Rapport du TD2 de Nachos
Système d'exploitation
11 Novembre 2019

ANEAS Alan et CERUTTI Marc
Master - Année 1
Spécialité informatique

Professeur de TD : GUILBAUD Adrien

Sommaire

1 Introduction

Le but de ce travail est d'implémenter des fonctionnalités d'un système d'exploitation, notamment créer un système d'entrée-sortie, sous Nachos.

Ce travail est la suite du TD1 qui nous a permis d'analyser la logique d'un système d'exploitation, en y intégrant et modifiant des tests, des appels système, et des composants d'entrée-sortie comme la console. Maintenant nous nous occupons de l'implémentation des threads utilisateur.

Lien du cours :
http://dept-info.labri.fr/ guermouc/SE/

2 Bilan

2.1 Stratégie d'implémentation et choix faits

Nous avons tout d'abord implémenté la synchronisation de la console et du buffer système. Pour la console, nous avons ajouté un sémaphore pour faire un verrou sur ses fonctions. Étant donné que nous avons un buffer global, il a fallu inclure synch.h afin d'avoir un sémaphore pour la synchronisation du buffer, et ajouter le verrou autour des appels systèmes l'utilisant.

Nous avons ensuite implémenté les appels système pour créer et libérer les threads utilisateurs, avec les fonctions intermédiaires du noyau dans un nouveau fichier userthreads. Nous avons choisi pour transmettre au thread utilisateur la fonction à utiliser et les paramètres de celle-ci dans une structure qui est allouée lors de la création du thread sur le tas, et libérer une fois que le thread est réveillé pour la première fois. On profite du réveil pour *set* l'espace mémoire et les registres, ainsi qu'allouer la pile du thread.

Pour l'allocation de la pile pour plusieurs threads, nous avons fait le choix d'implémenter un bitmap qui segmente la taille de la pile principale en plusieurs piles de threads. Par défaut le thread "main" est sur le premier segment mémoire. Pour ensuite créer plus de threads que ne le peut la pile, nous avons choisi d'utiliser un sémaphore pour retarder la création du thread s'il n'y avait plus de place disponible dans la partition.

2.2 Points forts et points faibles

Grâce à nos sémaphores utilisés pour la synchronisation du buffer et de la console, on a une certaine logique où un appel système utilisant la console ne peut pas se mélanger avec d'autres. Bien sûr ce n'est pas forcément plus efficace du point de vue des performance de temps, où l'on préfère préserver nos performances de mémoire.

Pour les threads, nos points forts sont que l'on peut relativement ne plus se soucier du nombre de threads maximal en "coexistence" que l'on peut créer, à défaut on retarde leur création jusqu'à qu'il y ait de la mémoire de libre pour la pile. Malheureusement nous n'avons pas su comment protéger les différentes piles de threads. Nos threads peuvent cependant "survivre" sans leurs parents grâce à ThreadExit qui garde une trace pour un processus de son nombre de threads.

2.3 Points non traités

Nous n'avons pas fait les exercices bonus.

Nous n'avons pas pu voir comment protéger les piles pour les threads et éviter que depuis un autre on puisse y accéder.

3 Points délicats

Le plus difficile a été de déboguer la création de nos threads via des tests et l'implémentation de la mémoire partagée. De plus, les fichiers addrspace et threads ne proposaient pas une différence propre entre un processus et des threads de base, et était trop complexe à comprendre en sa totalité. La sécurisation de la mémoire de la pile était donc trop de travail pour le temps imparti.

4 Limitations

Les limitations de notre implémentation sont :

- On n'est pas assuré que la mémoire de nos threads ne puisse être corrompue.
- On doit changer le nombre de threads coexistant en recompilant.
- On doit faire attention que la taille de pile du processus (l'espace mémoire partagé des threads) soit divisible par le nombre maximal de threads.

5 Détails sur les programmes de test

- *makethreads* teste simplement la création et la libération des threads.
  Il affiche juste des caractères pour le thread *main*, le père, et d'autres pour le thread créé, le fils.
- *threadsconsole* vérifie la bonne synchronisation de la console.
  Techniquement nous avons fait ces deux premiers tests en même temps pour pouvoir mieux déboguer les programmes.
- *threadexit* permet de tester la survie d'un thread fils à son père.
  En d'autres termes cela permet de tester que tous les threads d'un même processus peuvent se finir tranquillement jusqu'à ce qu'ils appellent eux-même ThreadExit, ou que l'un d'eux fasse *Exit* et "tue" les autres.
- *multithreads* permet de tester la création de plusieurs threads en même temps.
  On peut changer le nombre de threads pour voir les performances avec un grand nombre.
- *threadsacessotherstack* teste l'accès artificiel d'une variable d'un autre thread.
  Cela permet de vérifier si un thread peut en corrompre un autre en accédant à sa mémoire de pile, utilise un pointeur global et des boucles possiblement infinies, mais qui s'arrête, soit, car le programme a pu y accéder illégalement, soit, car le système a fait une interruption pour protéger la mémoire de l'autre thread.
- *stressthreadstack* teste d'une autre manière la corruption mémoire.
  En utilisant une fonction récursive infinie, on déborde de la pile et empiète sur la pile des autres threads. Avec de la chance, on a une erreur, sinon le comportement peut être totalement différent.