

Projet v0.3.0: Théorie des Systèmes d'Exploitation

Axel Viala <axel.viala@darnuria.eu>

15 janvier 2021

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. ;)

Qualité du rendu : Un code propre, bien indenté permet d'avoir plus de points. Vous êtes libres de choisir votre style mais soyez consistant dans celui ci, il existe des outils pour formater votre code comme `clang-format` ou une option de votre éditeur

Table des matières

1	Introduction	1
1.1	Processus	2
2	Pratique - micro-shell	2
3	Execution d'un Processus	2
3.1	Lire tout le sujet - réfléchir à comment faire	3
3.2	Lire une entrée tapée au clavier	3
3.3	Executer une commande	3
3.4	Gérer correctement l'attente de l'enfant par son parent	3
3.5	Enrober dans une boucle l'etape 0	3
3.6	Extensions du sujet	3
3.7	Readme	3
3.8	Fonctions et documentation	3
3.9	Faire un pipeline de commandes	3
3.9.1	Gestion des Taches en arrière plan	4
3.9.2	Utiliser Git ou équivalent pour la gestion de versions	4
3.9.3	Bonus ou malus : warning, erreurs ou erreur mémoire	4
3.9.4	Gestion des redirections	4
3.9.5	Ajout de commandes internes	4
3.9.6	Gestion des redirections	4
3.9.7	Personnalisation du prompt	4
3.9.8	Gestion de l'historique	4
3.9.9	Gestion des variables de shell	5
3.9.10	Ma micro-lib C	5
3.9.11	Mini-scripting	5

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.

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 uns 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 tous 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_FFFF¹ 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 leurs **linkers** et assembleurs la respectent.

2 Pratique - micro-shell

Pour s'appropriier 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 doit pouvoir exécuter une commande tapée par un utilisateur, dans les faits 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 revient 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 aie bien terminé³

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**

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 - réfléchir 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()`

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'exécute! 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 Extensions du sujet

Pour aller au-delà de 13.5, vous pouvez réaliser les taches exploratoires suivantes. Tout ajout non décrit est possible pensez à le documenter !

3.7 Readme

Difficulté : facile

Réaliser un `README.md` qui décrit votre projet et votre groupe documente comment utiliser votre shell.

3.8 Fonctions et documentation

Difficulté : facile

Organiser son code en fonctions et documenter par des commentaires avant les fonctions.

3.9 Faire un pipeline de commandes

Difficulté : Moyenne Reproduire le comportement de pipe en shell |.

Pour réaliser cette commande vous aurez besoin conjointement des appels systèmes `pipe` et `dup2`.

`pipe` pour créer le canal de communication. `dup2` pour clore et dupliquer votre pipe comme `STDIN` ou `STDOUT`.

3.9.1 Gestion des Taches en arrière plan

Difficulté : Moyenne

Gestion des taches en fond par exemple `ls &` qui va s'exécuter en toile de fond et afficher quand elle aura fini. Vous aurez besoin de `waitpid`

3.9.2 Utiliser Git ou équivalent pour la gestion de versions

Difficulté : Facile

Vous avez utilisé git pour gérer votre code bravo ça fait 2 points. Un commit final énorme ne compte pas. Je veux avoir le dossier `.git` dans le rendu, ne committez pas vos binaires.

3.9.3 Bonus ou malus : warning, erreurs ou erreur mémoire

Difficulté : variable

Vous avez plus d'erreur mémoire ou de warning félicitation c'est un bonus, vous en avez selon les cas c'est du malus.

Utiliser les options `-Wall -Wextra` ou `clang` avec `-Weverything` Et `valgrind` pour chasser vos erreurs mémoires!

3.9.4 Gestion des redirections

Difficulté : Facile

Gestion des taches en fond par exemple `ls &` qui va s'exécuter en toile de fond. Vous aurez besoin de `waitpid`

3.9.5 Ajout de commandes internes

Difficulté : variable

Ajouter des commandes internes à votre shell dite (*builtin*), par exemple : `cd`, `exit`, `help`, `history`, `kill`, `pwd`.

Voir section SHELL BUILTIN COMMANDS ou équivalente de votre shell favori.

3.9.6 Gestion des redirections

Difficulté : Moyenne

Dans l'esprit du pipeline gérer les redirections entrées sorties (Input/Output) > de la `STDOUT` vers un fichier.

Exemple : `ls > resultat_ls.txt`

Bonus bonus : gérer les redirections vers la `STDIN` avec <.

Super ultra bonus : pouvoir rediriger le flux qu'on souhaite pas seulement `stdin` ou `stdout`.

3.9.7 Personnalisation du prompt

Difficulté : facile

Via une variable en C permettre de customiser son prompt avec de la couleur.

3.9.8 Gestion de l'historique

Difficulté : facile Écrire les commandes réalisées dans un fichier nommé `tes_shell_history` la ou est exécuté initialement le shell.

Aller plus loin : Les commandes devront être ajoutées en ajout depuis la fin du fichier. Encore plus loin : Récrire le début du fichier après 256 lignes.

3.9.9 Gestion des variables de shell

Difficulté : dur

Pouvoir enregistrer des variables avec la syntaxe `VARIABLENAME=VALUE` l'accès à la valeur d'une variable sera réalisée en utilisant le caractère `$`

Une valeur doit être obligatoirement entre guillemets `" "` on ira pas gérer les guillemets dans les guillemets.

Conseil utilisez un tableau de structures ou une hash table (mieux).

3.9.10 Ma micro-lib C

Difficulté : moyen

Faire son wrapper ASM au-dessus du syscall Linux `write()` comme une libc classique.

3.9.11 Mini-scripting

Difficulté : super dur

Tout ajout dans cette section est libre mais sera très récompenser ajouter des fonctionnalités, pour gérer un mini interpréteur dans votre shell, avec son micro-langage.

Ne pas vous aventurer avant d'avoir sécurisé au moins 14!;)