Projet Système - Threads en espace utilisateur

Déroulement

Consignes

Projet à réaliser en équipe de 4 ou 5 étudiants. Les équipes ont été tirées au sort et sont disponibles sur le serveur Thor/ruby.

Le langage de programmation devra être le C. Vous devez utiliser le repository GIT sur le serveur Thor/ruby.

Rapport et démonstration intermédiaires

Une démonstration des fonctionnalités implémentées devra être présentée pendant la 6ème séance la semaine du **20 au 24 avril**. L'encadrant passera une dizaine de minutes avec chaque équipe pour faire un point détaillé sur ce qui marche ou ne marche pas, notamment en faisant tourner les différents programmes de tests.

Un rapport intermédiaire (environ 4 pages) sera rendu la semaine précédente (**jeudi 16 avril**) en PDF sur le serveur Thor. Le rapport décrira ce qui marche ou ne marche pas, pourquoi, et ce que vous avez prévu de faire pour la suite du projet. **Inutile de rappeler le sujet ou d'expliquer l'interface `thread.h`!**

Rapport et soutenance de fin de projet

La soutenance finale aura lieu le mardi 19 ou mercredi 20 mai.

Elle durera environ 13 minutes suivies d'environ 5 mn de questions, et consistera en une présentation sur vidéoprojecteur et une démonstration. **Vérifiez avant de venir que vous savez utiliser un vidéoprojecteur avec votre ordinateur, et allumez votre ordinateur avant d'entrer dans la salle** (et amenez un adaptateur VGA si nécessaire).

Un rapport d'environ 8 pages devra être rendu le week-end précédent (*dimanche 17 mai*), au format PDF. Le rapport décrira ce qui a été implémenté, comment et pourquoi. Il sera accompagné d'une archive tar.gz contenant tout le code source et un minimum de documentation permettant de compiler et tester le projet. **Les deux fichiers doivent être uploadés sur le serveur Thor.**

Les rapports et soutenances devront notamment expliquer comment vos tests montrent la validité du comportement de votre bibliothèque et indiquer les différents coûts que vous avez mesurés (voir la partie premiers objectifs ci-dessous). **Inutile d'écrire des pages pour rappeler le sujet, il faudra se concentrer sur les choses utiles et prêtant à discussion.** Montrez la complexité de votre code en traçant graphiquement ses performances pour plusieurs tests.

Evaluation

A partir des rapports (intermédiaire et final), de la démo à mi-parcours et de la soutenance finale, on jugera :

- Est-ce que le code compile ? (sans warning)
- Est-ce qu'il marche et quels programmes de test nous le prouvent?
- Quelle est la complexité des différentes fonctions, est-ce que le code marche vite, et quels programmes de test nous le prouvent?
- Quelles fonctionnalités sont supportées?
- Comment vous expliquez le fonctionnement de tout ceci, ses inconvénients, ce qui pourrait être amélioré?

Contenu du projet

Ce projet vise à construire une bibliothèque de threads en espace utilisateur. On va donc fournir une interface de programmation plus ou moins proche des pthreads, mais en les exécutant sur un seul thread noyau. Les intérêts sont :

- Les coûts d'ordonnancement sont beaucoup plus faibles
- Les politiques d'ordonnancement sont configurables

- On peut enfin expérimenter le changement de contexte pour de vrai

Mise en route

Pour commencer, on va construire un petit programme qui manipule différents threads sous la forme de différents contextes. On commencera par exécuter ce programme (ne pas compiler avec **`-std=c89`** ou **`-std=c99`**). Comment fonctionne-t-il et que se passe-t-il ?

Etendre le programme pour manipuler plusieurs contextes à la fois et passer de l'un à l'autre sans forcément revenir dans le main à chaque fois. En clair, montrer qu'on peut exécuter plusieurs tâches complexes et indépendantes en les découpant en morceaux et en entrelaçant l'exécution réelle de ses morceaux.

Objectifs pour les 2-3 premières séances

L'objectif du projet est tout d'abord de construire une bibliothèque de gestion de threads proposant un ordonnancement coopératif (sans préemption) à politique FIFO. Cela nécessitera une bibliothèque de gestion de liste (voir les ressources en bas de cette page au lieu de réinventer une roue bugguée). On devra donc tout d'abord définir une interface de threads permettant de créer, détruire, passer la main (éventuellement à un thread particulier), attendre la/une terminaison, ...

Concrètement, il faudra :

- **Implémenter cette interface de gestion de threads**. On pourra éventuellement s'en écarter si nécessaire, mais rester relativement proche de **`pthread.h`** afin de pouvoir facilement comparer les deux implémentations avec des programmes de test similaires.

- **Exécuter correctement ce programme d'exemple**. Sa sortie devra être similaire lorsqu'on le compile avec **`-DUSE_PTHREAD`** pour utiliser les pthreads à la place de votre bibliothèque (elle pourra être légèrement différente, pourquoi?).

- **Associer un thread à la fonction *main* du programme** : Être capable de le manipuler comme n'importe quel autre thread, sinon vous aurez rapidement des problèmes (pour que **`thread_self`** marche, pour qu'il puisse reprendre la main plus tard pendant l'exécution, ou s'il doit faire un **`join`** sur ses fils, ou le contraire).

- **Créer un Makefile avec les règles suivantes à la racine du repository :**
  - **`make`** (cible par défaut) : Compiler votre bibliothèque et les tests.
  - **`make check`** : Exécuter les tests, avec des valeurs raisonnables pour les tests qui veulent des arguments en ligne de commande.
  - **`make valgrind`** : Exécuter les tests sous valgrind.
  - **`make pthreads`** : Compiler les tests pour les pthreads.
  - **`make graphs`** : Tracer des courbes comparant les performances de votre implémentation et de pthreads en faisant varier les arguments passés aux tests.
  - **`make install`** : Installe les fichiers compilés dans le répertoire install avec l'architecture suivante:
    - install
      - lib
        - libthread.so (ou libthread.a)
      - bin
        - 01-main
        - 02-switch
        - 11-join
        - ...
        - 01-main-pthread
        - 02-switch-pthread
        - 11-join-pthread
        - ...

    La règle d'installation sera nécéssaire pour trouver les binaires dans le répertoire `install` à la racine pour que le serveur Thor puisse par exemple lancer `./install/bin/22-create-many-recursive`.

Tests de robustesse et performance

- **Faire tourner tous les programmes tests disponibles ici** (sauf celui sur les mutex).
  - Ils doivent retourner correctement (équivalent du **`make check`** dans de nombreux projets).
  - **valgrind** devra confirmer qu'il n'y a aucune fuite mémoire.
  - Quand le programme accepte un nombre en argument, regarder jusqu'à quelle valeur il fonctionne, tracer la courbe de temps d'exécution selon cette valeur, et comparer aux performances des pthreads (quand on compile avec **`-DUSE_PTHREAD`**).
  - L'en-tête de chaque programme précise toutes les choses que vous devez vérifier.

  Ajouter d'autres programmes de test.

- En plus de **`fibonacci.c`**, **tester d'autres applications parallèles créant beaucoup de threads** :
  - Calcul de la somme de tous les éléments d'un grand tableau par diviser-pour-régner.
  - Tri de très grand tableau (rapide, fusion, ...).
  - D'autres!

  On ne cherchera pas à optimiser le programme lui-même, on conservera un modèle simple créant beaucoup de threads simultanément afin de tester l'ordonnanceur. Cela implique notamment de faire tous les **`create`** puis tous les **`join`** plutôt qu'un **`join`** directement après chaque **`create`**. Dans le rapport, on pourra présenter une courbe de temps d'exécution en fonction du paramètre d'entrée.

On veillera de plus à ce que les tests de performance soient suffisamment longs pour être significatifs : inutile de mesurer la durée d'exécution d'un programme si son initialisation est dix fois plus longue que ce qu'on cherche à comparer, ou si son exécution prend un milliseconde.
Lors de la présentation de ces résultats dans le rapport, on précisera bien la machine utilisée (combien de processeurs?) afin que la comparaison avec pthreads soit significative. Si nécessaire, on pourra *binder* les programmes pour controller finement le nombre de processeurs physiques réellement utilisés.

Veiller à conserver une **complexité satisfaisante** du code afin d'assurer de bonnes performances pour les différentes opérations. Ces éléments seront mis en valeur dans les tests de performance. Cela implique notamment de :

- Ne pas parcourir plusieurs fois la même longue liste (ou long tableau) dans une même opération.
- Ne pas parcourir de longue liste ou long tableau inutilement : par exemple il est inutile de parcourir une liste contenant tous les threads (prêts, bloqués voire morts) quand on cherche uniquement un thread prêt.

Objectifs avancés

Une fois ce travail de base réalisé, chaque groupe devra s'intéresser à certaines des idées suivantes.

- **Support des machines multiprocesseur** (difficulté 4/4) :
  Utiliser plusieurs threads noyau pour exécuter vos threads utilisateur en même temps (quitte à utiliser des pthreads en interne dans votre bibliothèque). Cela nécessitera notamment l'ajout de fonctions de verrouillage et synchronisation.
  On observera également à l'impact de ce support sur les performances de l'ordonnanceur. Va-t-on vraiment deux fois plus vite avec deux processeurs ? Pour quel type d'applications ? On pourra également ajouter des fonctions permettant de verrouiller un thread sur certain(s) coeur(s).

- **Préemption** (difficulté 3/4) :
  On regardera comment utiliser les alarmes/timers pour provoquer la préemption, c'est-à-dire prendre de force la main au thread en cours d'exécution.
  On mesurera l'impact sur les performances du code, et on rajouter des tests pour montrer l'intérêt de la préemption.

- **Fonctions de synchronisation** (difficulté 2/4) :
  On ajoutera des mutex, sémaphores, voire variables de condition pour permettre aux threads de manipuler des données partagées de manière sécurisée. On veillera alors à faire fonctionner les tests 61 et 62 et à rajouter d'autres tests pour les éventuelles autres fonctionnalités (tests unitaires ou de performance).
  Ces techniques de synchronisation sont beaucoup plus intéressantes si la préemption est déjà implémentée (pourquoi ?).
  Les sémaphores pourront consister en une généralisation du **`join`**. On pourra utiliser les **`pthread_spinlock_t`**.
  On réfléchira à la validité de passer la main lorsqu'on tient un verrou et l'impact que cela peut avoir sur l'implémentation (attente active ou passive?).

- **Détecter les débordements de pile** (difficulté 2/4) :
  En utilisant par exemple **`mprotect`**, on détectera quand un thread déborde de sa pile et on le supprimera e thread fautif sans gêner les autres. On pourra utiliser **`sigaltstack`** pour donner une pile au traitant de segfault quand la pile du thread est déjà pleine.
  On pourra réfléchir à modifier le **`join`** pour signaler l'erreur proprement.

- **Signaux** (difficulté 2/4) :
  Par exemple permettre d'envoyer un signal entre deux threads de votre processus (indépendamment des signaux système, notamment s'ils sont utilisés pour la préemption), voire supporter la fonction **`sigwait`** pour dormir jusqu'à la réception d'un signal.

- **Priorités** (difficulté 2/4):
  Ajouter une priorité aux threads, soit lors de leur création, soit modifiée plus tard. Ce point est beaucoup plus intéressant si vous avez déjà implémenté la préemption car on pourra jouer sur les timeslices. Sinon il faudra faire **attention à la complexité des fonctions d'ordonnancement et aux famines**.

- **Amélioration l'ordonnancement des threads** (difficulté 1/4) :
  Proposer différentes politiques d'ordonnancement (FIFO, priorités, ...) avec un choix à la compilation (voire à l'exécution).

- Mettre en place un **système de compilation construisant une bibliothèque partagée** (difficulté 1/4) :
  On utilisera par exemple les autotools ou cmake. Cette bibliothèque et le fichier d'entête de l'interface devront pouvoir être installés pour que des programmes externes puissent les utiliser facilement.

Ressources

Listes

Pour éviter de réimplémenter vous même des listes et de passer des heures à les débugguer, regarder les Queue BSD (un peu obscur au premier abord mais très efficace).
Si vraiment vous ne voulez pas les utiliser, ou s'il y a un problème de complexité, on pourra éventuellement regarder aussi les CCAN list.h (similaire aux listes du noyau Linux) ou éventuellement les GList (mais attention à la gestion des fuites mémoire).

Valgrind

Valgrind va vous être indispensable pour trouver les fuites ou corruption mémoire, mais il va falloir l'aider un peu en lui disant où se trouvent les piles de vos threads. Pour ce faire :

```
#include <valgrind/valgrind.h>
...

...
/* juste après l'allocation de la pile */
int valgrind_stackid = VALGRIND_STACK_REGISTER(context.uc_stack.ss_sp,
                                               context.uc_stack.ss_sp + context.uc_stack.ss_size);
/* stocker valgrind_stackid dans votre structure thread */
...

...
/* juste avant de libérer la pile */
VALGRIND_STACK_DEREGISTER(valgrind_stackid);
...
```

Pour aller plus loin, setjmp/longjmp

`setjmp`/**`longjmp`** sont une variante un peu plus hardcore de l'interface **`makecontext`**/**`swapcontext`**. Elle est souvent utilisée dans les implémentations "sérieuses", mais le principe reste le même.

- GNU C library manual: System V contexts
- Combining setjmp()/longjmp() and Signal Handling.
- Implementing a Thread Library on Linux

- La page du cours