Rapport final
Projet de Système d'Exploitation

Maël CHANCOGNE, Raphaël MELINE,
Tarek LAMMOUCHI, Quentin LANNEAU

16 mai 2020

Contents

1 Fonctionnement général

1.1 Mise en place des threads et des files

Une structure *THREAD* a été définie pour contenir tous les éléments nécessaires à l'exécution correcte de chaque thread. Cette structure contient notamment l'identificateur du thread, mais également le contexte lié à ce thread, ainsi que la valeur devant être retournée.

Les threads sont répartis dans deux files distinctes: l'une, appelée *thread_queue*, est la file principale, c'est à partir de celle-ci que l'ordonnanceur choisit le

prochain thread à exécuter, comme nous le verrons un peu plus tard. La seconde, nommée *thread_finished_queue*, est utilisée pour stocker les thread ayant fini leur exécution. Il seront ensuite, à la fin du programme, libérés les uns après les autres par la fonction destructeur, nommée *destr()*, qui s'exécute lors du déchargement de la librairie, donc à la fin du programme. Ce comportement est rendu possible grâce à l'attribut *destructor*, qui est un attribut lié au compilateur *gcc*. On utilise les files simplement chaînées définies dans *sys/queue.h*.

Les deux files, ainsi que tous les éléments nécessaires au bon fonctionnement de la bibliothèque, sont crées et initialisées dans une fonction appelée *constr*, et exécutées lors du chargement de la librairie, encore une fois grâce à un attribut *constructor*.

1.2 Mise en place d'un système d'ordonnancement

La bibliothèque développée au cours de ces dernières semaines fonctionne grâce à un principe d'ordonnanceur, mis en place grâce à une fonction, appelée *schedule_fifo_func()*, qui s'occupe de sélectionner un thread parmi ceux disponibles à chaque fois que cela est nécessaire (que ce soit à la fin du déroulement d'un thread précédent, ou bien lors d'un appel à *thread_yield()*).

En fonction de la situation, le thread ayant abandonné la main est soit remis dans la file des threads actifs, soit, s'il est terminé, rajouté à la file de threads prévue à cet effet, ou encore, en cas de tentative ratée de verrouillage de mutex, uniquement laissé dans la file d'attente du mutex en question.

Le thread à exécuter en suivant est choisi en fonction de l'ordre de la file des threads actifs. Il n'y a en effet pas de système de priorité mis en place. On pourrait cependant imaginer d'autres ordonnancements, comme un ordonnancement exécutant en priorité les threads ayant une "date de péremption" proche.

2 Objectifs avancés

Nous avons également implémenté plusieurs fonctions appartenant à la liste des fonctionnalités avancées souhaitables.

2.1 Implémentation des mutex

Les mutex implémentés sont des structures de données à deux composantes: la première correspond au verrou proprement dit, appelé *has_lock*. La seconde est une liste d'attente de verrou , composée de tous les threads attendant leur tour pour verrouiller le mutex.

Cette implémentation permet une attente passive: en effet, lors du verrouillage, on teste si on peut accéder au mutex, et on change sa valeur si c'est le cas. Sinon, si la place est déjà prise, on rend la main, tout en étant placé dans la file d'attente du mutex, et uniquement là. Cela empêche que la main nous soit rendue sans que le mutex soit de nouveau libre. On peut noter que les signaux

sont suspendus sur la portion critique du verrouillage, pour éviter des problèmes de multiples threads accédant à la même ressource en même temps.

Lorsque le mutex se libère, on permet au premier élément de cette dernière file de tenter sa chance de nouveau. Cependant, sa demande peut de nouveau échouer si un thread étranger à la file d'attente effectue sa demande dans le laps de temps où le mutex est de nouveau libre.

2.2 Mise en place de la préemption

Certains threads peuvent prendre énormément de temps à s'exécuter, bloquant ainsi dans le même temps l'exécution de tous les autres threads. Pour pallier ce problème, on met en place une préemption.

Celle-ci se base sur un timer, réglé sur 5000 microsecondes, qui est une valeur arbitraire permettant une bonne répartition du CPU aux différents threads, sans toutefois passer trop de temps à changer de contexte.

Ce timer, une fois arrivé à échéance, envoie un signal (de la libraire standard) au thread actuel, qui est pris en charge par un gestionnaire de signal, lui faisant passer la main.

2.3 Système de signaux

Les signaux sont des messages envoyés par les threads (et les processus en général) pour communiquer entre eux. Ils nécessitent deux choses: une manière de recevoir un signal, et une manière de les traiter.

Pour mettre en place les signaux, une variable entière est ajoutée à la structure des threads. Celle-ci va servir à stocker jusqu'à 32 signaux (voire 64, selon les architectures), en associant à chaque bit de cet entier un état booléen lié à un signal. Cela permet de n'utiliser que très peu de place en mémoire, mais au prix d'un temps potentiellement plus long pour rechercher un signal en particulier, car il faudra alors effectuer des décalages pour isoler le signal qui nous intéresse (par exemple grâce à un masque).

Cependant, comme les signaux sont tous traités en même temps, ce défaut ne s'applique jamais vraiment, car on peut sauvegarder le masque courant, et le décaler à chaque étape de 1 bit.

Ces signaux ainsi stockés sont traités lors de l'appel à *thread_yield*. Ainsi, il peut s'écouler un certain temps entre l'émission d'un signal et le traitement de celui-ci, mais la mise en place de la préemption réduit potentiellement considérablement ce délai de traitement. Chaque signal a un gestionnaire dédié, stocké dans la structure du thread dans un tableau.

2.4 Optimisations effectuées et points importants

On peut noter que, contrairement à la version de mi-parcours, on n'effectue qu'un unique changement de contexte à la place des deux précédemment nécessaires.

On prend donc deux fois moins de temps pour changer de thread, ce qui, couplé à la préemption, accélère le déroulement des programmes.

De plus, le traitement des signaux lors du changement de contexte, encore une fois couplé à la préemption, permet de ne pas avoir à vérifier constamment l'état des signaux reçus, tout en simulant ce comportement pour l'utilisateur. En effet, les temps de préemptions étant très courts, cette latence induite dans la prise en charge des signaux ne joue que très peu, voire pas.

2.5 Optimisations et implémentations futures

On pourrait facilement implémenter un système de priorité, pour permettre à l'ordonnançeur de choisir avec plus de finesse quel thread effectuer. Cette fonctionalité pourrait soit être implémentée par plusieurs files différentes, une par niveau de priorité, ou par une unique file, l'information étant stockée au niveau de la structure du thread. Le premier choix permet de vérifier très vite qu'aucun thread d'une priorité donnée n'est actuellement présent, en vérifiant que la liste associée est vide, mais cette solution n'est pas forcément adaptée à un nombre très important de niveaux de priorité, car quel que soit le nombre de thread, on vérifiera potentiellement toutes les files.

La seconde solution a l'important inconvénient qu'il faut parcourir la file pour trouver quel thread doit être exécuté, ce qui peut prendre un temps considérable, suivant le nombre de threads présents. Il serait donc probablement meilleur de choisir la première option, car il est inutile d'avoir plus d'une dizaine de niveaux de priorité la plupart du temps.

3 Résultats

3.1 Contexte

Pour pouvoir vérifier la capacité de notre implémentation à gérer efficacement les différents threads, nous avons tracé des graphes du temps de calcul comparé à la bibliothèque pthread donnée par le sujet du projet.
L'ensemble des calculs ont été effectués avec un processeur 4 coeurs qui peuvent monter à 2,2 GHz.
Pour les graphes à montrer, nous avons choisi 3 tests différents en fonctionnement: create-many-recursive, switch-many-join et fibonacci.
Les temps sont tous en secondes, pour les deux premiers tests les graphes sont en fonction du nombre de threads et pour fibonacci c'est en fonction de l'entrée `x` donnée à la fonction `fibonacci(x)`.

3.2 Threads

Pour ces 2 graphes, à chaque nombre de threads différents nous lançons le calcul 10 fois, récupérant chaque temps en une somme que nous divisons par 10 pour en avoir une moyenne. Après cela, pour chaque ordonnée `i`, nous faisons une

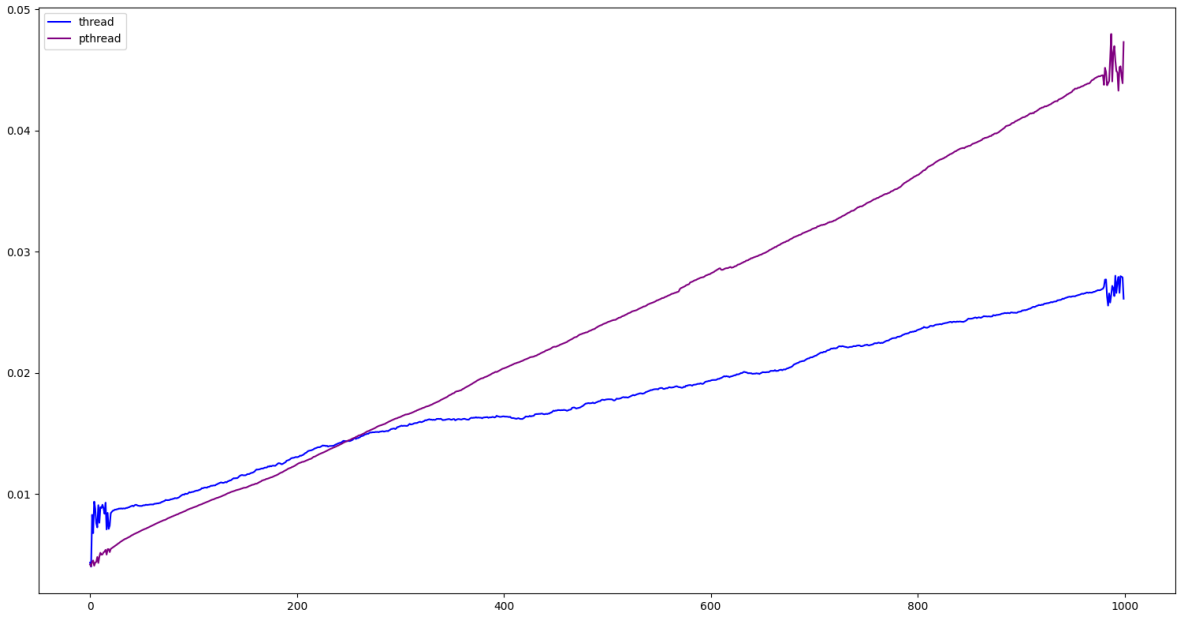

Figure 1: Create-many-recursive

moyenne du temps de calcul de `i-10` à `i+10` threads. Ces 2 procédés permettent d'avoir un bon compromis entre la "véracité" des résultats affichés sur les graphes et la lisibilité des courbes pour éviter des fluctuations trop fortes sur ces dernières.
Ainsi, pour les valeurs de 0 à 10 et de 990 à 1000 nous n'avons pas fait les deuxièmes procédés de moyenne, ce qui explique les fluctuations aux extrémités des courbes.
Nous avons tout d'abord remarqué que notre implémentation est plus efficace que la bibliothèque `pthread` à partir d'environ 300 threads pour le `create-many` (figure 1), et n'est pas plus efficace pour le `join-many` (figure 2). Cela a pu nous rassurer en partie sur la qualité de notre implémentation. Nous avons donc cherché l'origine de cette différence pour `join-many` sans pour autant trouver finalement.
Pour essayer de visualiser l'amélioration de notre implémentation au fur et à mesure de l'avancée de notre projet, nous avons aussi tracé ces graphes avec notre bibliothèque avant l'implémentation de certains points-clés de cette dernière.
Nous avons ainsi tracé des graphes avant la modification de la suppression du

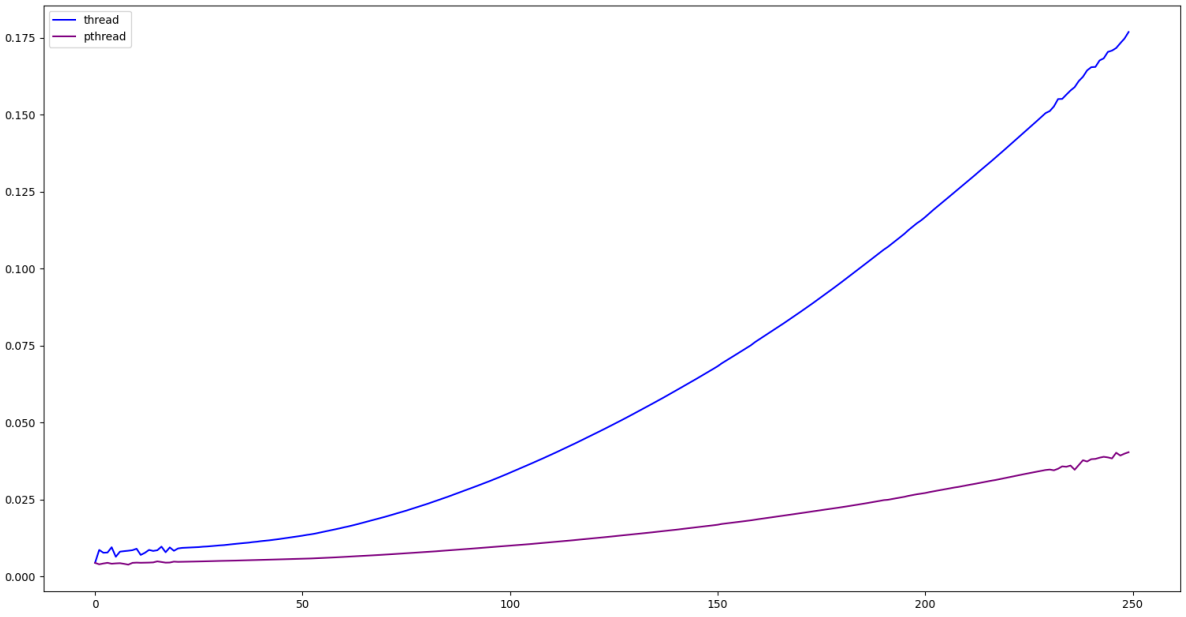

Figure 2: Switch-many-join

double changement de contexte, mais il n'y avait aucun changement majeur sur le temps de calcul pour ces 2 tests. Nous n'avons pas eu le temps de comprendre pourquoi les graphes n'avaient pas montré l'impact de ces améliorations majeures qui auraient du en théorie diminuer de manière sensible le temps de calcul.

3.3 Fibonacci

Pour Fibonacci, nous avons réalisé pour chaque valeur de `x` 10 calculs et fait une moyenne.
Notre bibliothèque devient plus rapide à partir de la valeur de 11 en entrée. De plus, pour la valeur 22, la bibliothèque pthread mettait trop de temps à calculer et l'ordinateur de calcul laissait un certain nombre des calculs par un `core dumped`, ce qui bien sûr réduit en moyenne le temps et c'est cela qui justifie la baisse drastique du temps de calcul pour `pthread` à l'itération 22 sur le graphe.
Néanmoins, un problème récurrent prenait place pour notre bibliothèque sur le calcul de fibonacci. Il arrivait que le calcul pour un thread s'arrête tout simple-

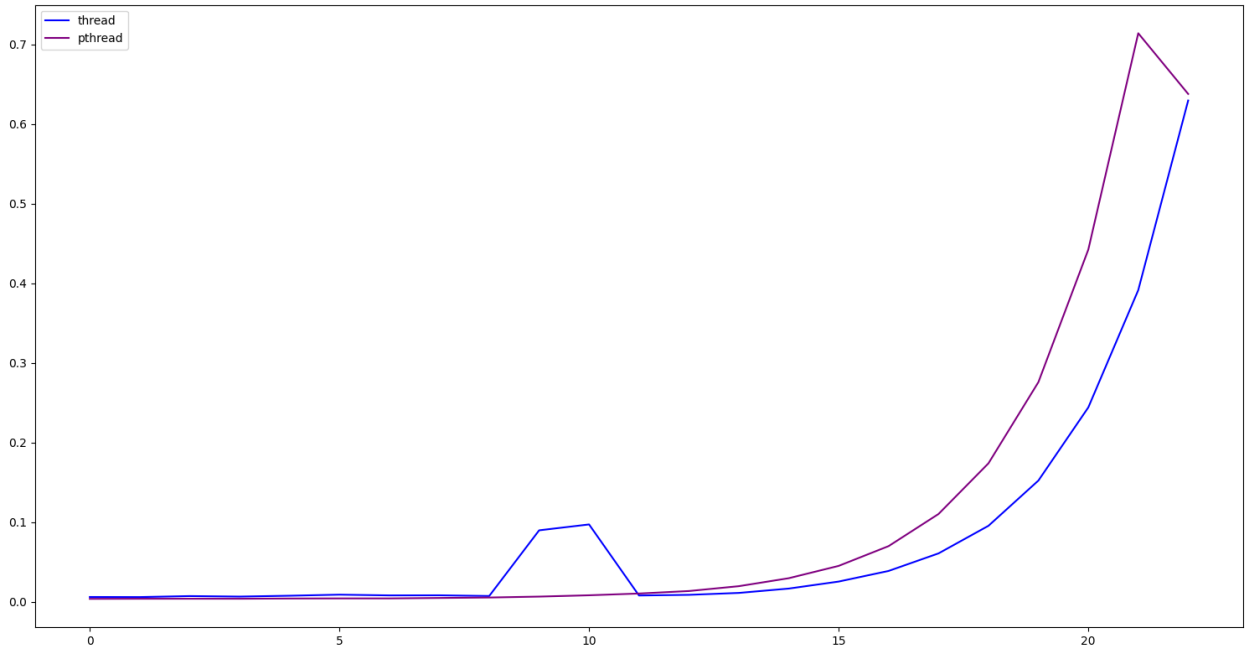

Figure 3: Fibonacci

ment laissant en suspend le processus. Ainsi le processus restait figé et ne lançait donc pas le calcul des prochaines itérations. Dans ces cas-là, nous lancions un signal d'interruption au processus via la commande `Ctrl + c` dans le terminal. Le processus bloquant est donc interrompu et passe la main au prochain calcul. Néanmoins, ce temps en "suspend" est comptabilisé dans notre programme calculant le temps de calcul des threads. Cela justifie la hausse de temps de calcul que nous pouvons voir sur la figure 3 vers l'itération 10.
Nous n'avons pas pu corriger cette erreur de "bloquage" mais sans cela, nous avons pu voir que notre bibliothèque est plus rapide que l'implémentation `pthread`.

4 Améliorations possibles

Évidemment, il convient de dire qu'implémenter l'ensemble des objectifs avancés serait une amélioration de notre bibliothèque. Néanmoins, nous allons nous attarder sur ce qui est améliorable sur le travail réalisé.
Nous n'avons pas ajouté beaucoup de tests de notre implémentation autres que

les tests donnés par le sujet. Pour vérifier si notre bibliothèque fonctionne bien en toute circonstance, il faudrait faire des tests supplémentaires, plus complexes, sur des tris de tableaux par exemple.
En plus d'une recherche approfondie des performances plus basse que pthread pour le `switch-many-join`, il faudrait aussi régler le problème des calculs de fibonacci se bloquant.

List of Figures