Rapport intermédiaire
Projet de Système d'Exploitation

Maël CHANCOGNE, Raphaël MELINE,
Tarek LAMMOUCHI, Quentin LANNEAU

16 avril 2020

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.

On utilise les files simplement chaînées définies dans *sys/queue.h*. En effet, cela permet de mettre en place rapidement des files contenant n'importe quel type d'élément, et nous évite d'avoir à réimplémenter une structure de file, pouvant être sujette à des bugs, ou à un manque d'optimisation.

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*.

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 exécutée 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 *sched-*

ule_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()*). Cette fonction d'ordonnancement est enregistrée dans un contexte, sur lequel on "swappe" lorsque c'est nécessaire. Pour changer de thread, on effectue donc deux changement de contextes.

2 Problèmes et bugs rencontrés

Comme on peut le voir sur la forge, tous les tests, exceptés ceux portant sur les mutex (encore non implémentés) passent. Cela nous montre que, dans l'ensemble, la bibliothèque répond plutôt bien aux sollicitations qui peuvent lui être faites.

Cependant, certains points peuvent être soulevés. En effet, on peut tout d'abord parler du double changement de contexte utilisé pour changer de thread. Il serait peut-être plus efficace de faire en sorte qu'il n'y ait qu'un seul changement de contexte à ce moment, les changements de contexte étant potentiellement lourds à effectuer, et prenant donc du temps.

3 Améliorations prévues

Dans un premier temps il convient de faire les graphes de temps d'exécution de notre bibliothèque. Ainsi, nous pourrons comparer le temps d'exécution de notre bibliothèque et celle des pthreads. Nous pourrons ainsi chercher les erreurs de performances dans notre code et l'améliorer. Cela nous permettra aussi de situer noter projet vis-à-vis des attentes.
Après avoir fait toute cette partie qui nous demandera un travail non négligeable, nous voulons résoudre le problème du double changement de contexte, qui sera par ailleurs sûrement résolu en cherchant à optimiser le temps d'exécution.
Ensuite, nous attaquerons les objectifs avancés du projet, mais nous n'avons pas encore décidés lesquels traiter en premier car il nous reste encore à traiter le gros sujet des graphes de performance de nos méthodes de threads.