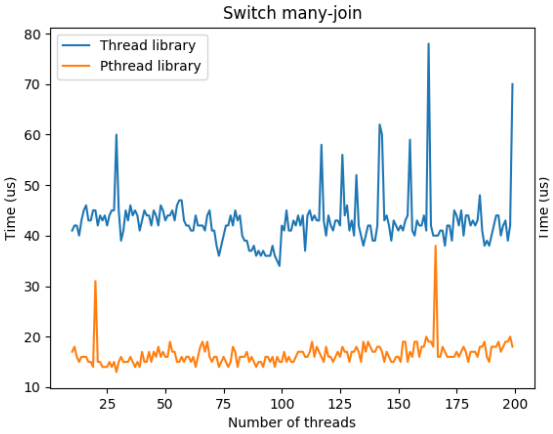

Figure 12: switch-many-join

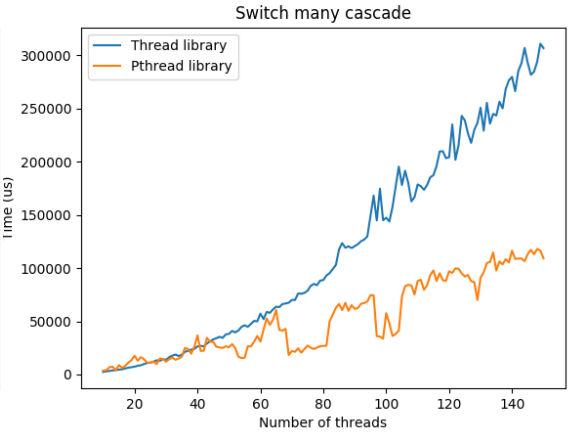

Figure 13: switch-many-cascade

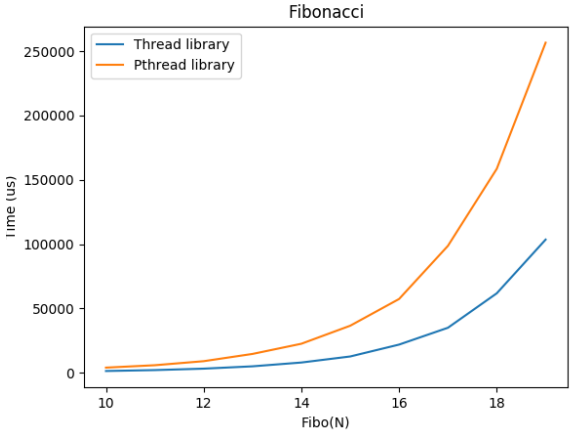

Figure 14: fibonacci

5.2 Interprétation

Nous remarquons sur la plupart des courbes que nous avons un comportement asymptotiques similaires à `pthreads`, et vu que nous restons en espace utilisateur, la librairie `thread` gagne en performance contre `pthread`. On voit bien la linéarité des courbes sur les figures 8, 9, 10 et 11.

La figure 13 indique une forte croissance de la courbe de temps en fonction du nombre de threads, cela est dû à l'attente active faite dans le join.

Pour le test `51-fibonacci`, figure 14, la différence est très nette. La courbe indique que lors du calcul du 20ème élément de cette suite, la librairie `pthreads` explose en complexité temporelle. Les deux courbes pouvant être observées sont exponentielles, ce qui est logique vu qu'il n'y a aucune conservation des valeurs calculées pour aboutir au résultat final.

De plus, à partir de l'élément 21 pour la plupart des machines, `pthreads` se voit limité en terme de ressources mémoires disponibles ainsi que par le nombre de threads maximal pouvant être créés. L'avantage de notre librairie prend ici tout son sens. En effet, nous disposons, en espace utilisateur, de beaucoup plus de ressources et pouvons donc pousser plus loin le calcul de cette suite[6].

6 Conclusion

Ce projet accomplit son objectif principal, il fournit une alternative à `pthread` mais en espace utilisateur, donnant au passage de meilleures performances dans certains cas (*cf section 5*). Cependant, nous sommes conscients que notre librairie aurait pu être améliorée à certains endroits. Les axes d'amélioration les plus importants sont les suivants :

- Utiliser de l'attente passive dans certaines zones de notre code, dans la fonction `join` par exemple.
- Nous aurions aussi pu mettre en place un système de priorité dans notre implémentation, permettant ainsi à un thread prioritaire de s'exécuter avant ou plus souvent que les threads moins prioritaires. Cela aurait été utile, notamment en ce qui concerne les mutex, afin de donner la priorité aux threads bloqués par un verrou. Faute de temps, nous n'avons pas pu explorer plus en profondeur cette option.

Finalement, notre équipe est assez satisfaite du travail accompli et de l'implication de chacun de ses membres et ce, depuis le début du projet. Les conditions dans lesquelles nous avons travaillé étaient assez difficiles, mais nous estimons que nous avons bien relevé le défi.
Ce projet nous a permis de mieux nous familiariser avec les différents concepts vus en programmation système et de les mettre en application à travers l'implémentation de cette libraire. En ce qui concerne la gestion de ce projet, nous avons pu assurer une bonne communication interne, un bon rythme de travail et un bon avancement global du projet malgré les circonstances exceptionnelles que nous avons vécues.

[6] en échange d'une partie de notre temps libre tout de même pour le moment pour le temps d'exécution...