TP

Mini-Shell

1 Informations importantes

- Le projet est à faire par groupe de deux personnes.
- **Les solutions ne pourront pas être partagées entre les groupes**.
- **Note** : Ce TP est noté.
- **Date limite** : Le TP est à soumettre pour le **7 février**.

2 Évaluation

Ce TP sera noté en fonction de la qualité du code produit et des fonctionnalités mises en œuvre.

2.1 Rendu

Votre soumission est à déposer sur Moodle en utilisant le lien prévu à cet effet :
- Soumettez une archive nommée à partir des noms de famille des 2 membres du binôme :
  — `Nom1_Nom2_tpshell.tar.gz`
- Soumettez une seule archive par binôme

Votre soumission devra contenir :
- Un court rapport au format `txt`, `md` (Markdown), ou `pdf`, qui comprendra :
  — Les noms des 2 membres du binôme
  — La liste des fonctionnalités que vous avez mises en oeuvre
  — La liste des bugs connus
- Votre code source
  — Qui devra pouvoir être compilé sur les machines de l'UFR

3 Introduction

Un shell est un interpréteur de commandes permettant aux utilisateurs d'accéder aux fonctionnalités offertes par le système d'exploitation. C'est ce que vous avez dans vos terminaux.

Un shell est en fait un programme comme un autre, qui attend des commandes sur son entrée standard et qui les exécute. Il est ainsi tout à fait possible d'écrire son propre shell. C'est ce que nous allons faire pendant ce TP.

Les shells installés sur les systèmes à votre disposition sont très complexes. Ils peuvent gérer des redirections complexes, manipuler des jobs (avant-plan/arrière-plan), gérer des variables, permettre des manipulations de texte (double quote, backquote, simple quote, ...), etc.

Notre but dans ce TP est de créer un shell simple, qui supportera un sous-ensemble des fonctionnalités des shells classiques.

4 Travail à réaliser

4.1 Programme principal

Compiler, tester et comprendre le programme fourni `shell.c`. Il permet de lancer une commande introduite par l'utilisateur : vous pouvez par exemple taper "ls" ou encore "ls -l".

Note: Utilitaires
Des fonctions pour vous aider sont disponibles dans le fichier `shell-utils.c` (avec les entêtes dans le fichier `shell-utils.h`). Vous pouvez utiliser ces fonctions, mais il ne faut pas modifier ces deux fichiers.

4.2 Les premières étapes

1. Faire afficher un prompt (cad, une invite de commande, par exemple "commande : ") pour que l'utilisateur voit quand il peut taper une commande.
2. Modifier ce programme pour qu'il lance des commandes tant que l'utilisateur n'aura pas tapé la commande `exit` ou `ctrl+D` pour signifier *End-Of-File* (EOF).
3. Faire en sorte que ce shell attende la fin de la commande précédente avant de demander la commande suivante.

4.3 Fonctionnalités avancées

1. *Commande 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`.
2. *Gestion des erreurs*
   Votre implémentation doit 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 inexistante (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`.
3. *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`.
4. *Séquence de commandes composée de plusieurs commandes et plusieurs tubes*
   Penser à tester des cas complexes pour vérifier que tout fonctionne correctement. Par exemple :
   `cat file1.txt | grep "toto" | wc -l > /tmp/count.txt`

4.4 Fonctionnalités bonus

Commentaire : Pour certaines des tâches ci-dessous, il sera peut être nécessaire de mettre en oeuvre des fonctions utilitaires en plus pour analyser les commandes entrées par l'utilisateur. Nous vous recommandons cependant, de ne pas modifier les fonctions fournies (fichiers `shell-utils.*`)

Si il vous reste du temps, vous pouvez choisir parmi les propositions suivantes des fonctionnalités à mettre en oeuvre :

- *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.
- *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.
  L'option `WNOHANG` de `waitpid()` pourra être utile ici (ainsi que le fait que avec $pid = -1$, `waitpid()` permet d'attendre n'importe quel processus). On remarquera aussi que quand un processus fils se termine, le père reçoit le signal `SIGCHLD`.
- *Gestion d'autres redirections* (">>", "<<", ">&")
- *Commande intégrée `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 exécutées en arrière-plan.