Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TP n°5 - APNEE : Réalisation d'un mini-*shell*

Ce TP consiste à réaliser un mini-*shell* pour Unix.

1 Introduction

Un système d'exploitation fournit à ses utilisateurs une interface de programmation comprenant des fonctions de création de processus, de manipulation des E/S, de travail avec les fichiers, etc. Ces *appels système* peuvent être faits dans un programme C quelconque (en utilisant les fonctions de bibliothèque standard, p.ex. `fork`, `open`, `dup`, ...) ou alors en ligne de commande. Dans le deuxième cas, un interprête de langage de commande qui transforme une commande tapée sous forme textuelle en un ou plusieurs appels système. Dans le système Unix, l'interprète du langage de commande est appelé un *shell*. Des exemples de shell sont `tcsh`, `bash`, `ksh`, etc.

2 Le langage de commande (rappels et compléments)

Une commande est une suite de mots séparés par un ou plusieurs espaces. Le premier mot d'une commande est le nom de la commande à exécuter et les mots suivants sont les arguments de la commande. *Chaque commande doit s'exécuter dans un processus autonome, fils du processus shell. Le shell doit attendre la fin de l'exécution d'une commande.* Les appels systèmes `wait` et `waitpid` permettent de réaliser cette attente et de récupérer la valeur de retour de la commande (`status`).

Une séquence est une suite de commandes séparées par le délimiteur "|" ; la sortie standard d'une commande doit alors être connectée à l'entrée standard de la commande suivante. Une telle connexion s'appelle en Unix un tube. La valeur de retour d'une séquence est la valeur de retour de la dernière commande de la séquence.

L'entrée ou la sortie d'une commande (ou l'entrée de la première commande d'une séquence ou la sortie de la dernière commande d'une séquence) peuvent être redirigées vers des fichiers. On utilise pour cela les notations usuelles d'Unix :

— `< toto` : redirige l'entrée standard vers le fichier `toto`
— `> lulu` : redirige la sortie standard vers le fichier `lulu`

Voici quelques exemple de séquences de commandes, avec ou sans redirection :

```
ls -a
ls -a >toto
ls jacques | grep en
cat < fichier1 | grep en > fichier2
```

3 Travail à réaliser

Le but du mini-projet est de réaliser un interprète pour un langage de commande simplifié. L'objectif est d'une part de comprendre la structure d'un shell et d'autre part d'apprendre à utiliser quelques appels systèmes importants, typiquement ceux qui concernent la gestion des processus, les tubes et la redéfinition des fichiers standards d'entrée et de sortie.

3.1 Analyse des lignes frappées au clavier

La procédure de lecture d'une ligne et son analyse vous sont fournies (fichiers `readcmd.h` et `readcmd.c`). Un programme `shell.c` est également joint pour vous aider à comprendre ce qui est renvoyé par `readcmd()`.

La fonction `readcmd()` renvoie un pointeur vers une structure `struct cmdline` dont les champs sont les suivants :

- `char *err` : message d'erreur à afficher, null sinon
- `char * in` : nom du fichier pour rediriger l'entrée, null si pas de redirection
- `char *out` : nom du fichier pour rediriger la sortie, null sinon
- `char *** seq` : une commande est un tableau de mots (`char **`) dont le dernier terme est un pointeur null ; une séquence est un tableau de commandes (`char ***`) dont le dernier terme est un pointeur null.

La complexité de la structure n'est qu'apparente ; vous vous apercevrez lors de la réalisation du programme qu'elle est très bien adaptée, tout particulièrement pour les appels à `execvp`.

3.2 Organisation du travail

Il vous est demandé de programmer un shell qui peut interpréter les commandes écrites avec le langage défini dans 2. Vous devez passer par/réaliser successivement les étapes suivantes :

1. *Compréhension de la structure cmdline et des résultats de readcmd.*
   Lisez attentivement les fichiers fournis et appropriez vous le code. N'hésitez pas à poser des questions sur la logique ou sur le langage C.
2. *Commande pour terminer le shell*
   Pour cette étape, vous devez modifier le code fourni de telle manière à implémenter une commande `quit` qui termine proprement votre shell. Vous pourrez valider cette première étape avec le fichier de test `test01.txt` (voir section suivante).
3. *Interprétation de commande simple*
   Pour cette étape, vous devez utiliser vos connaissances sur les fonctions de création de processus, notamment la famille de la commande `exec`, que nous avons vu en TP. Se référer aux points techniques à la fin du sujet et aux compléments fournis.
4. *Commande simple avec redirection d'entrée ou de sortie*
   Pour cette étape, vous devez utiliser vos connaissances sur les fonctions de redirection d'entrée/sortie, notamment `dup` et `dup2`.

5. *Gestion des erreurs*

   Tout au long de l'implémentation, vous devrez gérer correctement les erreurs en affichant un message adapté sur la sortie d'erreur. Par exemple, si l'utilisateur demande à votre shell d'exécuter une commande innexistante (e.g., `dvfefa`), celui-ci affichera le message d'erreur : `dvfefa: command not found` ou si l'utilisateur redirige la sortie standard sur un fichier dont il ne dispose pas des droits d'écriture, votre shell affichera un message du type `monFichier.txt: Permission denied`.

6. *Séquence de commandes composée de deux commandes reliées par un tube*

   Pour cette étape, vous devez utiliser vos connaissances sur la gestion de tubes, notamment `pipe`.

7. *Séquence de commandes composée de plusieurs commandes et de plusieurs tubes*

   Pour cette étape vous réutiliserez les points traités précédemment.

8. *Exécution de commandes en arrière-plan*

   Lorsqu'une commande est terminée par le caractère &, elle s'exécute en tâche de fond, c'est à dire que le shell crée le processus destiné à exécuter la commande, mais n'attend pas sa terminaison.

   Remarque. La détection du caractère & dans une ligne de commande implique une modification de la fonction readcmd() et de la structure cmdline.

9. *Gestion des zombis*

   Une tâche lancée en arrière plan ne doit pas être attendue par le shell. Néanmoins, le shell doit ramasser les processus terminés pour éviter la prolifération de zombis. Pour cela implémenter un traitant de `SIGCHLD`. Utiliser les options suivantes de la primitive waitpid : `waitpid(-1, &status, WNOHANG|WUNTRACED)`

4 Tests

En parallèle de l'implémentation de votre shell, il vous faudra écrire des fichiers de tests pour valider la bonne implémentation de celui-ci. Pour ce faire, nous vous avons mis à disposition un script perl (`sdriver.pl` [1]).

4.1 Fonctionnement du script de test

Le script de test lit un fichier texte contenant les instructions à envoyer à votre shell (commandes et/ou signaux). Le fichier de test devra contenir un header décrivant brièvement l'objet du test puis une série d'instructions permettant de réaliser ce test (une instruction par ligne). Le script vous permet soit d'exécuter des commandes classiques (e.g., `echo titi > toto.txt`) soit d'intéragir avec votre shell avec des instuctions spécifiques :

— Les instructions `TSTP`, `INT`, `QUIT`, et `KILL`, vous permettent d'envoyer respectivement les signaux SIGTSTP, SIGINT, SIGQUIT et SIGKILL à votre shell.

— L'instruction `CLOSE` pour envoyer `EOF` à votre shell (équivalent à un CTRL+D)

1. Script emprunté à R. E. Bryant, D. O'Hallaron. *Computer Systems : a Programmer's Perspective*, Prentice Hall, 2003.

— L'instuction `WAIT` permet d'attendre la fin d'exécution de votre shell
— L'inscription `SLEEP <n>` permet d'espacer deux instuctions de `<n>` secondes.

Pour tester votre shell avec les instructions contenues dans le fichier `test01.txt`, il vous faut exécuter `sdriver.pl` avec les paramètres suivants : `./sdriver.pl -t test01.txt -s ./shell`

4.2 Exemples de fichiers de test

A titre d'exemples, nous vous avons fournis 4 fichiers de test :
— `test01.txt` permet de tester l'implémentation de la commande quit. Il contient la commande `quit` suivie de l'instruction `WAIT` permettant de tester que le shell a bien terminé son exécution
— `test02.txt` permet de tester l'exécution d'une commande sans paramètre. Il fait appel à la commande `ls`.
— `test03.txt` permet de tester l'exécution d'une succession de commandes sans paramètre. Il exécute successivement les commandes `ls`, `echo`, et `ls`.
— `test04.txt` permet de vérifier que le shell ne crée pas de zombies. Il exécute successivement les commandes `ls`, `echo`, `ls`, `echo`, et `ps`.

5 Pour aller plus loin

Inutile d'implémenter les fonctionnalités de cette section si celles de la section 3 ne sont pas entièrement implémentées et fonctionnelles. Il vaut mieux aller moins loin dans le TP et avoir un mini-shell fonctionnel que l'inverse.

La suite des questions ont pour but de se rapprocher à une gestion de travaux (*jobs*) réaliste i.e. analogues à celle des *shells* Unix courants) :
— Les commandes exécutées en arrière-plan s'appellent des *jobs*[2]. Un *job* peut être désigné par le `PID` du processus qui l'exécute (exemple 14567) ou par son numéro de *job* précédé de `%` (exemple `%3`). Les numéros de *jobs* sont des entiers positifs, attribués à partir de 1.
— La frappe de *control-c* et *control-z* doivent respectivement envoyer un signal `SIGINT` et un signal `SIGTSTP` au(x) processus de premier plan (voir détails plus loin).
— La commande intégrée `jobs` doit lister tous les *jobs* avec leur état, le texte de la commande qu'il exécutent et le `PID` du processus qui les exécute. Exemple :

```
<myshell> jobs
[1] 3456  Stopped   myprog param1 param2
[2] 3460  Running   /usr/local/bin/nedit prog.c
<myshell>
```

— Les commandes intégrées `fg`, `bg` et `stop` s'appliquent à un *job*, désigné soit par son numéro de *job* soit par son numéro de processus. Les effets de ces commandes sont résumés sur la figure vue dans le TD n°4 (et le cours n°2), qui rappelle les états des travaux et les transitions entre ces états.

2. Par abus de langage, on utilisera le terme *job de premier plan* pour désigner une tâche en cours d'exécution au premier plan (soit parce qu'elle a été lancée directement en premier plan, soit parce qu'elle a été positionnée en premier plan ultérieurement).

Noter qu'un processus lancé en premier plan ou en arrière plan peut créer des processus fils. Par défaut, ceux-ci forment avec leur père un groupe, dont le numéro *pgid* est le *pid* du père. Les commandes (`bg`, `fg`, `stop`) et les signaux générés par *control-c* et *control-z* s'adressent en fait à tous les processus du groupe de premier plan.

5.1 Structures de données

Un travail (*job*) sera représenté par une structure contenant les éléments suivants : numéro du *job* (entier > 0), *pid* du processus qui l'exécute, état du *job* (cf. figure), ligne de commande exécutée par le *job* ainsi que toute autre information jugée utile.

Pour simplifier la programmation, il est conseillé d'utiliser un tableau pour contenir ces structures, plutôt qu'une liste chaînée. La taille de ce tableau (`MAXJOBS`) est un paramètre du système (le prendre de l'ordre de 10).

Programmer des fonctions d'accès pour donner notamment le numéro de la première case libre du tableau, le numéro du *job* de premier plan (s'il existe), ajouter un nouveau *job*, supprimer un *job* et libérer la case correspondante du tableau, initialiser le tableau.

5.2 Gestion des travaux

La gestion des *jobs* est le travail principal demandé.

Comme dans les *shells* usuels, `fg` envoie un signal `SIGCONT` au *job* et le fait exécuter au premier plan. La commande `bg` envoie un signal `SIGCONT` au *job* et le fait exécuter en arrière-plan. Noter que :

1. La différence entre un *job* d'arrière-plan et le *job* de premier plan est que la fin de ce dernier (ou plutôt sa disparition du premier plan) est attendue par son père (le processus qui exécute le *shell*). Il faut ici considérer deux points :
   — Il y a plusieurs manières pour le processus de premier plan de cesser d'être au premier plan : suspension (suite à la réception d'un signal `SIGSTOP` ou éventuellement `SIGTSTP`), mort par `exit`, mort causée par un traitant de signal. Dans tous ces cas, le système d'exploitation envoie un signal `SIGCHLD` à son père (le processus qui exécute le *shell*).
   — Comment faire attendre le processus *shell* après le lancement d'un processus de premier plan ? Une solution est de le mettre en attente, mais en testant périodiquement, par mesure de sécurité, l'état du processus de premier plan (boucle de test contenant un `sleep(1)`)[3].
2. Les processus d'arrière plan qui se terminent doivent être "ramassés" (c'est-à-dire doivent être traités par `waitpid` pour cesser d'être zombis). Cela doit se faire dans le traitant du signal `SIGCHLD`.
3. Il se pose un problème d'exclusion mutuelle : pendant que le *shell* modifie les structures représentant l'état des travaux, il est prudent de masquer certains signaux

3. Cette solution a le mérite de simplifier l'implémentation du *shell* car `waitpid` n'est appelée qu'à un seul endroit dans le code (au sein du traitant du signal `SIGCHLD`). Dans le cas contraire, il y a concurrence entre les appels à `waitpid` effectués par le *shell* et ceux effectués par le traitant du signal `SIGCHLD`.

(dont les traitants peuvent aussi manipuler ces structures), pour préserver la cohérence des données.

4. Un programme lancé par un *shell* fait l'hypothèse que son masque de signaux initial est vide et que ses traitants de signaux correspondent aux traitants par défaut. S'assurer que l'implémentation du mini-*shell* respecte cette règle.

5.3 Points à réaliser

1. *Changer l'état du processus en premier plan*
   Implémenter la gestion de `control-c` et `control-z` pour qu'ils envoient respectivement un signal SIGINT et un signal SIGTSTP au(x) processus de premier plan.
2. *Commande integrée `jobs`*
   Les commandes exécutées en arrière-plan s'appellent des jobs. Un job peut être désigné par le PID du processus qui l'exécute (exemple 14567) ou par son numéro de job précédé de % (exemple %3). Les numéros de jobs sont des entiers positifs, attribués à partir de 1.
   Implémenter la commande `jobs` qui donne la liste des commandes lancées.
3. *Agir sur les commandes en arrière plan*
   Implémenter les commandes `fg`, `bg` et `stop` qui agissent sur les jobs d'un shell et respectivement mettent un job en premier plan, en arrière plan ou l'arrêtent.
4. *Ajouter la possibilité d'utiliser le tilde, l'étoile et les variables d'environnements*
   Pour cette étape, vous devrez utiliser les primitives `wordexp` et `wordfree` qui vous permettront de remplacer, entre autres, le tilde, l'étoile et les variables d'environnements avant l'exécution des commandes.

6 Présentation des résultats

La présentation des résultats se fera de la manière suivante. Dans un premier temps, nous vous demanderons de téléverser les fichiers sources sur moodle. Puis, vous nous présenterez votre travail lors d'une soutenance de l'ordre de 15 minutes (Rien de trop formel, pas de transparents, vous nous montrez ce qui marche et nous discutons.).

6.1 Rendu des sources

Chaque binôme rendra le code source *commenté* des programmes réalisés et les fichier de test realisés. Nous vous demandons également de rédiger un bref compte-rendu présentant : (a) les principales réalisations et (b) une description des tests effectués.

Les différents fichiers seront rendus sous forme d'archive `tar.gz` ayant pour nom `NOM1-NOM2.tar.gz`. L'archive devra être structurée de la manière suivante :

— un dossier `src` contenant le code source
— un dossier `tests` contenant vos tests unitaires
— un makefile permettant de compiler votre programme
— votre rapport en PDF

6.2 Soutenance

Vous allez devoir faire une démonstration de votre shell. POur cela, nous vous demanderons de préparer une liste de commandes pour tester ses fonctionnalités. Les détails liés à cette soutenance vous seront communiqués en temps utile.