Systèmes d'exploitation

Rapport final de projet

Johan Chataigner
Alexandre Disdier
Emeric Duchemin
Laurent Genty
Mehdi Naciri

Dépôt Thor n° 9118

Sous la responsabilité pédagogique de
Philippe Swartvagher

Année 2019-2020 - Semestre 8

1 Objectifs du rapport et état du projet

Le présent rapport propose de s'intéresser aux choix que nous avons adoptés afin de modéliser notre bibliothèque de *threads*, de présenter ce qui a été accompli, les difficultés que nous avons rencontrées et les approches ayant été mises en oeuvre pour les surmonter.

A mi-parcours du projet, notre bibliothèque était coopérative, fonctionnelle et implémentait une première version des mutexes. Plusieurs objectifs avancés ont été accomplis ou partiellement accomplis depuis, parmi lesquels l'ajout de la préemption, l'amélioration des mutexes pour supprimer les attentes actives inutiles, la détection des débordements de pile et un début d'étude de différentes politiques d'ordonnancement et de priorisation dynamique des threads. Nous avons compilé tous ces ajouts et dénombrons au total quatre versions principales de la bibliothèque, chacune présente dans un dossier de l'archive finale `rendu_final.tar.gz` à la racine du dépôt (et chacune correspondant à une branche git du dépôt). Le rapport confrontera ces versions successives en comparant leurs performances respectives.

2 Structure de l'archive rendue

L'archive `rendu_final.tar.gz` à la racine du dépôt contient quatre dossiers, chacun contenant une version différente de l'état de notre application. La structure de chacun des dossiers est la même (à quelques fichiers sources prêt), à savoir qu'à la racine de chaque dossier, les commandes du Makefile spécifiées dans les consignes du sujet sont de vigueur (`make` , `make check` , `make valgrind` ...). Seule légère divergence, la commande `make graphs` génère les graphes de performance et les place dans le dossier `graphs/imgs` et `make plotting` génère les graphes mais seulement à partir des données déjà générées par un appel préalable à `make graphs` , permettant ainsi d'éviter de tout recalculer étant donné la durée de la génération.

Description des quatre versions

- `cooperative_scheduling` (`branche rendu_mi_rapport`) : Il s'agit de la version rendue à mi parcours du projet. Elle implémente la bibliothèque de base, optimisée, et dont l'ordonnancement se fait uniquement de manière coopérative par le biais d'appels à `thread_yield` .
- `preemptive_scheduling` (`branche preemption`) : c'est la seconde version de la bibliothèque, elle introduit la préemption au moyen d'émissions de SIGPROF, correctement coordonnées avec les appels à `thread_yield` .
- `preemption_with_stack_overflow` (`branches master et stack_overflow`) : C'est la principale version du programme. Elle reprend la version préemptive, mais ajoute la détection des débordements de piles ainsi qu'un système de pool de piles disponibles.
- `red_black_tree_runqueue` (`branche red_black_runqueue`) : C'est la dernière version développée. C'est une version un peu plus expérimentale, inachevée dans le sens où les performances sont faibles par rapport à la version précédente. Elle a pour but d'insuffler des priorités dynamiques aux threads en implémentant la file non plus sous forme de file, mais sous forme d'arbre rouge-noir trié en fonction du temps CPU consommé par les threads. Bien que relativement lente, elle reste fonctionnelle.

3 Architecture de la bibliothèque à mi-parcours

La version `cooperative_scheduling` représente l'état de l'application telle qu'elle était au moment du rendu de mi-parcours, à l'exception de la gestion des *mutexes* : ceux-ci ont en effet été légèrement modifiés pour supprimer une attente active coûteuse et inutile. Si l'on excepte cette amélioration qui sera détaillée dans la suite du rapport, l'organisation de la bibliothèque dans cette version est la suivante :

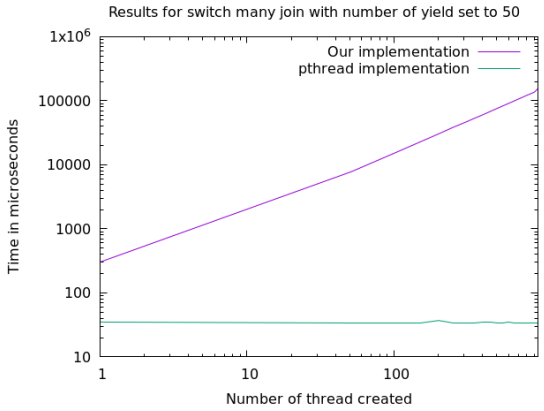
- La file d'attente de *threads* (ou *runqueue*) est une file circulaire. Un pointeur `current_thread` pointe sur le *thread* courant en cours d'exécution.
- L'identifiant de *thread* `thread_t` est un pointeur sur la structure représentant un *thread*.
- Avant même l'exécution de la fonction `main` et grâce au constructeur `master_context` , le programme initialise la *runqueue* et y insère un premier *thread* qui se charge d'exécuter la fonction `main` .
- Chaque *thread* est, en réalité, associé à la fonction `run_thread` . Elle fonctionne en deux parties : elle exécute d'abord la fonction qui a été associée au main par le biais de la méthode `thread_create` . Elle appelle ensuite la fonction `thread_exit` au nom de ce même *thread*. Cette façon de faire permet de s'assurer que tous les *threads* exécutent avant leur terminaison la fonction `thread_exit` , même si les fonctions qui leur sont associées ne l'appellent pas de manière explicite.
- L'ordonnancement des processus est la plus simple qui soit et s'inspire des politiques *First Come First Serve* et *Round Robin*. Ainsi, lorsque le *thread* courant est mis en attente par la commande `thread_yield` , `current_thread` pointe sur le thread suivant dans la file.

Durant la phase de développement de cette première version de la bibliothèque, l'accent a été mis sur les performances, et celles-ci étaient parfois bien moins bonnes que celles de la librairie *pthreads*. L'amélioration la plus significative avait alors consisté en la limitation du nombre d'appels de la méthode `thread_yield` . La méthode `thread_join` en particulier faisait une attente active lorsqu'elle devait attendre la terminaison d'un *thread*, en rendant la main à chaque fois que le *thread* en question s'avérait ne pas encore avoir retourné. Dans le cas où un nombre significatif de *threads* étaient entrés dans cette phase d'attente active, ils allaient devoir se passer la main à tour de rôle jusqu'à ce que potentiellement le seul *thread* encore en activité puisse enfin s'exécuter. Nous gâchions un temps CPU considérable, et le test `switch_many_join` a été le test dans lequel les effets de cette perte s'étaient le plus fait ressentir.

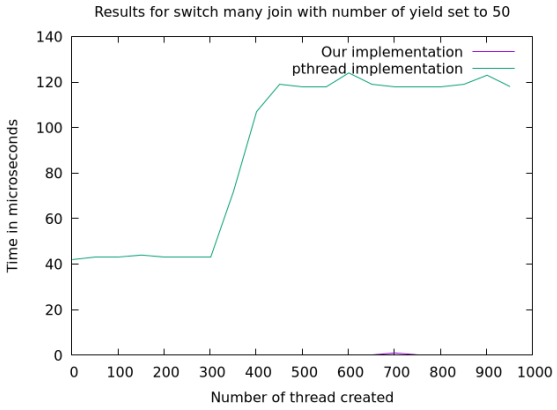
Une solution pour limiter le nombre de *threads* en état de "join" a été de retirer ces *threads* de la *runqueue* et de les ajouter dans une file propre au *thread* dont on attend l'exécution (nommée `joining_threads`). Lorsque ce dernier se termine (autrement dit lorsqu'il exécute la fonction `thread_exit`), il défile toute sa file de *threads* en état de "join" et les replace dans la *runqueue*. Les effets sur les performances de la suppression de l'attente active est illustrée en figure 1.

Correction des erreurs mémoire avec Valgrind

Un des points les plus délicats à gérer a été la correction des fuites mémoires et des accès invalides. Chaque *thread*, au moment du *join*, libère les ressources associées au *thread* en cours de terminaison. Ces ressources se composent du contexte du *thread*, de la structure *thread* en elle-même et de la pile allouée spécifiquement pour le *thread*. Cette dernière libération en particulier était la plus problématique : lorsque le dernier thread se termine, quelqu'un doit libérer les ressources lui étant associées. Or, si cette entité libératrice est elle-même un *thread*, quelqu'un doit également libérer ses ressources, et ainsi de suite. Une solution était alors d'inciter le dernier *thread* à libérer lui-même ses propres ressources, mais en contrepartie les dernières instructions qui suivent la libération de la pile causent alors des problèmes d'accès invalides (erreur de type `Invalid write of size 8...` de Valgrind). Il faut donc que le dernier contexte à qui l'on passe la main dispose d'une pile non pas allouée dynamiquement dans le tas, mais d'une pile correspondant en réalité à la "vraie" pile principale du processus.



(a) Avec attente active



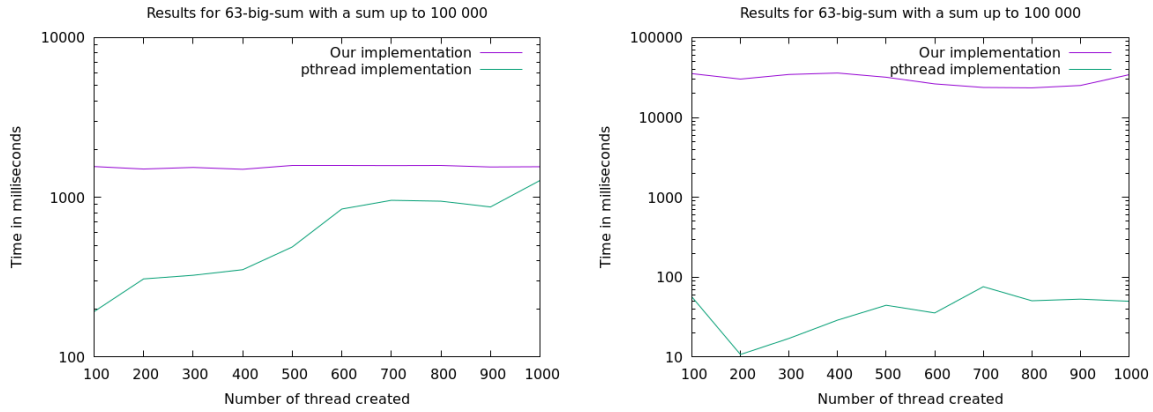
(b) Sans attente active (notez le petit bond violet autour de 700 threads)

FIGURE 1 – Effets de la suppression de l'attente active dans join (Test 32 *switch_many_join*, nombre de *yield* fixé à 50)

La solution que nous avons donc choisie pour pallier ce problème est de sauvegarder globalement le contexte principal du processus juste avant l'exécution du *thread* correspondant à la fonction `main` . A la fin de l'exécution du programme, le dernier *thread* sur le point de se terminer rend la main à ce contexte et passe son propre pointeur par le biais de la structure `parent_context` , de façon à ce que ses ressources soient libérées au même titre que tous ses prédécesseurs.

4 Suppression de l'attente active des *mutexes*

Dans la version de notre bibliothèque telle que nous l'avons laissée à mi-parcours, les *mutexes* souffraient du même défaut que les *threads* qui rendaient indéfiniment la main lors d'un appel à `thread_join` . Ainsi, les *threads* voulant acquérir le verrou d'un *mutex* étaient bloqués dans une attente active du relâchement du verrou. Or, en laissant les processus voulant acquérir un *lock* dans la *runqueue*, nous perdions du temps à leur laisser la main à chaque fois sans qu'ils puissent effectuer de travail utile. La solution que nous avons mise en oeuvre est donc analogue à celle des *join* et consiste à mettre en place une file de stockage locale à chaque *mutex* (nommée `waiting_threads`), ce qui permet de maintenir ces processus hors d'atteinte de l'ordonnanceur. Une fois que le *mutex* est libéré, le premier élément de cette file est remis dans la *runqueue*. Les performances obtenues sur la version coopérative après application de cette amélioration sont présentées en figure 2.



(a) Avec attente active

(b) Sans attente active

FIGURE 2 – Effet de la suppression de l'attente active dans lock (Test 63 *big-sum*, somme de 100000 éléments)

5 Ajout de la préemption

Après avoir achevé la version coopérative, nous nous sommes attelés à la préemption qui semble être l'un des objectifs avancés les plus importants. Nous devions donc mettre en place un ordonnanceur préemptif invoqué à intervalles réguliers (les *timeslices*) pour passer la main au *thread* suivant dans la *runqueue*, et ce sans compromettre le comportement coopératif déjà en place. L'enjeu est ensuite de tester plusieurs valeurs successives de la durée de cette *timeslice* afin de disposer des meilleures performances possibles.

La dossier `preemptive_scheduling` contient la version de la bibliothèque qui apporte la préemption au moyen de timers et de gestionnaires de signaux. Nous avions le choix entre trois types de signaux pour déclencher un changement de contexte : SIGALRM, SIGPROF et SIGVTALARM. L'appel système `setitimer(2)` prend en charge ces trois signaux. La différence réside dans le fait que les SIGALRM sont déclenchés à chaque écoulement du timer selon le temps réel écoulé, les SIGPROF selon le temps écoulé au sein du programme appelant et les SIGVTALARM selon le temps écoulé au sein du programme appelant uniquement en espace utilisateur. Sachant que notre programme tourne déjà en espace utilisateur, et que le processus entier est lui-même susceptible d'être fortement ordonnancé par le système d'exploitation, il paraît superflu d'utiliser le timer réel. En revanche, une fois que l'on considère uniquement le temps CPU utilisé par le processus, il nous semble raisonnable de prendre en compte le temps

passé dans les appels système dans le décompte, pour ne pas trop donner d'importance à des *threads* dans lesquels les appels systèmes sont fréquents. Nous utilisons donc des signaux SIGPROF, et mettons en place dès le début de l'exécution de la bibliothèque un gestionnaire de signaux (`scheduler_handler`) à l'aide de `sigaction(2)` .

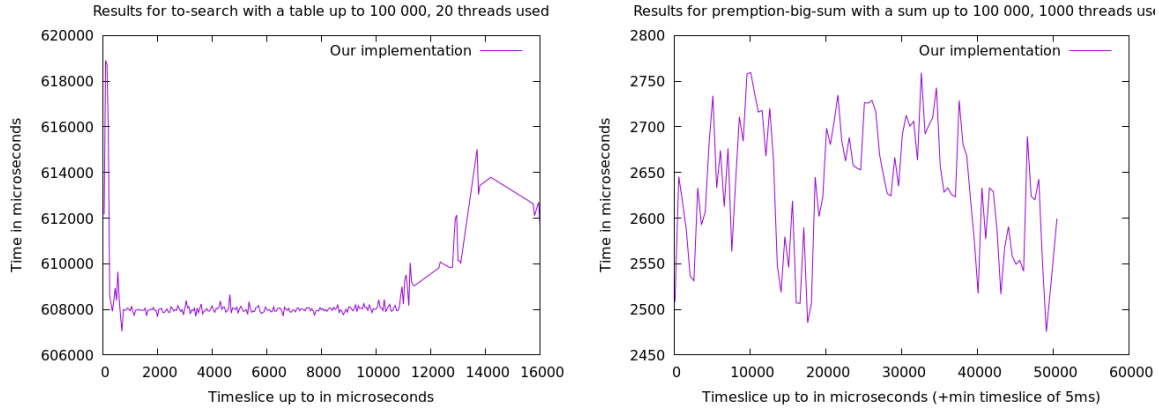
Cet handler se charge de passer la main au contexte de l'ordonnanceur, dont la tâche est de charger le *thread* suivant de la *runqueue* vers le pointeur global `current_thread` selon la politique d'ordonnancement en vigueur avant de basculer sur le contexte de ce dernier (les politiques d'ordonnancement sont discutées en partie 7). Pour simplifier et pour éviter au maximum des préemptions incongrues, toutes les fonctions de la bibliothèque (y compris celles liées aux *mutexes*) commencent par une désactivation de l'ordonnanceur préemptif, et terminent par sa réactivation.

Pour assurer une compatibilité avec la version coopérative, les *threads* ne donnent plus directement la main au *thread* suivant par le biais de `setcontext` ou `swapcontext` , mais invoquent le plus possible le contexte de l'ordonnanceur qui se charge de la transition. Il s'agit de factoriser le code de changement de *thread* et d'éviter des incohérences quant à l'état de la *runqueue*. De fait, dans cette version et dans les versions futures, tous les changements de contexte sont effectués par l'ordonnanceur (à l'exception du changement de contexte vers le main au début et le changement vers le *master context* pour libérer les ressources à la fin). Il est possible de désactiver la préemption en assignant 0 à la macro USE_PREEMPTIVE, mais les appels à `thread_yield` continueront de passer la main au scheduler pour effectuer le changement de contexte en leur nom. La multiplication des changements de contexte qui en résulte entraîne inévitablement une baisse des performances globales de l'application.

La durée de la timeslice peut avoir un impact significatif sur les performances du programme. Le graphe 3a a été réalisé sur un algorithme dans lequel la préemption présente un avantage. Dans ce programme de test, nous recherchons un élément dans un tableau à une place bien précise (dans notre exemple, c'est le *thread* 9 qui le trouvera après avoir parcouru un quart de l'intervalle de données qui lui est affecté). Ainsi, si le timer est judicieusement défini, le parcours du tableau sera suffisamment efficace pour que peu de temps soit passé sur des *threads* inutiles et pour que le temps alloué aux *thread* 9 soit maximisé. On se rend compte alors que si la timeslice est trop petite, alors le programme passe trop de temps à changer de contexte sans qu'aucun travail utile n'ait le temps d'être accompli. A l'inverse, si la timeslice est trop grande, la préemption n'est opérée que très peu fréquemment et on en revient à perdre du temps à parcourir des zones du tableau qui ne contiennent pas l'entier recherché. La figure 3a illustre bien cette forme en V caractéristique des faibles performances pour des valeurs de timeslice trop extrêmes.

Nous avons aussi testé la préemption sur un exemple où elle ne semble pas avoir d'intérêt particulier. Nous évaluons le temps pris pour plusieurs valeurs de timeslice différentes sur la réalisation de la somme des entiers de 1 à 100 000 par 1000 *threads*. Le graphe de la figure 3b nous montre que l'effet de la timeslice est beaucoup moins évident dans ce genre de cas. Ce résultat était prévisible, car chaque *thread* doit effectuer un travail indépendamment de celui des autres. Ainsi, rajouter de la préemption n'amène qu'une surcharge de changements de contexte coûteuse en temps et ce peut importe la valeur de la timeslice non extrêmale. Si l'on exclut le bruit, on remarque bien la descente de la courbe aux environ d'une timeslice égale à 35000, qui doit par conséquent être le temps au-delà duquel les préemptions sont trop rares pour que le programme les intercepte avant sa terminaison.

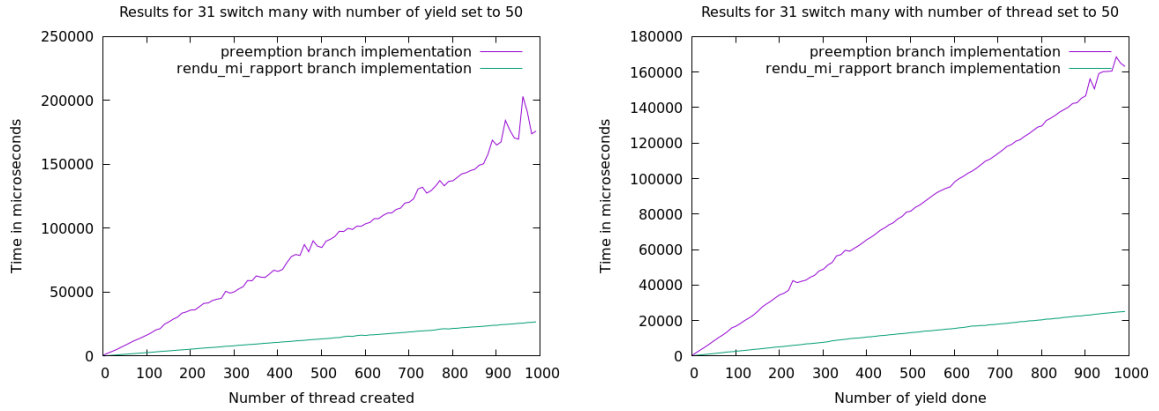
Nous avons naturellement cherché à confronter les différences de performances entre notre version coopérative et cette nouvelle version préemptive. Sur la figure 4, nous pouvons voir un exemple dans lequel la préemption a un effet sur les performances. À mesure que l'on augmente le nombre de *threads* et le nombre de yields, l'écart de performances croît en faveur de la version coopérative, ce qui est expliqué par le surcoût propre aux changements de contexte. La figure 5 offre une nouvelle comparaison entre les deux versions sur les tests 32 et 63.



(a) Test *to-search* (recherche d'un entier dans un tableau), préemption utile

(b) 63 *big-sum* (somme des entiers jusqu'à 100 000), préemption gadget

FIGURE 3 – Effets de la variation de la valeur de la timeslice sur deux tests différents (*to-search* et 63 *big-sum*)



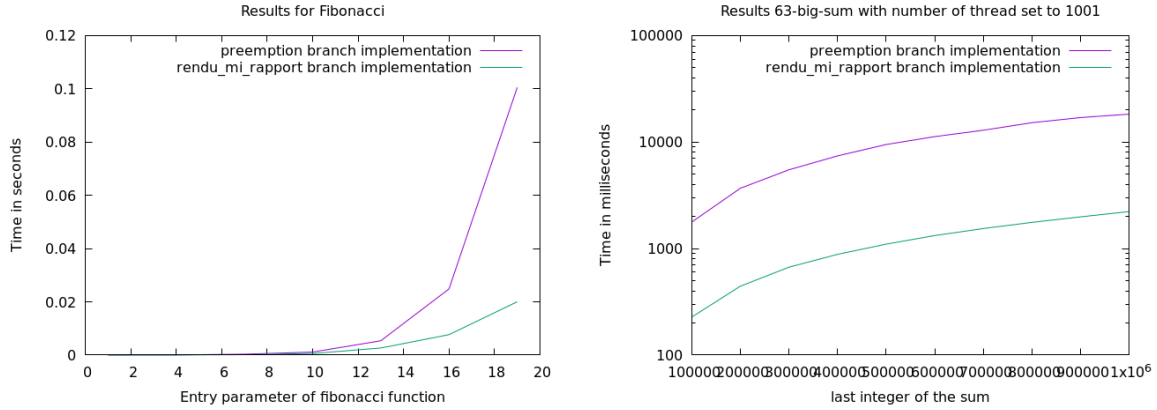
(a) Comparaison sur le test 31 entre la version coopérative et la version préemptive (nombre de yields fixé)

(b) Comparaison sur le test 31 entre la version coopérative et la version préemptive (nombre de threads fixé)

FIGURE 4 – Effets de la préemption sur le test 31 *switch-many*

6 Détection des débordements de piles et pool de piles

Nous avons mis en place un système de détection de débordements de pile. L'idée est de protéger la dernière page allouée de la pile d'un *thread* en écriture, lecture et exécution. L'appel système `mprotect(2)` permet de mettre en place une telle protection, mais nécessite que l'adresse de début de la zone que l'on souhaite protéger soit alignée sur un début de page mémoire. `malloc` n'est plus satisfaisant, il faut alors utiliser un service tel que `valloc(3)`, `memalign(3)` (tous les deux obsolètes), `posix_memalign(3)` voire allouer la pile via `mmap(2)` directement. La pile est l'espace d'allocation automatique d'un *thread* et il ne doit pas accéder à une zone mémoire autre que la sienne, auquel cas il pourrait endommager la mémoire des autres *threads*. `mprotect` en lui-même est relativement simple à mettre en oeuvre et provoque un envoi de signal SIGSEGV en cas d'infraction aux droits assignés à la page protégée. La page protégée est la première de la zone allouée, puisque le sens d'allocation



(a) Comparaison sur le test 51 entre la version coopérative et la version préemptive

(b) Comparaison sur le test 63 entre la version coopérative et la version préemptive (nombre de threads fixé)

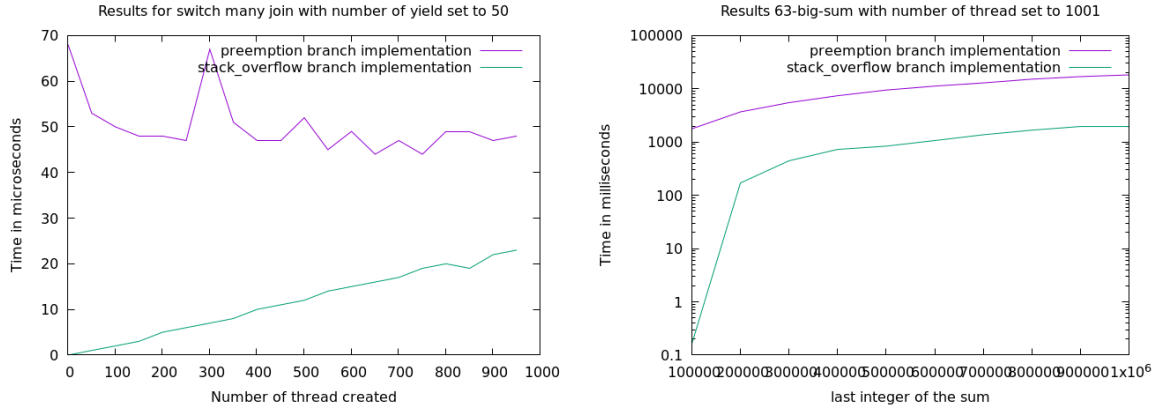
FIGURE 5 – Effets de la préemption sur les tests 51 *fibonnacci* et 63 *big-sum*

dans une pile est le sens descendant (le pointeur de pile décroît au fur et à mesure des empilements). Un simple handler de SIGSEGV capture les débordements et met fin au *thread* fautif en appelant `thread_exit` en son nom sur une pile dédiée définie par `sigaltstack(2)` .

Cette implémentation pose un problème. Le test de Fibonnacci a révélé notamment que le programme s'arrêtait avec un paramètre supérieur à 14. Le nombre important de création de *threads* entraînait en réalité un surnombre d'appels à `mprotect` , et par transitivité à `mmap` , dont le nombre d'appels total au cours de l'exécution d'un processus est limité par la valeur contenue dans le fichier `/proc/sys/vm/max_map_count` . Une solution, qui aurait d'ailleurs pu être mise en oeuvre avant même de s'intéresser aux débordements de piles, est de réutiliser les piles des *threads* ayant terminé. Il suffit de gérer une file de piles (la structure `stackpool`), dans laquelle on insère les piles de *threads* en phase de terminaison, et dont on récupère les piles pour les assigner aux *threads* nouvellement créés. Si aucune pile n'est disponible dans la *stackpool*, nous la créons de façon classique avec `mmap` (`valloc` et autres utilitaires d'allocation de plus haut niveau semblent favoriser les conflits avec `mprotect` , d'où l'utilisation de `mmap` qui ne pose aucun problème mémoire à l'exécution). Il faut bien entendu penser à libérer toutes les piles restantes avant la terminaison du programme.

Lorsque l'on sonde le nombre de *piles* créées durant les tests, le nombre d'allocations est effectivement amoindri. A titre d'exemple, sans cette file de piles, le test 21 *create-many* alloue mille piles pour mille *threads*, là où l'introduction de la *stackpool* lui permet de n'en allouer que deux durant toute la durée d'exécution du programme.

Nous avons comparé les performances de notre bibliothèque avec ou sans les pools de *threads* sur deux tests. La figure 6a montre que la différence de temps d'exécution peut être négligeable dans le cas où les piles sont peu recyclées, comme dans le test 32. Sur le test 63 illustré par la figure 6b, en revanche, l'écart est un peu plus significatif, bien que relativement négligeable, ce qui traduit une amélioration de l'espace mémoire utilisée plus que du temps économisé.



(a) Comparaison sur le test 32 entre la version préemptive et la version avec débordements (nombre de yields fixé)

(b) Comparaison sur le test 63 entre la version préemptive et la version avec débordements (nombre de yields fixé)

FIGURE 6 – Effets de la stackpool sur les tests 32 *switch-many-join* et 63 *big-sum*

7 Politiques d'ordonnancement

Jusqu'à présent, les versions successives de la bibliothèque se basaient sur un simple ordonnancement en Round Robin, dans lequel chaque *thread* était inséré de façon circulaire dans la *runqueue* et se voyait attribuer la main s'il était le dernier à l'avoir reçue. Il y a dans la bibliothèque au moins trois manières d'influencer l'ordre dans lequel les *threads* sont exécutés :

1. Donner la main à un *thread* dès sa création, sans attendre que le *thread* créateur lui passe la main explicitement ou soit préempté ;
2. Réinsérer dans la *runqueue* les *threads* en attente de terminaison d'un *thread* dans le join de manière différente ;
3. Instaurer une notion de priorité aux *threads*, soit de façon statique, soit de façon dynamique en comparant le temps CPU qu'ils ont consommé au total.

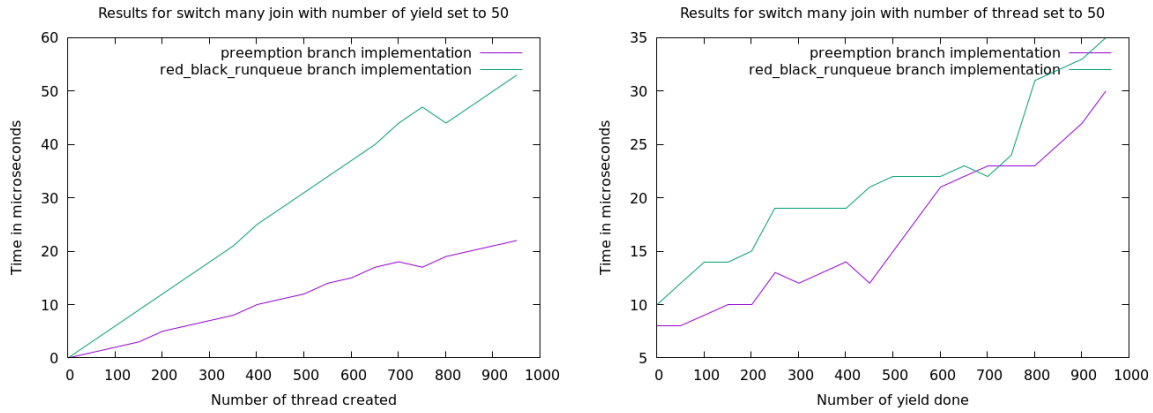
Cette dernière option est intrinsèquement liée aux deux objectifs avancés du sujet que sont les priorités et l'amélioration de la politique d'ordonnancement. La quatrième et dernière version de la bibliothèque, `red_black_tree_runqueue` , est une version qui propose de changer complètement la manière dont les *threads* étaient ordonnancés jusqu'alors. Le nouvel ordonnanceur s'inspire du Completely Fair Scheduler (CFS) qui a fait son apparition avec la version 2.6.23 du noyau Linux.

La mise en place de cette nouvelle politique d'ordonnancement se fait en deux étapes. Dans un premier temps, il convient de calculer pour chaque *thread* le temps CPU consommé. Un simple appel à `gettimeofday(2)` permet de mettre à jour un nouvel attribut du PCB des *threads*, `runtime` , qui est incrémenté du temps consommé à chaque fois qu'un *thread* est mis en pause en faveur du *thread* suivant. Il s'agit alors de trier la liste des *threads* de la *runqueue* selon la valeur de cet attribut, les *threads* ayant le moins de temps CPU à leur actif étant priorisés.

Vient alors le problème de la complexité : une file circulaire implique une recherche linéaire pour déterminer le *thread* dont la `runtime` est la plus faible. Le CFS utilise un arbre binaire de recherche, et plus précisément un arbre rouge-noir qui permet de conserver la propriété d'équilibrage de l'arbre, et par conséquent qui garantit une recherche du *thread* à la `runtime` la plus faible en temps logarithmique. Une implémentation des arbres rouge-noir est disponible sur le même modèle que les queues BSD, ce qui facilite leur intégration dans la bibliothèque.

La TIMESLICE n'est plus définie de manière absolue dans le CFS, mais calculée dynamiquement en fonction du nombre de *threads*. Chaque *thread* se voit attribuer une timeslice d'exécution inversement proportionnelle au nombre total de *threads*, de façon à partager le plus possible le temps CPU entre les *threads*. Chaque *thread* est donc exécuté durant une passe pour un temps égal à $TARGET\_LATENCY * (1/nb\_threads) + MIN\_TIMESLICE$, où TARGET_LATENCY est la durée partagée par les *threads* et est déterminée de façon assez empirique, et MIN_TIMESLICE est une valeur minimale de timeslice (typiquement autour de 5ms) dont a besoin un *thread* pour pouvoir exécuter un minimum de travail utile sans être immédiatement préempté.

Il s'avère que les performances ne se trouvent pas améliorées dans l'état actuel de cette version. Elles ont même plutôt tendance à la régression, et pour cause : la recherche, l'insertion et la mise à jour de l'arbre de recherche sont toutes en $O(\log_2 n)$, là où ces opérations étaient constantes dans les versions précédentes. Ainsi, sur un test qui réalise beaucoup de switch comme le test de la figure 7b, et encore plus sur la figure 7a dans laquelle on augmente le nombre de *threads*, les performances de la version avec arbre de recherche s'éloignent encore plus de celles de la version préemptive, puisqu'elle doit parcourir plus d'éléments au cours de sa recherche en temps logarithmique. La figure 8 montre les différences dans un cas plus "réel". On s'aperçoit alors que l'écart est faible grâce à une optimisation basique de la version avec arbre ce qui se traduit par des courbes quasiment confondues. Le manque de temps ne nous a pas permis d'essayer la mise en place de priorités en conservant une *runqueue* circulaire, mais une telle version aurait inévitablement été affectée également par ce coût de la recherche.



(a) Comparaison sur le test 32 entre la version préemp-(b) Comparaison sur le test 32 entre la version préemptive et la version avec arbre de reacherche (nombre detive et la version avec arbre de recherche (nombre de yields fixé) *threads* fixé)

FIGURE 7 – Effets de l'ajout de l'arbre rouge-noir sur le test 32 *switch-many-join*

Les exemples de tests exécutés dans le cadre de ce projet sont largement "CPU-bound", et les avantages d'un ordonnancement de type CFS ne sont pas immédiats. En réalité, un tel ordonnancement permet à tous les *threads* de disposer de temps équivalents en ressources CPU, là ou un algorithme plus classique de type FCFS ou Round Robin peut potentiellement créer des situations de quasi-famine, et surtout ne prend pas en compte l'attente générée par les blocages inhérents aux gestions des entrées-sorties. L'ajout de priorités s'avère donc coûteux dans un premier temps et ne semble pas bénéficier aux performances, mais offre des avantages à long terme tels qu'une compensation des tâches mettant l'accent sur l'interactivité avec l'utilisateur, ce qui résulte en une meilleure réactivité globale de l'application. De plus, les priorités instaurées ici sont très basiques, et il est possible de les influencer en introduisant un système de priorités statiques.

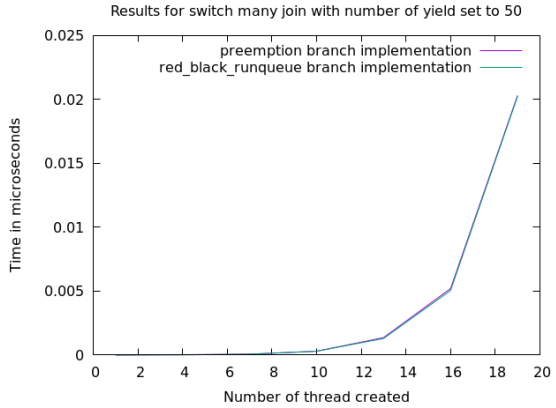
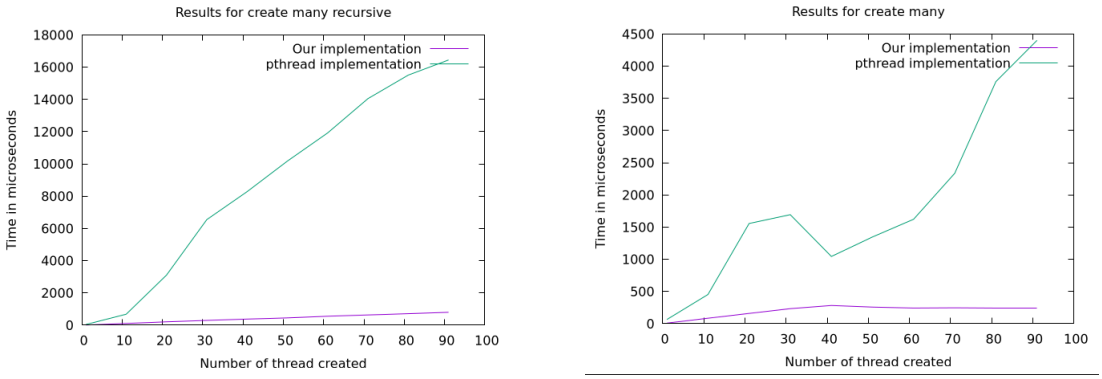


FIGURE 8 – Effets de l'ajout de l'arbre rouge-noir sur le test 51 *fibonnacci*

Observations sur l'exécution des *threads* dès leur création

La manière dont les *threads* sont insérés dans la *runqueue* a (parfois) un impact visible sur les performances de la bibliothèque. En particulier, la plupart des graphes de performance obtenus jusqu'à présent résultent d'une politique d'insertion de *threads* dans laquelle les *threads* venant d'être créés ne sont pas exécutés immédiatement et laissent la main au *thread* qui les a créés, et dans laquelle les *threads* en état de *join* sont insérés en fin de *runqueue* de façon à ce que la terminaison du *thread* qu'ils attendaient ne soit pas immédiatement suivie de leur propre exécution. Si l'on change notamment la première règle et que l'on exécute un *thread* immédiatement après sa création, les performances des tests basés sur un nombre important de création de *threads* peuvent (parfois) être améliorées comme le montre la figure 9 et le test de fibonacci avec une valeur d'entrée de 20 qui s'exécute en un temps de l'ordre de $1e-1$ ms au lieu de $2e-1$ ms sur la machine sur laquelle nous avons fait les tests (version coopérative).

En revanche, ce changement semble avoir un impact en contrepartie sur la performance des tests reposant sur un nombre important d'appels à *join* ou à *yield* (figures 10 et 11).

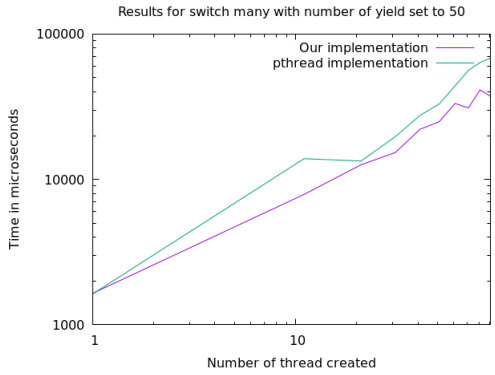


(a) Exécution reportée

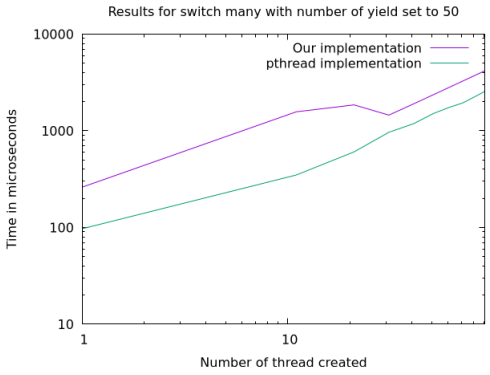
(b) Exécution immédiate

FIGURE 9 – Tests *create_many_XXX* avec et sans exécution immédiate des nouveaux *threads*

Nous expliquerions l'accélération de la création de *threads* par le fait que les *threads* venant d'être créés ont tendance à *yield* plus rapidement que le *thread* qui les a créés (souvent le *main*). Ainsi, le

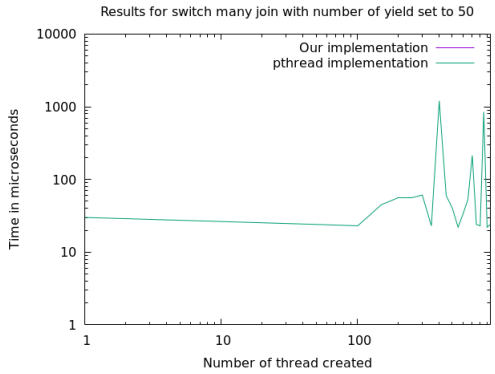


(a) Exécution reportée

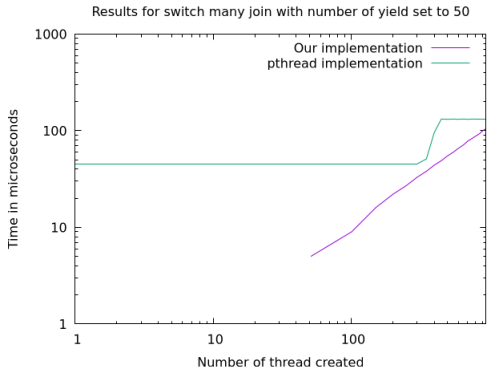


(b) Exécution immédiate

FIGURE 10 – Test 31 *switch_ many (yield set)* avec et sans exécution immédiate des nouveaux *threads*



(a) Exécution reportée (courbe suffisamment faible pour ne pas être visible)



(b) Exécution immédiate

FIGURE 11 – Test 32 *switch_ many_ join (yield set)* avec et sans exécution immédiate des nouveaux *threads*

thread créateur n'occupe pas tout le temps CPU en un coup et le temps d'attente moyen d'un *thread* situé en file d'attente s'en retrouve diminué.

8 Conclusion

Le projet s'achève sur quatre grandes versions successives qui représentent chacune leur intérêt propre. La version `cooperative_scheduling` propose un code fonctionnel et optimisé, mais qui n'inclut que des fonctionnalités basiques et encore loin de toutes les préoccupations de la gestion de tâches sur un système d'exploitation de type Unix. Nous pourrions envisager l'utilisation d'une telle version dans le cadre d'une utilisation coopérative de *threads* qui se veut très basique mais rapide.

La version `preemptive_scheduling` ajoute la préemption, et donc une première approche vers une vraie concurrence des *threads*. Cette concurrence n'a pas été totalement éprouvée dans le sens où certaines simplifications ont été faites : les fonctions de verrou et de relâchement des *mutexes* sont par exemples encadrées d'une désactivation puis d'une réactivation de l'ordonnanceur préemptif pour limiter les incohérences dans l'état des ressources partagées. Cette version apporte donc surtout un intérêt dans le sens où la gestion du temps CPU alloué aux *threads* est beaucoup plus flexible, et ne dépend plus seulement du bon vouloir du code client. Dans des cas de tests suffisamment artificiels, cela peut par exemple présenter un intérêt sur des applications du même type que celle de la recherche de nombre dans un tableau.

La branche `red_black_runqueue` offre l'avantage principal de supprimer de façon certaine la famine pouvant potentiellement être causée par un ordonnancement malheureux à l'égard de certains *threads*. Elle se veut ainsi la plus égalitaire possible et montre surtout qu'un ordonnancement plus élaboré se fait au détriment des performances de la bibliothèque en elle-même. Nous estimons que c'est en grande partie la nature des tâches ordonnancées (CPU-bound, I/O-bound) qui accorde à ce genre de politique de la valeur.

La dernière version non évoquée dans cette conclusion, `preemption_with_stack_overflow` , n'agit pas sur l'ordonnancement à proprement parler, mais offre une fonctionnalité liée à la protection mémoire qui aura eu l'avantage de considérer un défaut d'optimisation introduit par une mauvaise réutilisation des piles de *threads*. Elle offre donc le double avantage de permettre de sécuriser les programmes dont les fils d'exécution utilisent localement un espace important de données ainsi que d'alléger considérablement les allocations coûteuses en temps et en espace mémoire.

Au-delà de l'utilisation que l'on pourrait envisager de la bibliothèque, chaque version aura eu le mérite de mettre en valeur les enjeux les plus importants des systèmes d'exploitation en ce qui concerne l'ordonnancement de processus ou de *threads*, la gestion de la mémoire et le partage des ressources du système, ce qui attribue au projet un intérêt pédagogique fort. Le temps ne nous a malheureusement pas permis d'implémenter un support pour les machines multiprocesseurs. Une vraie parallélisation des tâches aurait mis en valeur la qualité de l'implémentation des *mutexes*. Elle aurait nécessité un verrou fonctionnel sur la *runqueue* qui aurait été partagée par les différents coeurs du système, l'ajout d'un support de *threads* noyau ainsi qu'un équilibrage des charges efficaces pour tirer parti de l'exécution parallèle des *threads*.

Deux autres objectifs avancés n'ont pas été exploités, à savoir la gestion des signaux entre *threads* et l'intégration de sémaphores. Les sémaphores peuvent être généralisés à partir de notre implémentation existante des *mutexes*. La gestion des signaux doit prendre garde à ce qu'il n'y ait pas d'interférences avec les signaux système, tels SIGPROFs, privatisés pour mettre en place la préemption.