(mini)Projet (mini)Shell

1 Objectifs

Ce projet vise à vous permettre de mettre en pratique les notions de base vues en TD et TP, autour de la gestion des processus, des signaux et des E/S, dans un contexte suffisamment réaliste. Il s'agit plus précisément de développer un interpréteur de commandes simplifié, offrant les fonctionnalités de base des shells Unix, comme le bash...

Les questions successives devraient être traitées dans l'ordre, et au fur et à mesure que les notions nécessaires seront abordées en TP. Elles constituent autant d'étapes qui devraient vous permettre de partir d'une base simple, qui sera progressivement étoffée. La dernière section précise les prérequis pour chaque étape.

2 Boucle principale

La première étape va être d'écrire le comportement de base de l'interpréteur, qui consiste en une boucle infinie. Chaque itération de cette boucle lit une ligne sur l'entrée standard, l'interprète comme une commande, puis lance un processus fils qui exécutera cette commande.

Par exemple, lors de la session suivante :

```
sh-3.2$ pwd
/Volumes/DD/Users/Philippe/Enseignement/Cours/Système/Projet/17/exemple
sh-3.2$ ls
Unix.aux       minishell.log           minishell.tex       scripts.pdf
figures        minishell.pdf           scripts.aux         scripts.synctex.gz
minishell.aux  minishell.synctex.gz    scripts.log
sh-3.2$ ls figures
exec.pdf logo-n7.png
sh-3.2$ wc -l figures/exec.pdf
112 figures/exec.pdf
sh-3.2$
```

l'interpréteur lit, reconnaît puis lance l'exécution des commandes `pwd`, `ls`, `ls`, `wc`, puis reste en attente de lecture de la ligne suivante.

2.1 Lancer une commande élémentaire

Question 1 (Lancement d'une commande) Réaliser la boucle de base de l'interpréteur, en se limitant à des commandes simples (pas d'opérateurs de composition), sans motifs pour les noms de fichiers.

Le projet étant centré sur les fonctions et notions système, et non sur l'analyse lexicale,

- on ne considèrera pas par la suite la possibilité de définir des motifs pour les noms de fichiers ;
- une fonction de lecture et d'analyse d'une ligne saisie sur l'entrée standard est fournie (fichiers `readcmd.c` et `readcmd.h`). Il n'est pas nécessaire [1] pour la suite de modifier le code de `readcmd.c`.

2.2 Synchronisation entre le shell et ses fils

Si, comme cela est demandé pour la question 1 le processus shell lance un fils, puis se met immédiatement en attente de lecture de la prochaine ligne de commande, il est possible que l'affichage de l'invite précède ou se mêle à l'exécution du processus fils.

Question 2 (Exemple) Construire une session simple (utilisant le code écrit pour la question 1) mettant en évidence ce comportement.

1. et même carrément déconseillé, si l'on ne souhaite pas s'engager dans le développement d'un analyseur syntaxique...

Question 3 (Enchaînement séquentiel des commandes) Modifier votre code afin qu'il attende la fin de la dernière commande lancée avant de passer à la lecture de la ligne suivante.

Question 4 (Commandes internes) Compléter votre code en ajoutant deux commandes internes, exécutées directement par l'interpréteur sans lancer de processus fils : `cd` et `exit`.

Question 5 (Lancement de commandes en tâche de fond) Le comportement du code initial (celui écrit en réponse à la question 1) correspond cependant à une possibilité utile offerte par les shells, à savoir le lancement de commandes en tâche de fond, spécifié par un & en fin de ligne. Compléter votre code pour offrir cette possibilité.
Note : Le parseur fourni (fichiers `readcmd`) permet d'analyser et reconnaître un & en fin de ligne.

3 Gestion des processus lancés depuis le shell

Il s'agit ici de réaliser une version simplifiée des commandes du shell qui facilitent la gestion des processus lancés depuis l'interpréteur de commandes lui-même.

Question 6 (Gérer les processus lancés depuis le shell) Compléter votre code par les commandes internes suivantes :

- `jobs`, qui donne la liste des processus lancés depuis le shell et non encore terminés, avec leur identifiant propre au minishell, leur pid, leur état (actif/suspendu) et la ligne de commande lancée.
- `stop`, qui permet de suspendre un processus (l'identifiant à fournir à la commande `stop` sera l'identifiant géré par le minishell).
- `bg`, qui permet de reprendre en arrière-plan (en tâche de fond) un processus suspendu (l'identifiant à fournir à la commande `bg` sera l'identifiant géré par le minishell).
- `fg`, qui permet de poursuivre en avant-plan un processus suspendu ou en arrière-plan (l'identifiant à fournir à la commande `fg` sera l'identifiant géré par le minishell).

Remarque : comme pour le shell standard, un processus lancé depuis le minishell pourra être suspendu par

- la frappe de `ctrl-Z` au clavier s'il s'agit du processus en avant-plan du minishell ;
- la commande `stop` ;
- l'envoi du signal `SIGSTOP`.

Afin de faciliter la mise en œuvre, vous ***pourrez*** en revanche faire, à votre convenance (mais en justifiant votre choix), l'hypothèse que pour les processus lancés depuis le minishell, le signal `SIGTSTP` est ignoré, et/ou qu'ils ne démasquent pas ce signal, et/ou ne redéfinissent pas le traitant de ce signal.

Indication : la mise en œuvre des commandes `jobs, stop, fg, bg` pourra s'appuyer sur l'utilisation de la primitive `waitpid(_)`, et l'exploitation du signal `SIGCHLD`. Ce point technique est développé et détaillé dans une note disponible sur Moodle, dans la section consacrée au projet.

4 Terminaison du processus en avant-plan

Question 7 (`SIGINT`) la frappe de `ctrl-C` au clavier se traduit par l'envoi à votre shell du signal `SIGINT`. La réception de ce signal ne doit pas provoquer la terminaison de votre shell, ni celle de ses processus en arrière-plan, mais devra amener la terminaison du processus en avant-plan (éventuel) de votre shell. Compléter votre programme pour traiter cette frappe en conséquence.

Comme pour la question précédente, afin de faciliter la mise en œuvre, vous ***pourrez*** faire, à votre convenance (mais en justifiant votre choix), l'hypothèse que pour les processus lancés depuis le minishell, le signal `SIGINT` est ignoré, et/ou qu'ils ne démasquent pas ce signal, et/ou ne redéfinissent pas le traitant de ce signal.

5 Gestion des redirections

Question 8 (Redirections) Compléter votre programme pour permettre d'associer l'entrée standard ou la sortie standard d'une commande à un fichier.
Exemple (classique) : `cat < f1 > f2` associe `f1` à l'entrée standard de `cat`, et `f2` à la sortie standard de `cat`.

6 Tubes

Question 9 (Tubes simples) Compléter votre programme pour permettre de composer des commandes en les reliant par un tube.

Exemple : `ls | wc -l` lance les commandes `ls` et `wc`, la sortie standard de `ls` étant connectée à l'entrée standard de `wc` par un tube.

Question 10 (Pipelines) Etendre la fonctionnalité précédent en offrant la possibilité d'enchaîner une séquence de filtres liés par des tubes, de sorte à obtenir un traitement en pipeline.

Exemple : `cat toto.c lulu.c | grep int | wc -l`

7 Guide de progression et modalités pratiques

Les notions qui faut avoir vu (en TP) pour traiter les différentes questions sont les suivantes :

Questions 1,2,3,4 : gestion des processus ;

Questions 5,6,7 : signaux ; [*Note* : la question 7 demande un temps de réflexion et de documentation]

Question 8 : fichiers, redirections ;

Questions 9,10 : redirections, tubes ;

L'analyse d'une ligne de commande est réalisée par une fonction fournie (voir Moodle). L'archive contenant le code de cette fonction comporte un fichier `LisezMoi`, qui documente son usage.

En termes d'échéances, ce projet comporte deux étapes :

- pour la fin de la semaine du 12 avril (plus précisément, le **18 avril à 23h45** au plus tard), vous devrez rendre votre travail portant sur les questions 1 à 7. Les livrables pour ce rendu seront constitués du code et d'une courte note (1 page) commentant vos choix de conception, et éventuellement vos points de blocage. À ce stade, il n'est pas attendu que vous remettiez un travail abouti, mais plutôt que vous ayez traité pour l'essentiel les questions les plus simples (1 à 5) et que vous ayez compris les difficultés posées par les questions 6 et 7, en ayant envisagé des pistes de solutions.
- pour la semaine du 17 mai, (le **23 mai à 23h45** au plus tard), vous devez rendre votre travail final (questions 1 à 10). Ce travail pourra intégrer les retours ou corrections faites à la suite du retour sur le rendu intermédiaire. Il sera déposé sur Moodle.
  Le lien de dépôt précisera le format des livrables et fournira un script de pré-validation qui vérifiera la conformité au format demandé, et effectuera quelques tests de base qui fourniront une indication sommaire sur la viabilité minimale du code remis.
  En substance, les livrables pour ce rendu seront constitués du code et d'un bref rapport présentant l'architecture de l'application, les choix et spécificités de conception, la **méthodologie de tests** suivie, avec quelques test significatifs. Le rapport devra en particulier comporter la réponses à la question 2, ainsi qu'une explication sur les choix opérés pour la mise en œuvre des réponses aux questions 6 et 7. L'évaluation du code tiendra compte de sa qualité.

Attention : ce projet est un projet individuel. Il est permis, et même bénéfique, d'échanger sur la conception, mais le rendu final doit refléter un travail strictement personnel. Nous vous demandons de bien noter que nous disposons d'outils spécifiques pour détecter la copie de projets, et que **toute fraude avérée sera sanctionnée sans appel, quelles que soient les circonstances.**