Rapport du TD3 de Nachos
Système d'exploitation
27 Décembre 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 TD 1 et 2 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, ainsi que de créer des threads utilisateurs et gérer la mémoire de la pile. Maintenant nous nous occupons de l'implémentation de la création de nouveux processus utilisateurs via la mécanique de l'appel système ForkExec.

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 l'utilisation d'adresses virtuelles pour la futur création de nouveaux processus.

Pour cela, nous avons créé la fonction ReadAtVirtual qui s'occupe de la lecture dans l'éxécutable des adresses translatés avec le même principe que la fonction ReadAt de départ, écrire dans la mémoire physique correspondante la section à l'adresse virtuelle donnée.

Mais du coup afin de gérer les adresses physiques de la mémoire, on a crée l'objet pageProvider qui nous permet de décerner les pages et de les libérer quand un processus se finit. Nous avons opté pour une allocation aléatoire pour s'assurer de son bon fonctionnement.

Nous avons ensuite implémenté la création et la destruction de processus via l'appel système ForkExec.Pour la terminaison des processus on a du par contre faire le choix de terminer tout les processus via un threadExit, sauf pour le dernier sur la machine, même si on fait appel à exit.

2.2 Points forts et points faibles

La stratégie d'attribution aléatoire permet de plus se rapprocher d'un point de vue sécurité des systèmes modernes.

Notre implémentation des threads et des processus suites aux tds précédents permet de gérer un grand nombre de threads sans se poser la question de la mémoire. Nous avons cependant pas fait la même chose pour ici les processus, sous peine de revenir à un lancement linéaire, ce qui est une faiblesse pour les threads lors de notre implémentations sur le Td précédent lorsqu'ils étaient en grand nombre. En contre partie, nous avons une erreur lorsqu'il y a plus assez de mémoire pour faire de nouveau processus. Ce système peut donc être faccilement remplacé par un swap pour de futurres implémentations.

Nous avons tester la création de threads multiples dans la création de nouveaux processus, et le mécanisme est visible à l'oeil nu. La création prend apparemment beaucoup de temps avec notre implémentation, ce qui est cependant prévisible. Aprés, le fait que se soit un thread ou un processus qui se termine n'a plus grand chose en différrence, ce qui fait que certains appels systèmes paraissent inutiles dans notre implémentation.

2.3 Points non traités

Nous n'avons pas fait les exercices bonus. La terminaison du coup des processus n'est donc pas différentes des threads, et on laisse gentiment chaque processus considéré grossiérement comme un thread se terminer.

3 Points délicats

Le plus difficile a été de déboguer les erreurs concernant la mémoire. Comme les messages d'erreurs ne sont pas forcément explicite car elles ne se déclenche que lors de l'accés, la résolution de celles-ci prend du temps.

4 Limitations

Les limitations de notre implémentation sont :

- Une fois que la limite des pages est atteinte, le sytème s'arête.
- Pour changer le nombre de processus coexistant on doit recompiler pour avoir plus de mémoire physique.
- On ne peut pas arréter les processus enfant d'un autre en tuant le parent.
- Un gand nombre de créations de processus et de threads ralentit tangiblement le système.

5 Détails sur les programmes de test

- *userpages0* et *userpages1*,, sont des programmes utilisateurs afin de tester les forks.
- *fork*, permey de dérouler l'appel système à ForkExec.
- *forkstress*, permet de génerer plusieurs fork qui eux même vont appeler des threads. En cas de mémoire insuffisant, un assertion failed s'affiche et arrête la simulation.
- *threadstress_forkfunc*, permet de tester la simulation en situation de stress pour les threads. Il est appelé aussi dans forkstress pour permmetre de voir la coexistence des threads et des forks.