Projet Système

DÉROULEMENT

Consignes

Projet à réaliser en groupe de 5 étudiants (5 groupes par groupe de TD), encadré par Nathalie Furmento, Mathieu Faverge, Emmanuel Jeannot et François Tessier. **Les groupes devront être formés et indiqués par mail à votre encadrant et à Brice Goglin avant la fin de la première semaine (vendredi 5 avril 2013).**

Le langage de programmation devra être le C. L'utilisation d'un système de gestion de version (SVN, GIT, ...) pour maintenir le projet est vivement recommandée.

Rapport et démonstration intermédiaires

Une démonstration des fonctionnalités implémentées devra être présentée lors de la 5ème séance (**date à confirmer**). Un rapport intermédiaire (4-5 pages) sera de plus rendu 3 jours avant.

Rapport et soutenance de fin de projet

La soutenance finale aura lieu la semaine précédent les examens (date à confirmer). Elle durera environ 13 minutes suivies d'environ 5mn de questions, et consistera en une présentation sur vidéoprojecteur et une démonstration. Les ordinateurs de la salle pourront être utilisés pour la démonstration, mais il sera préférable de présenter une démo vidéoprojetée depuis un portable.

Un rapport d'environ 8 pages devra être rendu la semaine précédente (date à confirmer), au format PDF par mail à votre encadrant et à Brice Goglin. 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. Une version papier du rapport devra de plus être disponible le jour de la soutenance.

Les rapports et soutenance 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 (complexité, modification de l'interface de programmation, ...).

Soutenances

Planning donné plus tard.

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 (avec une liste de threads). 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 une interface similaire à celle-ci**. 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 à celle de la version pthread du même programme, mais pas forcément identique (pourquoi?).

- **Régler le cas du thread principal** (le *main* du programme): Etre 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 pendant l'exécution, ou s'il doit faire un **`join`** sur ses fils).

Tests de robustesse et performance

- **Faire tourner tous les programmes tests disponibles ici**.
  - 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.
  - 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** : Utiliser plusieurs threads noyau pour exécuter vos threads utilisateur en même temps. 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** : On pourra commencer par une préemption pseudo-coopérative ou la préemption est cachée dans toutes les fonctions de la bibliothèque (par exemple dans **`thread_self()`**), avant de s'intéresser à une vraie préemption utilisant par exemple des signaux.

- Ajouter des **fonctions de synchronisation de type sémaphores et/ou mutex** pour permettre aux threads de manipuler des données partagées de manière sécurisée. Ce point est beaucoup plus intéressant si vous avez déjà implémenté la préemption.
  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** des threads (par exemple en utilisant **`mprotect`**) puis être capable de tuer le thread fautif sans gêner les autres. On pourra utiliser **`sigaltstask`** pour donner une pile au traitant de signal quand la pile du thread est déjà pleine.

- **Annulation d'un autre thread** (**`thread_cancel()`**). Si votre code est multiprocesseur, réfléchir à la sémantique en regardant ce que propose pthread.

- **Signaux** : Par exemple permettre d'envoyer un signal à un thread particulier.

- **Priorités** : Ajoutez une priorité aux threads. 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 aux famines.

- **Amélioration l'ordonnancement des threads** : 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** basé sur des Makefile (en utilisant éventuellement 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

Notez que **`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.

- Pour éviter de réimplémenter vous même des listes, regarder les GList, les Queue BSD, ou CCAN list.h.
- GNU C library manual: System V contexts
- Combining setjmp()/longjmp() and Signal Handling.
- Implementing a Thread Library on Linux
- Un TP sur le même sujet
- La page du cours

Valgrind est très utile 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);
...
```

Updated on 2013/03/20.