Projet: Théorie des Systèmes d'Exploitation

Axel Viala <axel.viala@darnuria.eu>

17 décembre 2020

Questions - rendu : Les questions de code et de compréhension sont à faire, celles notées *bonus* sont optionnelles mais recommandées ! Pour le rendu un dépôt git sans historique réécrit ni triche sera bien, les questions doivent être rendues au format *markdown* dans le fichier `readme.md` à la racine du dépôt.
Taille des groupes : max 4 par groupe 2 min pensez à noter les noms de votre binôme, votre dépôt contiendra tous les travaux pratiques. l'usage de git sera pris en considération en bonus. ;)

Table des matières

1 Introduction

Dans ce projet nous allons mettre en pratique les principes vu en cours sur les systèmes exploitations. Nous allons réaliser un shell basique ou (beaucoup plus dur) un debugger basique.

le sujet mini-debugger est plus dur (et pas encore fini) attention donc !

1.1 Processus

Au cours de ce travail pratique nous allons commencer par démystifier par la pratique la gestion des processus et entrées sorties en réalisant un mini invité de commande très simple (un shell).

Que vous soyez sur Linux, MacOS ou Windows vos programmes sont isolés les un des autres, cette abstraction s'appelle le « processus », parfois appelée «tâche». Ce mécanisme de base permet d'implémenter des isolations plus fortes comme celles utilisées dans docker ou les machines virtuelles type QEMU avec le concours du kernel et fondation du système en espace utilisateur.

Mais dans la vie de tout les jours les processus permettent d'écrire des programmes sans avoir accès aux autres programmes sur l'ordinateur.

Attention : Dans le semestre nous verrons un concept proche, le concept de *thread* ou fil de calcul. Un thread est un fil de calcul en plus dans un processus ! Pour avoir un autre programme il faut faire un processus, si vous voulez juste faire des calculs sur un autre processeur, un thread est ce qu'il vous faut !

Un processus possède son propre espace d'adressage découpé en segments, c'est à dire sa propre pile **stack**, son propre tas **heap**, son propre code dans le segment **.text**, ses propres données connues à la compilation **.data** et d'autres avec leur propre usage comme l'espace des bibliothèques dynamiques. (en tout cas sous linux x86_64)

La taille de cet espace dépend de votre système, sur un système 32bit cet espace va de l'adresse `0x0000_0000` à `0xFFFF_FFFF` sur 64 bits de `0x0000_0000_0000_0000` à `0xFFFF_FFFF_FFFF_FFFF`[1] la formule pour connaître l'espace maximum d'adressage est la suivante : $2^n - 1$.

Cet espace est découpé comme nous l'avons vu avant en segments, c'est votre système d'exploitation qui orchestre cette abstraction, les compilateurs lieurs **linkers** et assembleurs la respectent.

2 Pratique - micro-shell

Pour s'approprier les concepts, on va écrire un programme qui reproduit le comportement d'un shell. Un shell est le programme qui attend des commandes et organise l'exécution de programmes avec un langage dédié dans votre gestionnaire de terminal favori. Il en existe de nombreux, par exemple : Bash, Zsh, Ksh, PowerShell (Windows).

Votre programme à l'issue du projet fera au moins les actions suivantes :

— Vous affichez un caractère pour inviter l'utilisateur à écrire
— Vous attendez une saisie sur la `STDIN`
— Tentez d'exécuter la commande
— Relevez le statut de la commande
— Pourra gérer un pipeline de commande avec `|`
— Recommencez à l'étape initiale

3 Execution d'un Processus

Un shell dois pouvoir exécuter une commande tapée par un utilisateur, dans les fait cela va nécessiter de : créer un processus et lui faire exécuter un programme de notre choix, par exemple `ls`.

Par exemple si vous écrivez `sl` au lieu de `ls`[2].

Sous Linux et MacOS il y a `fork()` et `execve()` pour créer et exécuter un processus.

Un processus peut exécuter un processus, ça donne un parent et un enfant souvent exécuter et avoir un enfant sont deux actions séparées pour un OS et exécuter un nouveau programme reviens à créer un enfant puis exécuter. Votre programme enfant va avoir son propre espace d'adressage, mais il va hériter de vos descripteurs de fichier ouvert et vous aurez la responsabilité de vérifier qu'il ai bien terminé[3]

Par exemple sous Linux le seul programme sans parents c'est le processus 1 souvent appelé `init`. Sa responsabilité est assez importante, tous les autres processus sont des enfants ou petits enfants de celui-là.

3.1 Lire tout le sujet - reflechir à comment faire

Étape essentielle sur papier ou avec schéma, ou vous conceptualisez comment réaliser ce projet.

3.2 Lire une entrée tapée au clavier

Pouvoir afficher un invité de commande, c'est à dire un caractère ou une sequence qui invite a saisir une commande souvent c'est `'>'` ou plus.

Indices : `printf()`, `write()`, `read()`.

Approfondir : gérer les erreurs avec `perror()` et `assert()`

1. En décimal : $2^{64} - 1 = 18446744073709551615$ C'est très grand...
2. Pour la blague : Il existe un programme satirique `sl` *steam locomotive*.
3. Sinon ça fait un processus zombie et personne n'aime les zombies, si vous voulez une image pour illustrer Zombie processes by Daniel Stori]

3.3 Executer une commande

Pouvoir exécuter une commande avec `execvp()` sans arguments.
Réussir a pouvoir executer par exemple `ls`
Réussir à exécuter une commande avec `execvp()` avec un argument.
Réussir a pouvoir executer par exemple `ls -a`
Réussir à exécuter une commande avec `execvp()` avec arguments.
Par exemple `ls -a -h -l --`

3.4 Gérer correctement l'attente de l'enfant par son parent

Le shell ne dois pas continuer pendant que l'enfant d'execute ! Indice : `wait()`

3.5 Enrober dans une boucle l'etape 0

Pouvoir faire comme dans Etape 0 mais avec une boucle infinie qui s'arrette seulement si on fait `CTRL-C`

3.6 Faire un pipeline de commandes

Reproduire le comportement de pipe en shell `|`.

3.7 Extensions du sujet

Tout ajout pertinant : Gestion des taches en fond, gestion des redirections IO, configuration du `PROMPT`, mini-scripting. Ajout de commandes internes (*builtin*).
Faire son wrapper ASM au desuss du syscall Linux `write()` comme une libc.