Projet C : First Shell (FiSH)
Système et programmation système

Eric Merlet

Université de Franche-Comté – UFR Sciences et Technique
Licence Informatique – 2è année
2023 – 2024

Le but de ce projet est de coder un interpréteur de commande (*shell*) et comme ce sera votre premier interpréteur de commande, il s'appellera *First Shell*, ou en abrégé `fish`.

Spécification

Le shell que vous allez réaliser est une simple boucle interactive qui demande une ligne de commande (éventuellement constituée de plusieurs commandes simples communiquant avec des tubes, et/ou de redirections), puis l'exécute et recommence. La réalisation de ce shell sera progressive, en augmentant progressivement la difficulté des types de lignes de commande à traiter.

Pour vous aider, vous avez à votre disposition une archive contenant une unité de compilation (fichiers `cmdline.h` et `cmdline.c`) qui définit des fonctions de lecture et d'analyse d'une ligne de commande, ainsi qu'un programme de démonstration (`fish.c`) qui vous servira de code de départ. Le fichier d'entête `cmdline.h` définit une structure `struct line`, qui représente une ligne de commande et qui contient en particulier un tableau de `struct cmd` pour représenter les commandes simples. Pour manipuler la structure `struct line`, vous avez trois fonctions :

- `line_init` : pour initialiser la structure ;
- `line_parse` : pour analyser une chaîne de caractères issue d'une saisie clavier et contruire une structure `struct line` à partir de la chaîne.
- `line_reset` : pour remettre à zéro la structure après l'avoir utilisée.

Vous ne devriez pas avoir à modifier les fichiers `cmdline.h` et `cmdline.c`. Toutefois, si vous constatez un bug ou un manque, n'hésitez pas à m'en faire part pour qu'une version corrigée puisse être mise à disposition.

Le programme de test `cmdline_test.c` "teste" un ensemble de lignes de commandes valides, puis non valides.

Vous devrez prendre en charge les lignes de commande suivantes :

- les commandes simples en avant-plan, avec ou sans redirections ;
- les commandes simples en arrière-plan, avec ou sans redirection ;
- les commandes avec tubes en avant-plan, avec ou sans redirection ;
- les commandes avec tubes en arrière-plan, avec ou sans redirection.

Consignes

Vous devrez suivre les consignes suivantes :
— Rédiger le code et les commentaires **en anglais**.
— Séparer le code en plusieurs unités de compilation si besoin.
— Lire les pages de manuel de manière approfondie pour bien comprendre le comportement de chacune des fonctions utilisées.
— **Vérifier les valeurs de retour** des fonctions appelées.

Réalisation

Exercice 1 : Mise en route

Question 1.1 Construire les deux exécutables. Pour cela, vous pouvez, dans un premier temps, utiliser :

```
gcc -std=c99 -Wall -Wextra -g cmdline.c cmdline_test.c -o cmdline_test
gcc -std=c99 -Wall -Wextra -g cmdline.c fish.c -o fish
```

Tester les deux exécutables, et vérifier que tout fonctionne correctement. Pour terminer le programme `fish`, utiliser la combinaison de touches `CTRL+C` afin d'envoyer le signal `SIGINT` au groupe de processus en avant-plan du terminal.

Question 1.2 Écrire un `Makefile` permettant de générer les 2 fichiers exécutables. Les fichiers sources doivent être compilés avec `gcc` en utilisant les options `-std=c99 -Wall -Wextra -g`.

Question 1.3 Lire le code fourni, en particulier le fichier source `fish.c`, pour comprendre comment il fonctionne.

Exercice 2 : Création d'une bibiothèque dynamique

Question 2.1 Modifier le `Makefile` pour créer une bibiothèque dynamique `libcmdline.so` contenant les fonctions définies dans `cmdline.c`. Les 2 fichiers exécutables devront être liés à cette bibiothèque dynamique.

Exercice 3 : Commande simple

On suppose que la ligne de commande ne contient qu'une seule commande simple, c'est-à-dire qu'elle est sans tube et sans redirection. On suppose aussi qu'on attend la fin de la commande en cours avant de rendre la main à l'utilisateur (c'est-à-dire de réafficher le prompt).

Question 3.1 Traiter le cas sans argument. On prendra garde à bien vérifier si l'exécution de la commande (c'est-à-dire le recouvrement) réussit ou pas, et à afficher un message d'erreur si la commande n'existe pas.

Remarque : le tableau `cmds` de la structure `struct line` se termine par un `NULL`.

Question 3.2 Afficher un message sur l'erreur standard pour savoir comment le processus s'est terminé. S'il s'est terminé normalement, il faut indiquer son status de terminaison. Sinon (c'est-à-dire s'il s'est terminé suite à la délivrance d'un signal), il faut indiquer le numéro du signal qui a terminé le processus. Indice : argument `status` de `wait(2)`.

Remarque : le programme `fish` devra afficher sur l'erreur standard comment se terminent tous les processus qu'il crée.

Question 3.3 Traiter le cas avec arguments.

Exercice 4 : Commandes internes

Nous allons maintenant implémenter quelques commandes internes classiques. Pour cela, avant de lancer la commande, on vérifiera s'il s'agit d'une commande interne.

Question 4.1 Implémenter la commande `exit` qui permet de quitter le shell `fish`. Cette commande est nécessaire pour pouvoir bien gérer les signaux dans la suite, et notamment la combinaison de touches `CTRL+C` (qui jusqu'à présent est utilisée pour terminer le programme `fish`).

Question 4.2 Implémenter la commande `cd` qui permet de changer de répertoire courant. Indice : `chdir(2)`. Remarque : cette commande peut être utilisée sans argument ou avec un argument contenant un caractère `~`. Changer le prompt du shell pour y inclure le nom du répertoire courant. Indices : `getcwd(3)`, `basename(3)`.

Exercice 5 : Redirections

On traitera uniquement les redirections de l'entrée standard et de la sortie standard. Pour bien gérer les redirections, il sera nécessaire d'ouvrir les fichiers concernés en lecture seule ou en écriture seule suivant le type de redirection.

Question 5.1 Gérer les redirections possibles de la commande (entrée standard, et sortie standard en mode `TRUNC` ou en mode `APPEND`).

Exercice 6 : Interruption avec `CTRL+C`

Le shell va maintenant gérer l'interruption des commandes lancées en avant-plan à l'aide de la combinaison de touches `CTRL+C`, qui envoie le signal `SIGINT` au groupe de processus en avant-plan du terminal. Actuellement, quand ce signal est envoyé, il déclenche le gestionnaire par défaut qui consiste à terminer les processus qui le reçoivent. Il sera donc nécessaire de modifier ce gestionnaire de manière à terminer les commandes lancées en avant-plan par le shell `fish`, et pas le shell `fish` lui-même.

Question 6.1 Gérer l'interruption avec la combinaison de touches `CTRL+C` à l'aide de `sigaction(2)` (Rappel : vous n'avez pas le droit d'utiliser `signal(2)`).

Exercice 7 : Commandes en arrière-plan

Le shell va maintenant gérer les commandes en arrière-plan. Dans ce cas, le shell n'attend pas la fin de la commande, mais réaffiche immédiatement le prompt ce qui permet de saisir une nouvelle commande. Cependant, il faut également gérer la fin des processus lancés en arrière-plan. Lorsque qu'un processus se termine, le signal `SIGCHLD` est envoyé à son père : il faudra donc le gérer correctement de manière à éliminer **tous** les processus zombies. Indice : `waitpid(2)`.

Pour les commandes lancées en arrière-plan, si l'entrée standard n'est pas redirigée vers un fichier, il faudra la rediriger vers le fichier `/dev/null`.

Question 7.1 Gérer les commandes lancées en arrière-plan. On prendra garde à bien gérer les intéractions entre les commandes en arrière-plan et la commande en avant-plan, notamment quand on attend la terminaison de la commande en avant-plan.

Remarque : la combinaison de touches `CTRL+C` ne doit pas terminer les commandes lancées en arrière-plan.

Rappel : le shell doit afficher sur son erreur standard comment chacun de ses processus fils se terminent, qu'ils soient lancés en avant-plan ou en arrière-plan.

Exercice 8 : Commandes et tubes

Le shell va maintenant gérer les lignes de commande avec des tubes. Il faut veiller à créer le nombre de tubes suffisant, et à bien faire les redirections aux bons moments. Il faut aussi penser à fermer tous les descripteurs de fichiers inutilisés.

Question 8.1 Gérer les commandes avec un seul tube.

Question 8.2 Gérer les commandes avec un nombre arbitraire de tubes.

Exercice 9 : Bonus : Gestion des patterns shell

Cette partie est facultative, vous devez d'abord avoir fini tout le reste avant de traiter cette partie. Il s'agit ici pour le shell de gérer les patterns shell comme `*` ou `?` dans les noms de fichiers. Pour cela, il existe une fonction qui fait le plus gros du travail : `glob(3)`.

Question 9.1 Gérer les patterns shell à l'aide de `glob(3)`.

Rendu et évaluation

Le projet est à réaliser en monôme.

La note du projet tiendra compte des points suivants (liste non-exhaustive) :
— séparation du projet en plusieurs unités de compilation et bibliothèque(s) [1] ;
— présence d'un Makefile fonctionnel [2] ;
— optimalité du Makefile ;
— qualité du code :
  — découpage en fonctions, **factorisation** ;
  — optimalité des algorithmes ;
  — commentaires dans le code source (chaque fonction doit être précédée d'un bloc de commentaires qui indique son rôle et décrit les paramètres attendus et sa valeur de retour) ;
  — contrôle des valeurs de retour des appels système et fonctions ;
  — absence de fuites et d'erreurs mémoire.

Vous devrez déposer une archive compressée contenant vos programmes source dans le dépôt MOODLE prévu à cet effet avant le dimanche 26 mai 22h00, et indiquer dans un bloc de commentaires placé au début du fichier `fish.c`, votre nom et une conclusion/bilan sur le produit final (ce qui est fait, testé, non fait).

Ne vous y prenez pas à la dernière minute pour faire le dépôt. Celui-ci sera fermé à l'heure dite et aucun retard ne sera accepté par mail.

1. Le projet doit contenir au moins une bibliothèque dynamique : la bibliothèque `libcmdline.so`
2. Il ne doit y avoir qu'un seul fichier Makefile dans votre projet : il doit permettre de produire l'ensemble des exécutables et bibliothèque(s) du projet.