Projet de programmation système

Mini-Shell

DOMMARTIN Jules

GUEZE Lucas

Sommaire

Table des matières

I. Choix des fonctionnalités

Les commandes internes

Le choix des fonctionnalités de notre Mini-Shell s'est fait en plusieurs étapes. En effet, nous voulions tout d'abord réaliser un Mini-Shell avec des fonctionnalités « basiques », comme l'exécution de commandes dites « internes » (echo, cd), mais qui fonctionne correctement avec un traitement de toutes les erreurs.

Les commandes externes

Après avoir réalisé cette première étape, nous nous sommes décidés à travailler sur les commandes dites « externes », dans la même dynamique de réalisation, avec le traitement des erreurs. Par exemple, entrer une ligne de commande vide ne fait rien (ce n'est pas le cas si l'on ne traite pas cette entrée, le programme renvoie une erreur). L'exécution de commandes externes se fait en créant un processus fils qui va exécuter la commande. Voir partie II.

Les tubes

Nous avons voulu pouvoir combiner plusieurs commandes grâce au caractère « | », que l'on nommera « pipe ». La réalisation de tubes s'est faite en plusieurs étapes. Nous avons dû modifier les fonctions déjà existantes et créer deux nouveaux types structurés (que l'on a appelé job_t et ensemble_job_t) pour la réalisation de cette fonctionnalité. Elle est fonctionnelle et permet d'enchaîner jusqu'à trois commandes. Nous avons décidé arbitrairement de limiter le nombre de commandes en pipe à trois, mais nous pouvons le changer très facilement puisqu'il s'agit en réalité d'une variable interne au programme.

L'exécution en arrière-plan

Une commande peut-être lancée en arrière-plan grâce au caractère « & » placée à la fin de la commande. Ainsi, le Mini-Shell n'attend pas la mort du processus fils créé pour continuer son exécution.

Les signaux

En dernière fonctionnalité, nous avons ajouté le traitement des signaux. Ainsi, Ctrl+D arrête le Mini-Shell, Ctrl+C ne fait rien dans le cas où aucun processus en premier plan n'est lancé (alors qu'il devrait arrêter le Mini-Shell s'il n'était pas traité), sinon il arrête le processus en premier-plan en cours. NB : Deux processus lancés en pipe s'arrêtent tous les deux avec un Ctrl+C.

II. Organisation du répertoire et des fonctions

Le répertoire contient plusieurs fichiers « .c » et « .h ». L'ensemble des fonctions définies dans ces fichiers suivent une logique précise de programmation.

Le fichier mon_shell.c

La fonction main (le point d'entrée du programme) se trouve dans le fichier « mon_shell.c ». Elle contient plusieurs variables et fonctions d'initialisation du programme :

- Une variable « sigact » qui est une structure de contrôle représentant les différents signaux.
- Une variable « ligne » qui est une structure de contrôle représentant une ligne de commande.
- Une fonction d'initialisation des jobs qui initialise une structure de contrôle pour un pointeur ensemble_job_t passé en paramètre.
- Une fonction d'initialisation des signaux, qui suspend le signal « SIGINT » (Ctrl+C) dans le programme.

Cette fonction lance ensuite une boucle infinie sur trois fonctions :

1. affiche_invite() qui affiche une ligne contenant le répertoire courant suivi du caractère « > ». NB : Cette fonction utilise la fonction get_current_dir_name() qui a besoin de « `#define _GNU_SOURCE` » pour fonctionner.
2. lit_ligne() qui reçoit la ligne de commande entrée et la traite pour la stocker dans la variable ligne. Elle est définie dans le fichier « ligne.h ».
3. execute_ligne() qui exécute la ou les commandes contenues dans la variable ligne.

Le programme ne s'arrête pas tant qu'il n'a pas reçu son signal d'arrêt.

Le fichier ligne.c

Ce fichier contient les fonctions utiles à la lecture de la ligne de commande entrée. On a vu précédemment que la fonction main appelait une fonction lit_ligne(). Cette fonction est implémentée dans ce fichier, et, à l'aide de la fonction fgets(), va récupérer la ligne entrée.

Il contient également les fonctions utiles à la découpe et à l'extraction d'une commande. En effet, une ligne a besoin d'être découpée pour analyser chaque paramètre de chaque commande, en récupérant tous les mots entre chaque espace.

La structure ligne_analysee_t

Cette structure nous permet de stocker une ligne de commande. Les variables NB_MAX_COMMANDES, NB_MAX_MOTS et NB_MAX_CAR sont des variables prédéfinies dans le fichier.

Voici la structure :

```
typedef struct ligne_analysee_t
{
  int    nb_fils;
  char*  commandes[NB_MAX_COMMANDES][NB_MAX_MOTS];
  char   ligne[NB_MAX_CAR];
} ligne_analysee_t;
```

Les structures job_t et ensemble_job_t

Ces structures nous permettent de stocker les différentes lignes de commandes à exécuter. Un ensemble de job est utilisé pour les commandes exécutées avec un tube.

Voici ces structures :

```
typedef struct job_t
{
  char   nom[NB_MAX_CAR];
  pid_t  pids[NB_MAX_COMMANDES+1];
  int tubes[NB_MAX_COMMANDES-1][2];
} job_t;

typedef struct ensemble_job_t
{
  job_t  jobs[NB_MAX_JOBS];
} ensemble_job_t;
```

Exécution des commandes internes et externes

L'exécution des commandes internes et externes se fait dans les fichiers « internes.c » et « externes.c » respectivement. Le programme essaie dans l'ordre tout d'abord d'exécuter une ligne de commande interne, et si la ligne n'est pas reconnue en tant que commande interne alors elle est exécutée en tant que commande externe.

Si la ligne est exécutée en commande interne, alors le programme n'a pas besoin de créer de processus fils et exécute directement la commande à l'intérieur du processus.

Si la ligne est exécutée en tant que commande externe, alors la création d'un fils est nécessaire pour son exécution. Aussi, une commande externe lancée en background (avec le caractère « & ») ne peut pas être tuée avec un Ctrl+C. A l'inverse, si une commande externe est lancée en avant-plan, le Mini-Shell attendra la mort du processus fils créé pour l'exécution de cette commande avant de reprendre.

On peut également noter que l'exécution d'une commande externe entraîne une redirection des signaux vers le processus fils créé. Ainsi, un programme lancé en avant-plan à partir du Mini-Shell pourra être stoppé par un Ctrl+C.

Les différents fichiers

Pour résumer, le répertoire contient pour chaque fichier « .c » un fichier « .h » contenant les déclarations des fonctions et de types. Les fichiers sont listés ici :

- mon_shell.c
- ligne.c
- job.c
- internes.c
- externes.c
- Makefile

Le Makefile sert à automatiser la compilation de nos fichiers.

La commande « make menage » écrase tous les points .o existants et l'exécutable « mon_shell » créé.

La commande « make » compile tous les fichiers et crée un exécutable « mon_shell » dans le même répertoire.

III. Problèmes rencontrés

Même s'ils n'ont pas été nombreux, nous avons rencontré certains problèmes tout au long de l'implémentation de notre Mini-Shell. Certains étant algorithmiques, d'autres par manque de connaissance du langage C.

Nous n'allons pas lister ici les problèmes que nous avons jugés « facile à résoudre ».

Au début, lorsque nous avions simplement implémenté l'exécution des commandes internes et externes, le Mini-Shell n'était pas en mesure de réagir face à une commande vide. C'est-à-dire lorsque nous appuyions simplement sur la touche « Entrée ». Le problème venait du fait que lorsque nous analysions syntaxiquement une ligne de commande, nous vérifions que la ligne ne commençait pas par un espace, or le caractère contenu dans une ligne vide est un caractère vide, donc un tableau d'un caractère vide en C. Notre programme faisait en sorte de passer à la case suivante de ce tableau pour récupérer le mot suivant. Or cette case de tableau n'existait pas, créant donc une « Segmentation fault » dans notre programme. Nous avons résolu le problème en traitant ce cas d'erreur possible et en retournant un entier qui définissait le comportement du programme par la suite.

Autre problème rencontré, la redirection. Nous n'avons pas réussi à mettre en place la redirection, car nous avions mal commencé à l'implémenter algorithmiquement parlant, et ensuite par manque de temps. En effet, lors de l'analyse d'une ligne, si celle-ci contenait un caractère « > » il fallait prendre le premier puis le deuxième argument ensuite pour exécuter la redirection. Mais dans le cas contraire (avec un caractère « < »), c'est dans le sens inverse que la commande devait s'effectuer. Nous n'avons pas pris le temps de réfléchir correctement à une solution adaptée.

Nous avons voulu implémenter la commande interne « kill », mais nous n'avons pas réussi à la mettre en place. Elle est toutefois encore dans le fichier « internes.c » et commentée.

La mise en place de l'enchaînement de plusieurs commandes à l'aide du caractère « ; » n'a pas été possible également, par un cruel manque de temps. Nous nous sommes concentrés sur les fonctionnalités énoncées dans la partie I.

IV. Le rendu

Pour résumer, notre Mini_Shell est fonctionnel, et tout le code présent dans les fichiers est commenté afin de le rendre plus accessible et de nous y repérer également.

Les fonctionnalités développées sont les suivantes :

- Exécution de commandes internes
- Exécution de commandes externes
- Les tubes
- L'exécution en arrière-plan d'un programme
- Le traitement des signaux

Certaines fonctionnalités comme la redirection n'ont pas pu être réalisée. D'autre non pas réussi à voir le jour à cause d'une mauvaise implémentation, comme la commande kill.

Cependant, toutes les fonctionnalités listées ci-dessus sont opérationnelles et permettent déjà à notre Mini-Shell de remplir des tâches fondamentales.

Annexe

Fonctionnalités traitées :

Fonctionnalité	Réalisée (oui/non)	Nom de fonction, nom de fichier
Commandes internes	Oui	internes.c
Redirection d'E/S	Non	
Tubes	Oui	externes.c
Enchaînement de commandes	Non	
Mise en background	Oui	externes.c, mon_shell.c
Interception de signaux	Oui	mon_shell.c, externes.c
Autres :		

Compilation du projet :

- Par cible Make : « make » pour compiler le projet
- Par cible gcc : "`gcc -g -Wall -W -Werror -Os -std=gnu11`"

Exécution du projet:

Commande : « ./mon_shell »