

# TP1

Guillaume Chanel

March 2020

## 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 avec la fonction *fork*;
- gérer ces processus, notamment éviter les zombies;
- gérer les redirections ('>') et les "pipes" ('|').

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 peut interagir avec le 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 pourquoi c'est un 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).

- `find . -exec ls -ld \;` : liste récursivement tous les fichiers du dossier courant avec leurs permissions, tailles, etc. Utile pour tester votre commande `rls` et pour avoir une commande fournissant un gros output (pour les redirections et "pipes").

### 3.1 Analyse syntaxique (parsing)

Une des priorités 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 chaîne de caractères en un tableau de chaînes 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-chaînes une chaîne de caractères en se 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 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 la 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.

### 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 se 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.
- *pwd*: affiche le répertoire courant du shell.
- *rls*: liste récursivement le contenu du dossier courant et de tous les sous-dossiers en indiquant les informations des fichiers/dossiers dans le même format que *ls -l*. La commande suivante permet d'avoir un exemple du résultat attendu: `find . -exec ls -ld \;`.

### 3.4 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 le symbole '>' dans un des arguments de la chaîne *argv*, l'unique argument suivant le symbole sera considéré comme un chemin vers un fichier. Si plusieurs arguments suivent la chaîne alors uniquement le premier sera considéré;
- ouvrir le fichier indiqué après '>' et récupérer le descripteur de fichier correspondant;
- une fois le nouveau processus créé (*fork*) mais avant d'exécuter le job (*exec*) il faudra manipuler les descripteurs de fichiers avec *dup2* pour 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 obtenir des bonus en implémentant les redirections de type ('>>' et '<').

### 3.5 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 nommé sera utilisée:

- identifier le symbole '|' dans la suite de paramètres *argv*; Ce symbole sépare deux jobs (i.e. deux *fork* et *exec*);
- créer un fifo nommé dans le répertoire /tmp en utilisant les fonctions permettant d'avoir un nom de fichier temporaire;
- ouvrir ce fichier pour obtenir un descripteur puis le supprimer immédiatement (le lien pas le descripteur de fichier);
- comme pour les redirections il faudra s'assurer que les descripteurs de fichiers de chaque processus soient correctement configurés pour que le pipe fonctionne.

Toujours pour simplifier on considérera qu'il n'est pas possible d'avoir en même temps une redirection vers un fichier et un pipe. Si les deux symboles sont présents sur une commande alors uniquement le pipe sera exécuté (i.e. la redirection sera annulée).

Que se passe-t-il si deux jobs tentent d'ouvrir un pipe en lecture ? Comment résoudre ce problème ? A votre avis comment les pipes sont implémentés dans d'autres shell ? (laisser les réponses comme commentaire dans la plateforme moodle, ne pas implémenter).