Unix : concepts et outils avancés
TD no 7 : mini-shell

Olivier Aumage
LIP, bureau 302
email : oaumage@ens-lyon.fr

15 novembre 2000

1 Introduction

1.1 Mini-shell

L'objectif des trois TD à venir est la réalisation d'un *mini-shell* dont l'évaluation sera prise en compte dans la note *contrôle continu* du cours « Unix, concepts et outils avancés ». Le mini-shell est à réaliser en binôme (avec exceptionnellement un trinôme pour cause de nombre impair d'étudiants) et devra être rendu pour le jeudi 6 décembre, à la fin du TD (10H00).

1.2 Consignes

L'accent devra être mis sur la <u>qualité du code source</u> et notamment sur les points suivants :

Modularité le code devra être réparti sur plusieurs fichiers sources (la compilation se fera par l'intermédiaire d'un Makefile), chaque module présentant une interface claire et précise au reste du logiciel. Note : ce point est vraiment essentiel pour garantir l'évolutivité de votre *shell* au cours des prochains TD.

Clarté le code devra être rigoureusement présenté : indentation, noms de variables et de fonctions explicites, un minimum de documentation.

Simplicité le code devra être constitué de fonctions courtes et réalisant des tâches simples.

Robustesse le logiciel devra prendre en compte, dans la mesure du possible, la qualité d'être humain de l'utilisateur.

Le respect de ces quelques règles devrait vous assurer une notation à la hauteur de vos efforts, tout en limitant la durée de développement et de mise au point du logiciel. À ce propos, n'oubliez pas les enseignements du TD sur GDB et *Electric Fence* !

2 Manipulations préalables

Les quelques exercices ci-dessous ont pour but de vous familiariser avec la génération de processus et l'exécution de programmes avant de débuter la réalisation du shell proprement-dit.

– Exécutez le programme ci-dessous et observez les deux processus générés à l'aide de la commande `top`. Quel est l'état du processus fils ?

```
#include <unistd.h>

int
```

```
main()
{
  if (fork())
    {
      sleep(10);
    }
}
```

- Modifiez le programme ci-dessus pour écourter la phase *zombie* du processus fils en utilisant `wait` ou `waitpid`.
- Les fonctions de la famille `exec` permettent de remplacer le programme exécuté par un processus par un autre programme exécutable. Ecrivez un programme `my_exec` qui utilise `execvp` pour exécuter une commande spécifiée en argument 1 avec comme arguments les paramètres 2 à `n` du programme.
- Modifiez `my_exec` pour afficher le code de retour de la commande exécutée si celui-ci est non nul.
- Modifiez `my_exec` pour que l'utilisateur puisse spécifier, optionnellement, un fichier dans lequel la sortie standard de la commande sera redirigée. Exemple :
  ```
  > ./my_exec -o sortie ls /usr/local/bin
  ```
  Dans l'exemple donné, `my_exec` exécute `ls` avec `/usr/local/bin` comme argument et redirige sa sortie standard vers le fichier `sortie`.

3 Le shell - première étape

3.1 Objectif

L'objectif de cette première étape est de concevoir et écrire un interpréteur de commandes présentant les fonctionnalités élémentaires des shells UNIX tels que `sh` ou `csh` par exemple. Voici les spécifications minimales pour cette première étape :

- l'usage de variables globales non statiques est interdit ;
- l'interprète lit ligne par ligne son entrée standard ;
- l'interprète exécute une ligne à la fois ;
- la fermeture de l'entrée standard entraîne la terminaison du shell ;
- la longueur d'une ligne peut être bornée, la borne ne doit pas être inférieure à 255 caractères ;
- l'interprète affiche un message d'erreur si la ligne lue est trop longue, la ligne est ignorée dans ce cas ;
- le séparateur de mots est l'espace (`' '`) ;
- le nombre de commandes par ligne peut être borné, la borne ne doit pas être inférieure à 5 ;
- les caractères séparateurs de commandes sont le caractère de fin de ligne, le caractère `;` et le caractère `&`. Ainsi, la ligne
  ```
  > com1 arg1 ; com2 arg2 & com3 arg3
  ```
  est équivalente aux lignes
  ```
  > com1 arg1
  > com2 arg2 &
  > com3 arg3
  ```
- le nombre d'arguments par commande peut être borné, la borne ne doit pas être inférieure à 10 ;
- les commandes sont exécutées en mode synchrone par défaut : le shell attend la fin de la commande en cours avant d'exécuter la suivante ;
- le caractère `&` en fin de commande indique que la commande doit être exécutée en mode asynchrone (c'est-à-dire en tâche de fond) ;
- chaque tâche lancée en mode asynchrone possède un identifiant ;
- l'identifiant d'une tâche asynchrone vaut 1 si aucune tâche asynchrone n'est active, ou `n + 1` si `n` est le plus grand numéro d'une tâche asynchrone active ;
- le nombre d'identifiants de tâches asynchrones peut être borné, la borne ne doit pas être inférieure

à 10 ;
– les entrées et sorties standards des commandes pourront être redirigées sur des fichiers ;
– le fichier recevant la sortie standard doit être créé s'il n'existe pas ;
– les redirections peuvent se trouver entre les arguments de la commande correspondante, les trois commandes suivantes sont équivalentes :

```
> com arg1 arg2 < file
> com arg1 < file arg2
> com < file arg1 arg2
```

– le shell devra disposer des commandes internes ci-dessous :
  – `jobs` affiche la liste des commandes s'exécutant en tâche de fond ;
  – `wait` prend un identifiant de commande asynchrone en paramètre et attend sa terminaison.
– le shell devra veiller à ne pas laisser des processus zombies s'accumuler.

3.2 Indications

D'après l'analyse des spécifications ci-dessus, une organisation en au moins 4 modules est probablement une bonne option (mais pas une obligation) :
– un module principal contenant la fonction `main` et la boucle principale du shell ;
– un module analyseur chargé de la lecture des lignes de commandes sur l'entrée standard ;
– un module d'exécution chargé de la génération et du contrôle des processus fils ;
– un module implémentant les fonctions internes.

3.3 Structures de données

De la qualité de vos structures de données dépendra la facilité de réalisation du shell. Un soin tout particulier doit donc être apporté à leur conception, notamment en évitant les structures fourre-tout ainsi que la duplication des informations.

— ☆ —