Rapport de Projet : Minishell

ELLAIK Fadwa

1 Architecture de l'Application

Le projet **Minishell** est un interpréteur de commandes simplifié développé en C. Son architecture repose sur une boucle principale qui lit, analyse et exécute les commandes. Il gère les processus, signaux, E/S, et tubes.

Fonctionnalités clés

- **Exécution de commandes** : via `fork()` et `execvp()`.
- **Gestion des processus** : support de l'avant-plan et arrière-plan.
- **Gestion des signaux** : `SIGCHLD`, `SIGINT`, `SIGTSTP`.
- **Redirections d'E/S** : vers des fichiers.
- **Commandes internes** : `cd`, `dir`.
- **Tubes (Pipes)** : support des commandes chaînées.

Structure du code (`minishell.c`)

- `main` : boucle principale.
- `traitement_signal` : handler de signaux.
- `changer_rep`, `afficher_rep` : implémentations des commandes `cd`, `dir`.
- `executer_commande` : exécution avec redirections et tubes.

2 Choix et Spécificités de Conception

Le développement du minishell a été progressif. Voici un résumé par étapes :

Étapes 1 à 4 : TP1 - Gestion des Processus

Étape 2 - Lancement d'une commande :

```
pid_t pid = fork();
if (pid == 0) {
    execvp(cmd[0], cmd);
    exit(EXIT_FAILURE);
}
```

Étape 3 - Enchaînement séquentiel : utilisation de `wait()`.
Étape 4 - Tâches de fond : commande avec `&` non bloquante.
Prompt retardé ? Si le père attend tous les fils, le prompt est bloqué.
Solution : ne pas attendre les fils en arrière-plan, utiliser `pause()`.

Étapes 5 à 9 : TP2 - Signaux

Handler `SIGCHLD` :

```
void traitement_signal(int signal) {
    int status;
    pid_t pid;
    if (signal == SIGCHLD) {
        while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED |
   WCONTINUED)) > 0) {
            printf("Le processus fils %d a change d' tat : ", pid);
            if (WIFEXITED(status)) {
                printf("il s'est termine avec le code %i\n", WEXITSTATUS(
   status));
            } else if (WIFSIGNALED(status)) {
                printf("il  s est  termine (killed) par le signal %i\n",
   WTERMSIG(status));
            } else if (WIFSTOPPED(status)) {
                printf("il s'est suspendu par le signal\n");
            } else if (WIFCONTINUED(status)) {
                printf("il continue\n");
}   }   }   }
```

Utilisation de `pause()` pour attendre un signal.

TP3 : Contrôle clavier

Étape 11.1 – Traitement personnalisé des signaux clavier :
Permettre la réception des signaux `SIGINT` (Ctrl+C) et `SIGTSTP` (Ctrl+Z) sans interruption du minishell

```
// Dans traitement_signal
else if (signal == SIGTSTP) {
printf("\nsignal SIGTSTP re u\n> ");
} else if (signal == SIGINT) {
printf("\nsignal SIGINT re u\n> ");
}
```

```
// Dans main
if (sigaction(SIGINT, &action_sig, NULL) == -1) {
perror("erreur de sigaction pour SIGINT");
exit(EXIT_FAILURE);
}
if (sigaction(SIGTSTP, &action_sig, NULL) == -1) {
perror("erreur de sigaction pour SIGTSTP");
exit(EXIT_FAILURE);
}
```

Étape 11.2 – Héritage et réinitialisation des signaux dans les processus fils :

L'ignorance des signaux n'est pas transmise au processus fils, car les signaux sont explicitement rétablis à leur comportement par défaut dans le fils.

```
// Avant le fork dans le parent
signal(SIGINT, SIG_IGN);
signal(SIGTSTP, SIG_IGN);

// Dans le fils, apres fork
signal(SIGINT, SIG_DFL);
signal(SIGTSTP, SIG_DFL);
```

Étape 11.3 – Masquage temporaire des signaux :

Pour éviter les interruptions inappropriées durant certaines opérations critiques, les signaux `SIGINT` et `SIGTSTP` sont temporairement bloqués via `sigprocmask` :

```
sigset_t masque;
sigemptyset(&masque);
sigaddset(&masque, SIGINT);
sigaddset(&masque, SIGTSTP);
sigprocmask(SIG_BLOCK, &masque, NULL);
```

Étape 12 - Processus détaché :

```
if (commande->backgrounded != NULL) {
    setpgrp();
}
```

TP4 : Fichiers et Redirections

Etape 13 :

On gère les redirections d'entrée (<) et de sortie (>) vers des fichiers en utilisant open() et dup2().

```
if (commande->in != NULL) {
    int entree = open(commande->in, O_RDONLY);
    dup2(entree, STDIN_FILENO);
    close(entree);
}
if (commande->out != NULL) {
    int sortie = open(commande->out, O_WRONLY|O_CREAT|O_TRUNC, 0644);
    dup2(sortie, STDOUT_FILENO);
    close(sortie);
}
```

Étape 14 - Commande cd :

Implémentation de la commande interne cd en utilisant chdir(). La variable d'environnement HOME est utilisée pour le répertoire racine.

```
void changer_rep(const char *nouv_rep){
    if (nouv_rep == NULL){
        chdir(getenv("HOME"));
    } else {
        chdir(nouv_rep);
    }
}
```

Étape 15 - Commande dir :

Implémentation de la commande interne dir pour lister le contenu d'un répertoire en utilisant opendir(), readdir() et closedir().

```
void afficher_rep(const char *rep){
    DIR *repertoire = opendir(rep);
    struct dirent *entree;
    while ((entree = readdir(repertoire)) != NULL){
        printf("%s\n", entree->d_name);
    }
    closedir(repertoire);
}
```

TP5 : Tubes

Création et fermeture des tubes :

```
int tubes[nb_commandes - 1][2];
for (int i = 0; i < nb_commandes - 1; i++) {
    pipe(tubes[i]);
}
...
close(tubes[i][0]);
close(tubes[i][1]);
```

Le minishell supporte désormais les commandes reliées par des tubes (|). Pour n commandes dans un pipeline, n-1 tubes sont créés. La fonction executer_commande a été généralisée pour prendre en charge les descripteurs de tube pour l'entrée et la sortie.

3 Méthodologie de Tests

Les tests ont été itératifs, chaque fonctionnalité testée après implémentation.

Exemples de tests

- `sleep 5 &` puis `sleep 10` : test de la tâche de fond.
- `ps -fjn` : vérification de l'absence de zombies.
- `sleep 50` + `Ctrl-C/Z` : contrôle du signal.
- `ls > out.txt`, `cat < in.txt > out2.txt`.
- `cd /tmp`, `pwd`, `dir /usr/bin`.
- `ls | wc -l`, `cat minishell.c | grep include | wc -l`.

Autres test et signification des résultats

TP1 :

"Z : zombie, le processus est terminé mais que le père n'a pas pris compte de notre terminaison".

Ceci était un problème initial résolu par l'implémentation de SIGCHLD.

TP2 :

"fils : variable = 10, fin du fils, variable = 10" et "pere : variable = 100, fin du père, variable = 100".

Ce test a confirmé que les processus fils et père ont des copies indépendantes des données après un fork.

TP3 :

Lancement des signaux Ctrl+C et Ctrl+Z.

"signal SIGINT reçu" ou "signal SIGTSTP reçu" affiché par le minishell, confirmant le traitement du signal.

TP4 :

dir et dir /usr/bin : Vérification de l'affichage correct du contenu des répertoires.