

# TP 7

SYSTEM INFORMATIQUE

# INFORMATIONS GÉNÉRALES

- Références au cours :
  - 10.Processus
- Date de reddition : 11 decembre 2018 à 23:59
- Délivrables
  - Code source commenté
  - Dossier à rendre : <Prenom1>.<Nom1>.< Prenom2>TP7.zip(ou tar.gz)

# OBJECTIF

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:

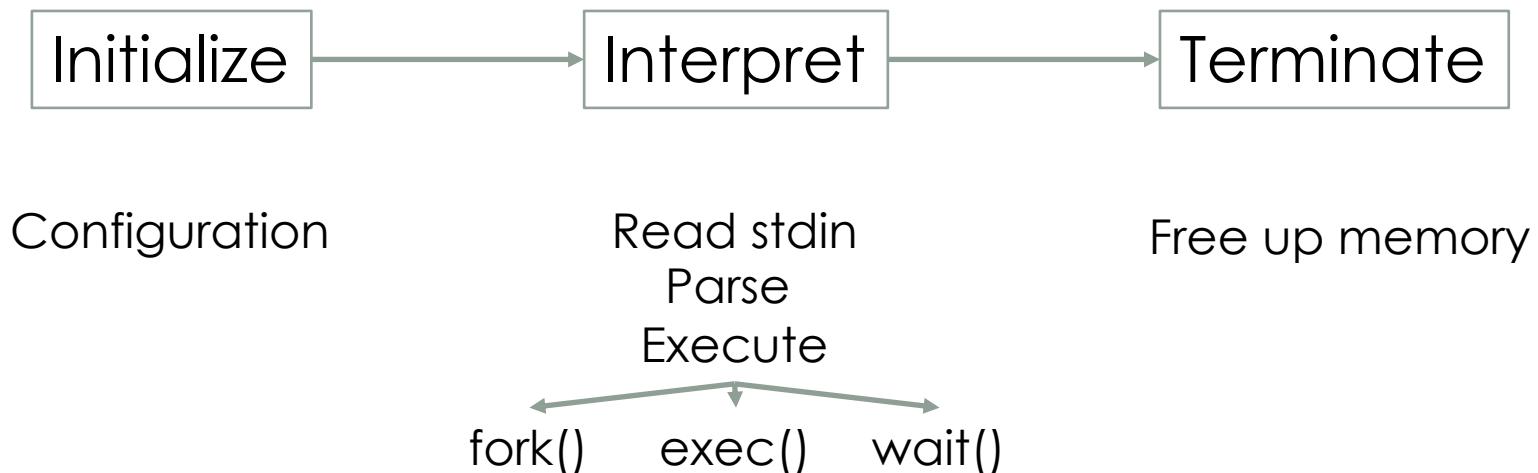
- créer des processus avec la fonction *fork*;
- gérer ces processus, notamment éviter les zombies;
- gérer les signaux envoyés au shell.

Ce TP ce fera sur deux séances. Dans le cas où vous terminez la première séance en avance nous vous recommandons 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 pourra 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 mais aussi de capturer plusieurs signaux et d'en informer ses fils correctement. Finalement le shell possède également quelques commandes internes appelées *builtin*. 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 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 exemples 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.

### 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 chaîne de caractère en un tableau de chaîne 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 caractère en 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.

# FUNCTIONS

- #include <sys/wait.h>
  - waitpid()
- #include <unistd.h>
  - chdir()
  - fork()
  - exec()
  - pid\_t
- #include <stdlib.h>
  - malloc()
  - realloc()
  - free()
  - exit()
  - execvp()
  - EXIT\_SUCCESS, EXIT\_FAILURE
- #include <stdio.h>
  - fprintf()
  - printf()
  - stderr
  - getchar()
  - perror()
- #include <string.h>
  - strcmp()
  - strtok()
- #include <glib.h>
- #include <stdbool.h>
- #include <signal.h>
- #include <fcntl.h>
- #include <sys/types.h>
- #include <sys/stat.h>

### 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.