R. Carpa, S. Castellan et V. Villebonnet

# Projet 1: un *shell*

Ce projet encadré se déroulera sur 3 séances de TP. Il est impératif de finir les requis de la séance *n-1* en arrivant à la séance *n*, mais l'avancement plus rapide n'est pas interdit.

La version finale sera à rendre avant le début de la 6-ième séance de TD (16/03/2016 à 8h). Le rendu sera noté. La note du projet sera tronquée sur 20.

**Modalités de rendu** (**1pt**) :

Une seule archive avec le nom *NOM_Prenom.zip* ou *NOM_Prenom.tar.gz* sera envoyée à vos encadrants de TD dans un e-mail avec le sujet "[ASR2] Rendu Projet1 de NOM Prenom".

L'archive **doit** contenir un seul répertoire : *NOM_Prenom* respectant l'arborescence qui vous est fournie. Le projet doit se compiler avec la commande *make* à la racine du répertoire et doit générer un seul exécutable *main* dans ce même répertoire.

Un fichier README.txt est impératif et doit, au moins, contenir quelques phrases expliquant quelles parties du sujet ont été réalisées ou non. De plus, c'est dans ce même fichier que vous devrez répondre aux questions qui vous le demandent explicitement.

## TP3 (6pts + 1pt)

Le but du TP3 est de créer un *shell* capable d'exécuter une commande à travers le *builtin exec*. Cela signifie que le *shell* remplace son propre processus par le processus du programme à exécuter. Si le programme fini l'exécution, la ligne de commande ne revient plus.

Pour simplifier, nous considérons qu'il n'y a pas de caractères spéciaux dans les lignes de commande tapées. Les commandes contiennent uniquement les caractères [A-Za-z0-9]. Seul l'espace est utilisé comme séparateur.

**Note** : Vous devez veiller à la bonne gestion des erreurs. Le code de retour de tous les appels système doit être testé. Une erreur ne doit pas avoir d'effet indésirable. Il est préférable de simplement notifier l'utilisateur de l'erreur et ne pas exécuter la commande.

**Question 1**

Localisez la fonction *run_command()* dans le fichier *src/shell_skell.c* qui vous est fourni. Cette fonction prend en unique paramètre une chaîne de caractères qu'elle doit *parser* afin d'extraire la commande à exécuter et ses paramètres.

A noter que la chaîne de caractères peut contenir plusieurs commandes séparées par un pipe '|' (redirection d'entrées-sorties). On ne s'occupera pas des autres cas comme l'enchaînement de commandes (&&); l'exécution impérative d'une liste de commandes ';', etc.

Pour pouvoir gérer le pipe, la fonction *run_command()* va mettre à jour la chaîne de caractères pointée par son unique paramètre *command_ptr* pour qu'elle pointe vers le début de la commande suivante.

Exemples: Si en entrée *command_ptr* vaut "exec firefox", à la fin de la fonction la chaîne devient la chaîne vide "\0". Si *command_ptr* vaut "ls -l | grep toto", elle devient "grep toto" à la fin de la fonction.

En cas d'erreur, la fonction retourne -1, sinon elle doit renvoyer 0.

Utilisez *strtok_r(3)* pour extraire la partie de la chaîne de caractères contenant la première commande. Ensuite, utilisez cette même fonction sur la sous-chaîne restante pour extraire les arguments qui sont séparés par des espaces.

**Question 2**

Complétez la fonction *run_command()* pour qu'elle teste si la commande à exécuter est un *builtin*. Pour cela, voir la structure *bltins* fournie dans le squelette. Si le premier mot de la ligne est un *builtin*, la fonction correspondante est appelée, avec pour arguments les autres mots de la ligne de commande.

**Note** : Un exemple d'itération sur la structure *bltins* peut être trouvé dans la fonction *builtin_help()*.

**Question 3**

Complétez les fonctions *builtin_cd()* et *builtin_pwd()*. Dans le cas de *cd*, n'oubliez pas de mettre à jour la variable globale *wd* avec le nouveau chemin. Les fonctions *chdir(2)* et getcwd(3) vous seront utiles.

**Question 4**

(Répondre dans le README) Dans l'exemple suivant vous pouvez remarquer que pour les commandes *ls* et *cd*, le répertoire symbolicLinkToProj/.. représente des endroits différents dans l'arborescence. Donnez une explication de ce phénomène.

```
$ ls
some_file.txt
$ ln -s /tmp/ASR2/Proj/ symbolicLinkToProj
$ ls symbolicLinkToProj/..
Proj  tds
$ cd symbolicLinkToProj/..
$ ls
some_file.txt  symbolicLinkToProj
```

**Question 5**

Complétez la fonction *builtin_exec()*. Le manuel des fonctions *exec** (man 3 exec) vous sera utile. Vous utiliserez probablement la fonction *execvp*. Ne vous préoccupez pas du cas où le *builtin exec* est appelé sans arguments.

**Question 6**

**Bonus** : Modifiez le code de *builtin_exec()* pour qu'il soit possible de faire une redirection de la sortie standard avant d'effectuer l'*exec*.

Pour cela, votre programme doit détecter le caractère ">" suivi d'un nom de fichier. *open(2)* sera utilisé pour ouvrir le fichier. Ensuite, *close(2)* et *dup2(2)* permettront de remplacer la sortie standard par le descripteur de fichier retourné par *open(2)*.

**Note** : Pour simplifier, nous considérons que le caractère ">" est toujours précédé et suivi d'un espace.

## TP4/5 (7pts + 2pts)

**Question 1**

L'appel système *fork* permet de créer une copie du processus courant. Chacun des processus : l'initial, que l'on appelle le processus père et le nouvellement créé, le processus fils, continuent donc à s'exécuter indépendemment. Habituellement, l'appel à *fork* est suivi d'un test *if* pour savoir si l'exécution continue dans le processus père ou dans le fils.

Complétez la fonction *run_command()* : si la commande à exécuter n'est pas un *builtin*, il faut faire un appel à *fork*. Dans ce cas, le processus père continue d'exécuter le *shell*, pendant que le processus fils exécute la commande voulue via un appel à *execvp*. En cas de succès, la fonction doit renvoyer 0 pour un *builtin*, ou le PID du processus fils sinon.

**Question 2**

Vous pouvez remarquer que l'invite de commande revient immédiatement, même si l'exécution de la commande n'est pas terminée. Utilisez un appel à *waitpid* dans le processus père pour attendre la terminaison du processus fils avant d'afficher l'invite de commande de nouveau.

**Question 3**

(Répondre dans le README) Ne pas faire d'appel à *waitpid* engendre des problèmes plus graves qu'une invite de commande qui revient immédiatement. En analysant la documentation, expliquez pourquoi c'est important de faire un *waitpid*.

Lancez un processus "long" dans votre *shell* (par exemple, `sleep 100`, qui effectue une pause de 100s). Si vous faites *ctrl+c* pour l'arrêter, vous remarquez que le *shell* est également arrêté.

L'appel *signal(3)* permet d'intercepter des signaux et de leur attribuer des fonctions *callback*. Ainsi, si un signal est délivré a votre application, comme lors d'un *ctrl+c* par exemple, la fonction correspondante *callback* que vous aurez définie sera appelée. Une fonction "par defaut"(`SIG_DFL`) et une pour ignorer le signal(`SIG_IGN`) sont déjà proposées.

**Question 4**

(Répondre dans le README) A quoi correspond le signal `2` ?

**Question 5**

Complétez la fonction *main* pour désactiver (ignorer) les signaux `2` et `SIGQUIT` avant de commencer la boucle infinie du *shell*.

**Note** : Sachant que le masque de signal est hérité par le processus fils après un *fork*, n'oubliez pas de ré-activer les signaux dans le processus après le *fork* et avant l'*execvp*.

**Question 6**

Localisez la fonction *expand_arguments()*. Complétez la pour qu'elle gère le caractère spécial '`*`' (précédé et suivi d'un espace). Elle doit remplacer l'étoile par la liste des noms de fichiers situés dans le répertoire courant et dont le nom ne commence pas par un point. Vous devrez utiliser *opendir* et *readdir*. Exemple d'utilisation :

```
DIR *dir = opendir("/chemin/du/repertoire/");
struct dirent *entry;

while ((entry = readdir(dir)))
{
    //entry->d_name contient le nom du fichier
    printf("%s\n", entry->d_name);
}
```

**Note** : Lors de vos tests, vous pouvez avoir des résultats inattendus si les noms de fichiers du répertoire courant contiennent des espaces. Nous ne demandons pas de traiter ce cas. Évitez-le pendant les tests.

**Question 7**

**Bonus** : Implémentez le caractère spécial `&` qui permet de lancer l'exécution d'une commande en arrière-plan.

L'implémentation de cette fonctionnalité est plus complexe qu'on pourrait penser. Il faut exécuter *waitpid* sur le processus fils qui a fini son exécution en arrière-plan. Pour cela, il est nécessaire d'attribuer une fonction de gestion spécifique au signal `SIGCHLD`.

De plus, il est déconseillé d'utiliser l'appel système *signal* dans les nouveaux programmes. Une alternative plus flexible existe : *sigaction*. Utilisez *sigaction* pour intercepter le signal de fermeture d'un fils et appeler *waitpid* sur le PID du fils.

**Note**: Vous devrez donc supprimer l'appel à *waitpid* que vous avez fait dans une des questions précédentes.

**Question 8**

**Bonus** : Créez une fonction *expand_ arguments2()* qui fait appel à *glob(3)* pour faire le remplacement de pattern à votre place. Remplacez l'appel à *expand_ arguments()* par un appel à *expand_ arguments2()*.

**Note**: Ne supprimez pas la fonction *expand_ arguments()*.

# TP5 (6pts + 1pt)

**Question 1**

Nous voulons implémenter le *pipe* '|' et ainsi rendre possible l'exécution de cette commande :

```
echo toto | sed s/toto/titi/ | sed s/t//g
```

Les appels système *pipe* et *dup2* devront être utilisés. Un nombre quelconque de processus devra être accepté par votre *shell*. Une attente correcte de la fin de chacun des processus avec *waitpid* est requise.

**Question 2**

(Répondre dans le README) Soit la commande : `echo toto | read a ; echo $a`.
Qu'est ce que cette commande est supposée faire ?

Testez la commande dans *bash*. Est-ce qu'elle donne le résultat attendu ? Développez votre réponse.

**Question 3**

**Bonus** : Rendez possible l'exécution d'un script dans un fichier ou passé en argument à travers l'option "-c".