Benjamin Gaillard

IUP GMI 2e année
2004/2005

Projet de Systèmes d'Exploitation : Mini-Shell Interactif

Rapport

Introduction

Lish (*Lightweight Interactive SHell*) est un interpréteur de commandes de type *Bash*. Il dispose de la même syntaxe pour les commandes, et essaye d'imiter au maximum son comportement. Bien entendu, étant un petit programme, il ne possède de loin pas toutes les fonctionnalités de *Bash*, mais est tout-à-fait fonctionnel en tant que simple intérpréteur de commandes avec quelques fonctionnalités supplémentaires. Il gère les redirections de fichiers et de descripteurs, les pipelines et les commandes conditionnelles de la même manière que *Bash*.

Exécution des commandes

Prototypes

Note Les noms des types de données ont été modifiés. Ils ont été traduits en anglais, ne contiennent plus de majuscule et finissent tous par « _t ».

```
int   exec_command(command_t *command);
int   exec_sequence(sequence_t *sequence);
int   exec_conditional(conditional_t *conditional);
int   exec_pipeline(pipeline_t *pipeline);
pid_t exec_redirected(redirected_t *redirected);
pid_t exec_simple(simple_t *simple);
```

Commande

L'exécution d'une commande, c'est-à-dire d'une séquence complète, est assurée par la fonction `exec_command`. En fait, elle ne fait qu'appeler `exec_sequence` avec comme paramètre la première partie de la séquence.

Séquence

La fonction `exec_sequence` s'occupe de cette tâche. Une séquence étant une liste chaînée d'éléments de type *sequence_t*, elle les passe en revue l'un après l'autre dans une boucle, et appelle la fonction `exec_conditional` avec leur champ ***conditional*** en paramètre.

Pour les conditionnelles en séquence, l'appel est direct ; pour celles en arrière-plan, l'exécution est faite dans un *fork* du processus. Un pipe est également créé avec `pipe`. Son entrée remplacera l'entrée standard dans le fils, et sa sortie sera fermée. Dans le père, les deux descripteurs de fichier du pipe sont fermés. Cela permet à la commande s'exécutant en arrière-plan de ne pas lire l'entrée standard du shell : il recevra tout de suite une fin de fichier.

Commande conditionnelle

De même que pour la fonction précédente, `exec_conditional` est appelée avec la tête de la liste chaînée de *conditional_t*. Elle exécute, c'est-à-dire appelle `exec_pipeline` sur le membre ***pipeline*** de chaque élément de la liste, si le code de retour de l'appel précédent est compatible avec l'opération demandée (AND ou OR) (sauf, bien sûr, dans le cas du premier pipeline qui est toujours exécuté).

Pipeline

Comme précédemment, la fonction `exec_pipeline` reçoit en argument la tête de la liste des commandes redirigées formant le pipeline. Si une seule commande est présente, dans ce cas, elle est exécutée directement au moyen de `exec_redirected`. Sinon, chaque commande est exécutée avec la même fonction, mais dans un *fork* du processus courant et avec les descripteurs d'entrée et de sortie standard modifiés.

À chaque commande, de la première à l'avant-dernière, est créé un pipe avec la fonction `pipe`. L'entrée de ce pipe est gardé en réserve pour la commande redirigée suivante, et la sortie du pipe remplace la sortie standard courante. L'entrée

standard est remplacée par l'entrée du pipe créé pour la précédente commande redirigée, exception faite de la première commande qui utilise l'entrée standard du shell. De la même manière, la sortie standard de la dernière commande redirigée du pipeline est la même que celle du shell.

Commande redirigée

La fonction `exec_redirected` parcourt la liste des redirections associées à la commande redirigée passée en paramètre. Suivant le type de redirection, elle effectue différentes actions :

- c'est une redirection de fichier : dans ce cas, le fichier est ouvert dans le mode spécifié par la redirection. Le nouveau descripteur est copié et remplace l'ancien descripteur (également spécifié par la redirection) s'il existe, puis est fermé ;
- c'est une redirection de descripteur : le mode du descripteur source est d'abord vérifié, puis le descripteur est copié dans le descripteur destination, le remplaçant s'il existe ; enfin, le descripteur source est fermé si spécifié par la redirection.

Finalement, une fois ces redirection effectuées, la commande est exécutée. Au retour, les descripteurs créés sont fermés ; l'entrée standard et les sorties standard et d'erreur sont restaurées. Cela est fait car il se peut que la commande ne sont pas exécutée dans un *fork*. C'est `exec_simple` qui se charge de l'exécution de la commande simple.

Commande simple

C'est la fonction `exec_simple` qui gère les commandes simples. Les descripteurs ont déjà été mis en place par les fonctions précédentes. Il ne reste donc plus qu'à passer à l'exécution elle-même. Tout d'abord, la fonction transfore la liste chaînée de *words_t* en tableau de type ***argv***, comme utilisée dans les fonctions `main`. Puis, elle teste si cette commande est interne en tentant de l'exécuter au moyen de la fonction `exec_internal`.

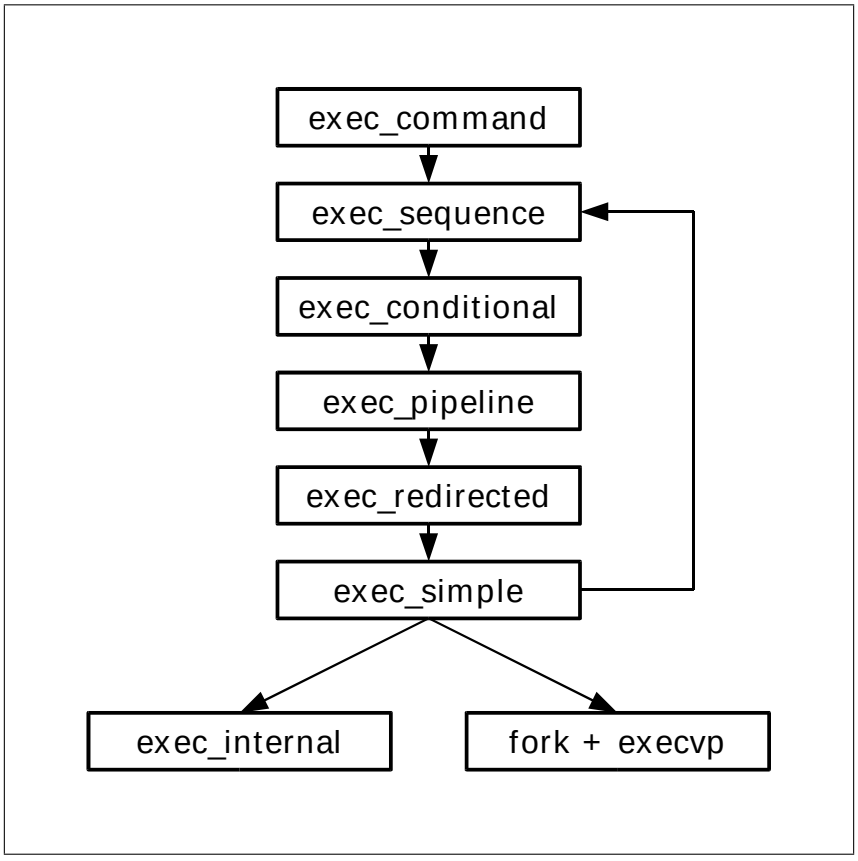
Sinon, elle appelle `fork` et utilise `execvp` pour remplacer le processus fils par celui de la commande. Dans le cas d'un sous-shell, le *fork* est également effectué, mais le fils appelle `exec_sequence` pour exécuter la commande complète du sous-shell. Dans les deux cas, le processus père attend la fin de l'exécution du fils avec `wait`.

Remarque Les paramètres commençant par « `$` » sont remplacés par les variablse d'environnement correspondantes.

Commandes internes

Les commandes internes sont exécutées via `exec_internal`. Cette fonction est appelée avant toute tentative de *fork* puis appel d'`execvp` pour tester si une commande est déjà disponible en interne. Si c'est le cas, elle est directement exécutée dans le processus courant, ce qui facilite grandement les commandes telles que `cd`, `exit`, etc. Certaines commandes supplémentaires, non présentes dans le sujet, ont été implémentées. Voir plus pas pour plus d'informations.

Schéma récapitulatif



Gestion des signaux

Lish intercepte et gère cinq signaux. Nous allons voir en qui ils consistent, et de quelle manière leur gestion est implémentée.

Terminaison : SIGTERM

Le signal SIGTERM sert à terminer un processus de façon normale. C'est la méthode standard pour mettre fin à l'exécution d'un processus, c'est aussi le signal envoyé par la commande `kill` par défaut. En fait, ce signal est tout simplement ignoré, comme le fait *Bash*. Cela permet de ne pas fermer le shell en exécutant la commande « `kill 0` ».

Terminaison d'un fils : SIGCHLD

Lorsque ce signal arrive, cela veut dire qu'un processus fils s'est terminé. Lish récupère alors son code de retour avec `wait` pour qu'il ne devienne pas un processus zombie, et affiche sur la console le PID du processus ainsi que son code de retour. Pour les fils ayant été stoppés (c'est-à-dire ayant reçu le signat SIGTSTP), le signal SIGCONT leur est tout de suite envoyé afin qu'ils soient mis en arrière-plan.

Interruptions : SIGINT, SIGQUIT, SIGTSTP

Ces trois signaux sont ignorés par Lish mais transmis à tous les fils en cours d'exécution (ceux qui sont en premier plan uniquement). SIGINT permet d'interrompre le processus (déclenchable par Ctrl+C), SIGQUIT de le terminer (Ctrl+\) et SIGTSTP de le mettre en pause (Ctrl+Z). Sur le système où le shell a été développé (*Linux*), ces signaux étaient transmis automatiquement, mais dans l'éventualité où ils ne le seraient pas sur un autre système, le gestionnaire de signal les transmet tout de même aux fils actifs au premier plan.

La liste des processus auxquels envoyer le signal est connue grâce à la fonction `exec_pipeline` qui crée et remplit un tableau des processus de premier plan en cours d'exécution.

Historique

L'historique de Lish est stocké en mémoire partagée et utilise des sémaphores pour contrôler l'accès à cette mémoire. Nous allons voir comment chacun d'eux intervient dans ce but.

Mémoire partagée

Un segment de mémoire partagée est obtenu avec `shmget` et `shmat`. Il est partagé par tous les processus en cours d'exécution par l'utilisateur. Sa clé est dépendante de l'UID de l'utilisateur, ce qui permet à plusieurs shells de s'exécuter sous plusieurs identités différentes. Le nombre d'attachements à ce segment de mémoire partagée (obtenu avec `shmctl`) sert à savoir si le shell courant est le premier (ou le dernier) à être exécuté. S'il est le premier, il initialise la mémoire partagée et charge l'historique depuis le fichier ∼/.history. S'il est le dernier, il sauvegarde l'historique et détruit le segment de mémoire partagée.

Le segment de mémoire partagée ne contient qu'une structure, formée par un tableau de lignes de commandes, ainsi que de 3 variables permettant de connaitre la position de la dernière commande tapée, de la plus ancienne et le décalage par rapport à la toute première commande tapée. Chaque accès au segment est protégé par deux sémaphores.

Sémaphores

Les sémaphores permettent de garantir qu'un seul processus à la fois a accès au segment de mémoire partagée. Ils sont au nombre de deux : un pour la lecture et un pour l'écriture.

Ils sont créés au début de l'exécution du programme : d'abord de manière exclusive, pour s'assurer qu'aucune autre instance du shell n'est en cours d'exécution (si cela réussit, alors il faut initialiser les sémaphores), ensuite de manière normale. Cela se fait au travers de la fonction `semget` (`semctl` pour l'initialisation et la destruction). La fonction `semop` servira à effectuer les opérations suivantes sur les sémaphores.

Pour qu'un processus aît accès en lecture à la mémoire partagée, il ne faut qu'aucun autre processus soit en train d'y écrire. Lorsque c'est le cas, le processus se rajoute dans le sémaphore de lecture. Dès qu'il a terminé son travail, il se retire du sémaphore.

Afin qu'un processus gagne le droit d'écrire dans la mémoire partagée, il est impératif qu'aucun autre processs soit en train de lire ou d'écrire dans la mémoire partagée. Quand ces deux conditions sont réunies, le processus se rajoute dans le sémaphore d'écriture et s'en retire une fois son travail accompli.

Fonctionnalités supplémentaires

Lish dispose de quelques fonctionnalités supplémentaires, non incluses dans le sujet. En voici une description.

Commandes supplémentaires

En sus des commandes `cd`, `kill`, `exit` et `history` sont gérées les commandes `echo`, `exec` et `export`. Voici une description de ces commandes :

- `cd [répertoire]` : change de répertoire courant ;
- `kill [-signal | -l] [PID...]` : envoie un signal à un processus ;
- `exit [code]` : quitte le shell en renvoyant un code d'erreur ;
- `history [-c]` : affiche/vide l'historique ;
- `echo [arg...]` : affiche les arguments ;
- `exec commande` : exécute la commande en remplaçant le shell ;
- `export [clé=valeur...]` : définit une variable d'environnement.

Remarque La commande `cd` met à jour la variable d'environnement *`PWD`* correspondante.

Affichage de l'invite

L'invite est basée sur la variable d'environnement *`PS1`* : la plupart des codes utilisés par *Bash* sont compatibles. Par défaut, si cette variable n'est pas définie, le shell affiche son nom suivi du caractère « `#` » si l'utilisateur est root, « `$` » sinon.

En lançant Lish avec le paramètre `-s` ou `--sexy`, l'invite par défaut est remplacée par une autre invite, affichant plusieurs informations, le tout avec de la couleur.

Conclusion

LISH est un petit shell assez complet. Il nous a permis de faire le tour des principales commandes système d'Unix, et a été une excellente manière de nous faire apprendre cette partie des systèmes d'exploitation par la pratique.