

TP1

Guillaume Chanel

March 2022

1 Objectifs

L'objectif général du TP est de créer son propre 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 grâce aux mécanismes vus en cours;
- gérer ces processus, notamment éviter les zombies;
- gérer les redirections ('>') et les tubes ("pipes", '|').
- gérer les jobs en tâche de fond et les signaux ('&').

Ce TP se fera sur plusieurs séances et sera noté.

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 interagit avec le système et exécute 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 pourquoi c'est une exemple idéal d'utilisation d'API POSIX.

Un shell doit créer plusieurs processus, en fait un pour chaque programme exécuté. De plus il doit gérer ces processus ce qui implique de suivre leur état. Le shell possède également quelques commandes internes appelées builtin. Finalement, un shell permet aussi de rediriger des entrées / sorties vers un fichier, ou vers un autre processus à travers les "pipes". Toutes ces facettes seront 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 correspond 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 exemples de ces commandes par *man bash*.

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

- `ps --forest -f`: bon test de ligne de commande à plusieurs options et permet de voir l'état des enfants du shell (e.g. de contrôler si il y a des processus zombie / defunct).

- `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.
- 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 avez à disposition un module livré par l'enseignant et disponible sur cyberlearn. Ce module fait usage de la fonction *strtok* qui divise en sous-chainnes 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. Le module utilise également la fonction *realloc* qui permet de réallouer de l'espace mémoire dynamiquement (pour simplifier c'est l'équivalent de *free* + *malloc*).

Vous êtes libre d'utiliser ce module ou de créer votre propre analyse syntaxique. Dans tous les cas il est intéressant de lire le code donné pour comprendre son fonctionnement et son utilisation.

3.2 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 créera un nouveau processus grâce aux appels systèmes vu en cours;
- il exécutera le programme par le méthode vue en cours. L'exécutable devra être cherché dans le PATH;
- il attendra que le programme se termine puis devra afficher le code de sortie du programme en affichant la phrase "Foreground job exited with code X" uniquement si le programme c'est terminé normalement avec le code de retour X.

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

3.3 Commandes builtin

En utilisant 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.

Les commandes suivantes seront implémentées:

- *exit*: le shell ce termine proprement;
- *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.

4 Manipulation des entrées / sorties

4.1 Redirections d'entrées / sorties

Il s'agit ici de rediriger la sortie ou l'entrée standard d'un job vers un fichier en utilisant les symboles '>', '>>' et '<'. Uniquement les symboles séparés par des espaces seront considérés (i.e. il doit y avoir un espace avant et après les symboles). Les IO des commandes builtin ne seront pas redirigées. Pour cela il faudra:

- identifier les symboles '>' dans un des arguments de la chaîne *argv*, comme effectué par le module d'analyse syntaxique. À noter que l'unique argument suivant le symbole sera considéré comme un chemin vers un fichier. Si plusieurs arguments suivent le symbole '>' alors uniquement le premier sera considéré. Ceci est effectué par le parser fourni;
- ouvrir le fichier indiqué après '>' avec les bons paramètres et récupérer le descripteur de fichier correspondant;
- entre la création du nouveau processus et l'exécution du nouveau programme, il faudra manipuler les descripteurs de fichier par un des moyens vu en cours. L'idée est de s'assurer que tout ce qui est écrit sur le descripteur 1 (STDOUT_FILENO) est redirigé vers le fichier.

Le résultat pourra être testé avec la sortie de tout job affichant (normalement) quelque chose à l'écran. L'implémentation de redirection de type '>' est suffisante pour valider le TP. Toutefois vous pourrez si vous le souhaitez implémenter les redirections de type '>>' et '<'.

4.2 Pipes

De la même manière que le symbole '>' était détecté précédemment, il va falloir cette fois détecter le symbole '|' qui séparera nécessairement deux jobs. Pour simplifier nous nous limiterons à l'implémentation de deux jobs dont la sortie de l'un est redirigé vers l'entrée de l'autre (i.e. au maximum un symbole '|' avec un espace avant et après). Une méthodologie par pipe simple sera utilisée:

- identifier le symbole '|' dans la suite de paramètres *argv*; ce symbole sépare deux jobs. Ceci est effectué par le parser fourni;
- créer un tube (pipe) anonyme en utilisant l'appel système *pipe*;
- créer deux processus correspondants aux deux commandes séparées par le '|', ces processus doivent faire partie du même groupe de processus;
- comme pour les redirections il faudra s'assurer que les descripteurs de fichiers de chaque processus soient correctement configurés pour que le pipe fonctionne;
- **Attention**, pour que les processus se terminent correctement il faut veiller à ce que chaque processus (i.e. le parent et les deux enfants ferment les descripteurs dont ils n'ont plus besoin).

Toujours pour simplifier on considérera qu'il n'est pas possible d'avoir en même temps une redirection vers un fichier et un pipe. De même il ne sera pas possible de mettre une commande contenant un pipe en tâche de fond (voir section suivante). Si le symbole pipe est présent sur une commande en même temps que un autre ('>' ou '&') alors uniquement le pipe sera exécuté (i.e. l'autre sera ignoré).

5 Jobs en tâche de fond et signaux

Dans cette section nous allons ajouter la possibilité d'exécuter des jobs en tâche de fond. 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 de fond le shell devra:

- reconnaître le '&' à la fin de la commande et l'éliminer des paramètres du programme à exécuter. Ceci est effectué par le parser fourni;
- 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 devient pas un zombie et afficher le message "Background job exited with code X" uniquement si le job c'est terminé normalement avec le code de retour X.

Le dernier point est délicat 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 à ce que le shell gère les signaux de la manière suivante:

- SIGTERM et SIGQUIT seront ignorés (c'est le comportement standard de bash mais cela peut être différent pour d'autres shells).
- SIGINT sera redirigé vers le(s) processus composant le job principal. De cette manière l'utilisateur pourra utiliser Ctrl+C pour arrêter une tâche principale sans stoper le shell. Presser Ctrl+C dans un shell sans job principal doit être 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 réception d'un signal débloque la fonction. Voir l'option SA_RESTART de la fonction *sigaction* pour résoudre ce problème.