Université de Grenoble alpes

Implémentation d'un mini Shell en langage C

Compte-rendu

Système et réseau

Mekaoussi Mohamed Fares
Djaghloul Ayoub Ahmed Amine

Introduction :

L'objectif de ce projet est d'implémenter un programme qui offre des fonctionnalités similaires à celles d'un Shell Bash. Il a été nécessaire de mettre en pratique différentes notions étudiées en cours, notamment, la gestion des processus, les redirections et les pipes, en utilisant des appels système. Le but est de créer un programme fonctionnel qui permettra à l'utilisateur d'interagir avec l'ordinateur en entrant des commandes textuelles.

Organisation du code

Le dossier "src" contient plusieurs fichiers qui sont utilisés pour implémenter différentes fonctionnalités du programme. Le fichier "**csapp.c**" et "**csapp.h**" sont des fichiers qui contiennent des fonctions qui permettent d'interagir avec les appels système.
Les fichiers "**function.h**" et "**function.c**" contiennent les fonctions qui ont été implémentées pour les fonctionnalités du programme telles que l'exécution de commandes, la redirection des entrées/sorties et la gestion des erreurs.
Le fichier "**readcmd.c**" et "**readcmd.h**" sont utilisés pour la lecture et l'analyse des commandes saisies par l'utilisateur.
Enfin, le fichier "**shell.c**" est le programme principal qui utilise toutes ces fonctions.

Les Fonctions implémentées :

Commande pour terminer le shell :

Pour implémenter la commande **quit**, nous avons ajouté une fonction de test et l'appeler dans la boucle principale du Shell pour détecter si la commande entrée est "**quit**" ou "**exit**". Si c'est le cas, nous avons utilisé la fonction exit pour quitter le Shell proprement.

```c
//exit the shell
void quit(char **cmd){
    if ((strcmp(cmd[0], "quit")&&strcmp(cmd[0], "exit")) == 0)
                exit(0);
}
```

Interprétation de commande simple :

Pour implémenter l'interprétation de commande simple, nous avons utilisé la fonction **execvp** pour exécuter la commande entrée par l'utilisateur. Nous avons également ajouté un traitement d'erreur pour afficher un message d'erreur si la commande n'a pas pu être exécutée.

```
void execute(char **cmd){
        //check if the command is in the path
        if (execvp(cmd[0], cmd) == -1) {
            printf("%s : Command not found\n",cmd[0]);
            exit(EXIT_FAILURE); //exit failure
        }
}
}
```

Redirection d'entrée ou de sortie :

Pour implémenter la redirection d'entrée ou de sortie, nous avons utilisé les primitives **Open**, **Dup** et **Dup2,** dans une fonction "redirect", qui permet de rediriger l'entrée et la sortie standard d'un programme vers des fichiers. Elle vérifie également les permissions d'accès aux fichiers avant de les ouvrir et de les utiliser.

```
// redirect the input and output
void redirect(char *in, char *out){
    if(in){
        //check the permission of the file
        if (access(in, R_OK) == -1) {
            printf("Permission denied");
            exit(0);
        }
        int fd0 = open(in, O_RDONLY, 0);
        dup2(fd0, 0);
        close(fd0);
    }
    if(out){
        //check the permission of the file
        if (access(out, W_OK) == -1) {
            printf("Permission denied");
            exit(0);
        }
        int fd1 = open(out, O_WRONLY, 0);
        dup2(fd1, 1);
        close(fd1);
    }
}
```

Séquence de commandes :

Lorsque nous exécutons plusieurs commandes, nous utilisons la primitive Pipe avec un tableau de deux cases qui est un descripteur de fichiers. Cependant, nous avons rencontré une difficulté lorsqu'il s'agissait de diriger la sortie de la commande suivante. Pour résoudre ce problème, nous avons basé notre code sur la vérification s'il y avait des commandes suivantes ou non, ce qui nous a permis de diriger les entrées et sorties de chaque commande de manière appropriée.
La fonction fork nous permis de créer des processus fils pour chaque commande. Ces processus fils ont été utilisés pour exécuter chaque commande appart la dernière qui sera exécuter par le père.

Exécution de commandes en arrière-plan

On a ajouté une variable de type entier dans la structure de cmdline qui porte la valeur 1 si le & existe dans la commande, si ce dernier est vrai on appelle la fonction **command_bg** qui a le même principe d'exécution d'une commande normale mais le père utilise le paramètre : **WNOHANG**

```c
void command_bg(struct cmdline *l, char **cmd) {
    pid_t pid;
    int status;
    if ((pid = fork()) == 0) {
        redirect(l->in, l->out);
        execute(cmd);
        exit(0);
    } else {
        waitpid(pid, &status, WNOHANG);
        printf("Background job started with pid %d\n", pid);
    }
}
```

Gestion des zombis

Notre fonction de traiteur est appelée suite à la réception d'un signal, elle exécute une boucle qui récupère l'état de tous les processus fils qui ont été terminés ou arrêtés en utilisant la fonction waitpid(). Elle vérifie ensuite l'état de chaque processus fils récupéré en utilisant les macros WIFEXITED, WIFSIGNALED et WIFSTOPPED, qui permettent de déterminer si le processus fils a terminé normalement, a été tué par un signal ou a été arrêté par un signal respectivement. En fonction de l'état du processus fils.

```c
void handler(int sig){
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        if (WIFEXITED(status)) {
            printf("Background job with pid %d exited normally\n", pid);
        } else if (WIFSIGNALED(status)) {
            printf("Background job with pid %d was killed by signal %d\n", pid, WTERMSIG(status));
        } else if (WIFSTOPPED(status)) {
            printf("Background job with pid %d was stopped by signal %d\n", pid, WSTOPSIG(status));
        }
    }
}
```

Description des tests effectués :

- Tester la redirection d'entrée : ls > tests/output.txt
- Tester la redirection de sortie : cat < tests/input.txt
- Tester la gestion des erreurs de commande : thisisnotacommand
- Tester la commande pwd : pwd
- Tester la commande echo avec arguments : echo hello world
- Tester la commande en arrière-plan : xeyes & ls
- Tester les commandes en pipeline : ls | grep txt

Conclusion :

Dans l'ensemble, nous avons réussi à implémenter la plupart des fonctionnalités d'un shell basique. Nous avons eu des difficultés lors de la gestion de plusieurs pipes et dans le traitement des signaux, mais nous avons pu les résoudre en comprenant mieux les appels systèmes.
Nous n'avons cependant pas réussi à implémenter la commande cd pour changer de répertoire, mais cela n'affecte pas la fonctionnalité générale du shell.

En résumé, malgré quelques difficultés, notre shell est fonctionnel et nous avons appris beaucoup sur la gestion des processus et des appels systèmes.