TP6

Guillaume Chanel

November 2021

1 Objectifs

L'objectif général du TP est de créer son propore shell avec lequel on pourra exécuter les programmes du système. Dans ce genre d'exercice, un moment particulièrement plaisant est lorsque l'on compile son shell à partir de son propre shell.

Les objectifs pédagogiques sont de:

- créer des processus avec la fonction *fork*;
- gérer ces processus, notament éviter les zombies;
- gérer les signaux envoyés au shell.

Ce TP ce fera sur deux scéances. Dans le cas où vous terminez la première scéance en avance nous vous recommendons de continuer directement sur la deuxième.

2 Architecture et fonctionnement du shell

Historiquement, les shells (en ligne de commande) sont les premières interfaces utilisateurs. C'est à travers un shell que l'utilisateur pour intéragir avec les système et exécuter des commandes qui lui permettent de manipuler des fichiers, des dossiers ou des processus. Le shell est donc un lien direct entre l'utilisateur et le système, c'est pouquoi c'est une exemple idéal d'utilisation d'API POSIX.

Un shell doit créer plusieurs processus, en fait un pour chaque programme executé. De plus il doit gérer ces processus ce qui implique de suivre leur état mais aussi de capturer plusieurs signaux et d'en informer ses fils correctement. Finalement le shell possède également quelque commandes internes appellées builtin. Toutes ces facettes serons implémentées lors de ce TP.

3 Execution de commandes

La première partie du TP visera à développer un shell qui lit une commande utilisateur et l'exécute. Cette commande pourra être de deux types:

- job: la ligne de commande corresponds à un programme du système (e.g. ls, pwd, ps). Ce programme sera alors exécuté en tant que job.
- builtin: ce sont des commandes qui sont implémentées directement dans le shell. Vous pouvez avoir des examples de ces commandes par *man bash*.

La suite de cette section est divisée en trois partie. Il est recommandé de développer votre programme avec un module pour chaque partie ci-dessous. Cependant des architectures différentes ne seront pas sanctionnées.

Les programmes suivants peuvent être utiles pour tester votre shell tout au long du TP:

- *sleep*: très utile pour vérifier si un processus deviens orphelin. Il suffit de lancer sleep avec un nombre important de secondes puis de terminer le shell (exit, kill, etc.) pour vérifier que le processus est bien terminé avec le shell.
- *ps –forest -f*: bon test de ligne de commande à plusieurs option et permet de voir l'état des enfants du shell (e.g. de contrôler si il y a des processus zombie / defunct).
- lancer son shell en tâche de fonds de son shell est aussi un test intéressant. Dans ce cas faites particulièrement attention au shell qui execute la commande que vous tapez (i.e. le shell en tâche de fond ou le principal ?).

3.1 Analyse syntaxique (parsing)

Une des priorité d'un shell est de lire l'entrée utilisateur (STDIN) puis d'en faire une analyse syntaxique pour déterminer le nom de la commande et ses arguments. L'objectif est donc de transformer une chaine de caractère en un tableau de chaine au format *argv* (c.f. fonction *main*), voir d'obtenir le nombre d'arguments *argc*.

Pour cela vous pouvez consulter la fonction *strtok* dans le manuel. Cette fonction divise en sous-chaines une chaine de caractère en ce basant sur des caractères de séparation. Dans notre cas on considérera que l'espace et la tabulation sont les deux seuls caractères qui séparent les arguments de notre commande. Il est également probable que vous ayez besoin de la fonction *realloc* qui permet de réallouer de l'espace mémoire dynamiquement (pour simplifier c'est l'équivalent de *free* + *malloc*).

3.2 Commandes builtin

Une fois la commande analysée et segmentée sous la forme *argv*, le shell devra tester si le premier argument (i.e. le nom du programme / de la commande) fait partie des commandes builtin et exécutera cette commande le cas échéant.

Deux commandes seront implémentées:

- *exit*: le shell ce termine proprement (c.f. jobs en tâche de fond et signaux);
- *cd*: le shell change le répertoire courant en fonction du deuxième paramètre, tout comme la commande habituelle. Notez que la commande *cd* doit être implémentée par n'importe quel shell. D'ailleurs vous pourrez constater que l'exécutable */bin/cd* n'est en fait qu'un lien vers la commande builtin d'un shell.

3.3 Jobs

Ce module permettra l'exécution de programmes du système. Lorsque la commande tapée par l'utilisateur n'est pas builtin, le shell effectuera les opérations suivantes:

- il exécutera le programme par le méthode vue en cours. L'exécutable devra être cherché dans le PATH (i.e. utilisation de la bonne fonction de la famille *exec*);
- il attendra que le programme se termine puis devra afficher le code de sortie du programme si il est disponible (e.g. "Foreground job exited with code 0") et un simple message sinon (e.g. "Foreground job exited").

Il est IMPERATIF que le shell ne laisse pas de processus sous la forme de zombies.

4 Jobs en tâche de fond et signaux

Dans cette section nous allons ajouter la possibilité d'exécuter des jobs en tâche de fonds. Toutefois, par souci de simplicité, on interdira d'avoir plus d'un job en tâche de fond à la fois. Notre shell ne

pourra donc exécuter que deux jobs: un en tâche de fonds et un en tâche principale. Dans notre shell, une commande builtin ne sera jamais exécutée en tâche de fond.

Pour exécuter un job en tâche fond le shell devra:

- reconnaitre le '&' à la fin de la commande et l'éliminer des paramètres du programme à exécuter;
- rediriger l'entrée standard du job vers */dev/null* pour éviter les conflits avec le shell;
- lancer le programme de manière similaire à un job principal;
- ne PAS attendre que le job soit terminé et rendre la main à l'utilisateur;
- lorsque le job est terminé s'assurer qu'il ne deviens pas un zombie et afficher "Background job exited".

Le dernier point est particulièrement difficile car il faudra savoir quand le job / processus ce termine. Cela peut-être implémenté en utilisant un handler sur le signal SIGCHLD. Ce handler sera de type *sa_sigaction* afin de pouvoir identifier le pid du processus et le comparer au pid du job en tâche de fond.

Finalement on veillera a ce que le shell gère les signaux de la manière suivante:

- SIGTERM et SIGQUIT seront ignorés (c'est le comportement standard des shells).
- SIGINT sera redirigé vers le processus principal. De cette manière l'utilisateur pourra utiliser Ctrl+C pour arrêter un job principal sans stoper le shell. Presser Ctrl+C dans un shell sans job principal est sans effet.
- SIGHUP sera redirigé vers TOUS les processus/jobs du shell et terminera le shell. Historiquement SIGHUP était envoyé lorsque la connection avec le terminal était perdu, aujourd'hui il est utilisé dans le cas ou la fenêtre du terminal est fermée. Il faut alors s'assurer de terminer le shell proprement (i.e. pas de processus orphelins).

Attention: bien que les fonctions de la famille *wait* soient bloquantes, la récéption d'un signal débloque la fonction. Voir l'option SA_RESTART de la fonction *sigaction* pour résoudre partiellement ce problème.

5 Pour aller plus loin

Notez que les shells possèdent de MULTIPLES autres fonctionnalités qui ne seront pas évaluées dans ce TP. Parmis celles-ci on peux citer:

- l'exécution de plusieurs jobs en tâche de fond ainsi que leur gérance (e.g. remettre un job en tâche principale).
- la redirection des entrées / sorties (i.e. symbole '>').
- l'implémentation de pipes (i.e. symbole '|').