Projet Système – Shell

Objectif

Le but de ce projet Système est de programmer un shell en C. Ce projet exploratoire à pour but de vous familiariser avec les couches basses du système. Afin de le réaliser vous allez devoir vous documenter. Pour cela, les principales sources d'informations que vous pouvez utiliser sont les pages de manuel en ligne de commande, la documentation de la GNU libc ainsi que les livres de programmation UNIX et POSIX.

Votre shell de base sera une simple boucle d'interaction qui affiche une invite de commande de la forme `user@machine:repertoire/courant>`, lit une ligne entrée au clavier, l'exécute, et recommence. Le shell quitte quand l'utilisateur tape `ctrl D` à la place d'une commande (cela peut être détecté en testant "if (caractère == EOF) alors ..."), ou bien lorsque 'exit' est tapé (cf. étape 3).

Note : (presque toute) l'étape 1 ainsi qu'une partie des redirections de l'étape 3 ont été faites au TP2.

Partie A : shell basique

Étape 1 : boucle d'interaction

On suppose pour commencer que la commande est simplement le nom d'un programme à exécuter dans un processus séparé. Le shell doit attendre que le programme se termine avant de rendre la main à l'utilisateur.

Fonction "system()" interdite : utilisez execXX(), **sans appeler un autre shell** :-)
Par exemple, l'implémentation de la fonction system() visible ici est interdite.

On suppose maintenant que la ligne de commande peut contenir un nombre arbitraire d'arguments, séparés par un ou plusieurs espaces. Elle peut aussi occuper plusieurs lignes, les lignes intermédiaires se terminant par un antislash '\'.

Astuce : pour ne pas stocker en dur la taille maximale d'une ligne de commande, utilisez la fonction `realloc()`.

Étape 2 : les commandes internes

Les commandes internes se comportent comme les autres mais ne donnent pas lieu à l'éxécution d'un programme dans un processus séparé. Elle sont réalisées directement par le shell soit parce qu'il est impossible de faire autrement (cas de la commande `cd`), soit pour des raisons de performances. Vous ajouterez dans un premier temps juste les commandes `cd` et `exit`.

Remarques : quelques indications pour 'cd' ; 'exit' ne doit pas être confondu avec la fonction C `exit()`, même si la commande fera sans doute appel à la fonction...

Etape 3 : Tubes et redirections

Cette dernière étape du tronc commun est celle qui va introduire les outils qui rendent les shells UNIX si puissants. Vous ajouterez à votre shell la possibilité d'utiliser les trois redirections standards : `<`, `>` et `2>` qui permettent de rediriger respectivement l'entrée et la sortie standard ainsi que la sortie d'erreur.

Ensuite vous ajouterez la possibilité de connecter deux commandes à l'aide d'un tube. On ne s'occupe ici que du cas de deux commandes et non pas d'une séquence arbitrairement longue de commandes connectées deux par deux.

Un petit tuto très clair au sujet des pipes.

Partie B : Finitions et commandes supplémentaires

Le shell réalisé dans la première partie comporte beaucoup de petits défauts qui le rendent inutilisable de manière quotidienne ; cette section vise à obtenir un shell plus abouti.

Etape 1 : Finitions

Vous allez commencer par modifier l'analyse des paramètres de manière à permettre l'utilisation de guillemets `"` afin de pouvoir spécifier des paramètre contenant des espaces. Vous ajouterez aussi une commande permettant de modifier les variables d'environnement telles que le `PATH`.

Etape 2 : Tubes avancés

Vous généraliserez votre code pour le lancement de deux commandes connectée par un tube de manière à ce qu'il puisse gérer un nombre arbitraire de commandes et qu'il soit possible d'utiliser une redirection d'entrée sur la première et une redirection de sortie sur la dernière.

Conseils : inspirez-vous de tout ce que vous pouvez trouver sur stackoverflow & cie, ainsi que dans le tuto de l'étape 4. Séparez la gestion des redirections via '>' et '<' des pipes dans un premier temps pour simplifier le problème. Enfin, écrivez le séquentiellement pour 3 ou 4 commandes avant d'utiliser une boucle.

Etape 3 : Commande internes

Vous reprendrez le code réalisé en TP1 pour ajouter une commande interne de copie supportant la copie récursive. Vous ajouterez des implémentations internes des commandes suivantes :

- `cat` sans options supplémentaires
- `ls` avec le support des options `-l` et `-a`
- `find` uniquement les options `-name` et `-exec`

Partie C : Job control

L'objectif cette partie est d'incorporer des mécanismes de Job control à notre shell. Le job control est la capacité qu'a un shell d'arrêter, de suspendre, de mettre en tâche de fond (background), de mettre en avant (foreground) ou même de continuer l'exécution d'une commande.

Un job est une commande ou une série de commandes séparées par des tubes "|"(exemple: `cat toto.txt | grep tata`). Un job est associé à une entrée de la ligne de commande ; c'est-à-dire que même si une entrée contient une série de commandes connectées, elle constituent un unique job.

Le job peut démarrer en arrière-plan si la commande se termine par '&'. La suspension d'un job est déclenchée par un appui sur `ctrl-Z`, le job peut ensuite être placé en avant ou arrière plan à l'aide des commandes `fg` et `bg`. Dans cette section, vous devrez implémenter ce mécanisme et ces deux commandes. Cette partie est très exploratoire et vous trouverez beaucoup d'informations complémentaires sur la manière de l'implémenter dans la section dédiée de la documentation de la GNU libc.

Note : cette page d'aide peut être utile aussi pour les parties A et B, notamment pour le choix des structures de données, mais aussi pour la gestion des signaux (en particulier `ctrl-C` ne devrait pas quitter le shell, mais juste le job en court d'exécution).

Informations

Le projet se fera par groupes de deux étudiants. L'évaluation du projet portera essentiellement sur le code rendu, mais aussi sur le compte-rendu et la soutenance.

Concernant le rapport, considérez le comme l'aide de votre shell : il doit

- expliquer comment compiler le programme − idéalement, fournissez un Makefile ;
- illustrer son utilisation, ses options par divers exemples ;
- lister les bugs connus (6 semaines c'est court, il y en aura sans doute) ;
- parler éventuellement des choses que vous avez essayées mais qui n'ont pas abouti, ou encore des améliorations que vous n'avez pas apportées faute de temps, etc.

Pour la soutenance prévoyez de parler de votre code pendant quelques minutes avec une petite démo, montrant les points forts/faibles, ce qui reste à faire et comment, ...etc.