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

Eric Merlet

Université de Franche-Comté – UFR Sciences et Technique
Licence Informatique – 2è année
2020 – 2021

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 commande, éventuellement composée de 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 commandes 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, ainsi qu'un programme de démonstration (`fish.c`) qui vous servira de code de départ. Cette unité de compilation définit une structure `struct line`, qui représente une 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 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 cette unité de compilation. Toutefois, si vous constatez un bug ou un manque, n'hésitez pas à en faire part à vos encadrants 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 commandes 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 Compiler et exécuter les 2 programmes fournis (`cmdline_test.c` et `fish.c`). Vérifier que tout fonctionne correctement. Arrêter le programme `fish` à l'aide de `CTRL+C` qui envoie le signal `SIGINT` au groupe de processus en avant-plan du terminal.

Question 1.2 Ecrire un `Makefile` permettant de générer les 2 fichiers exécutables.

Question 1.3 Lire le fichier `fish.c` pour comprendre comment fonctionne ce programme.

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 qu'il n'y a qu'une seule commande, c'est-à-dire aucun tube et aucune redirection. On suppose aussi qu'on attend la fin de la commande en cours avant de rendre la main au shell.

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.

Question 3.2 Afficher un message sur l'erreur standard pour savoir si le programme s'est terminé normalement (en indiquant dans ce cas le status de terminaison), ou s'il s'est terminé avec un signal (en indiquant le signal qui a terminé le processus). Indice : argument `status` de `wait(2)`.

Question 3.3 Traiter le cas avec arguments. Remarque : le tableau `cmds` de la structure `struct line` se termine par un `NULL`.

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 ou pas à l'aide de `strcmp(3)`.

Question 4.1 Implémenter la commande `exit` qui permet de quitter le shell. Cette commande est nécessaire pour pouvoir bien gérer les signaux dans la suite, et notamment `CTRL+C` (qui pour l'instant permet de sortir du programme).

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 chemin absolu du répertoire courant. Indice : `getcwd(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).

Exercice 6 : Arrêt avec `CTRL+C`

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

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

Exercice 7 : Commande 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 "continue" en permettant la saisie d'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 les processus zombies. Indice : `waitpid(2)`.

Pour les commandes lancés 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 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. 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 des tubes entre les commandes. 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 motifs

Cette partie est facultative, vous devez d'abord avoir fini tout le reste avant de vous attaquer à cette partie. Il s'agit ici pour le shell de gérer les motifs 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 motifs à l'aide de `glob(3)`.

Évaluation

La note du projet tiendra compte des points suivants (liste non-exhaustive) :
- la factorisation et la qualité du code ;
- la présence d'un Makefile fonctionnel ;
- les commentaires dans le code source (chaque fonction doit être précédée d'un bloc de commentaires indiquant notamment son rôle, les paramètres attendus, et le cas échéant ce qu'elle renvoie) ;
- le contrôle des valeurs de retour des appels système et fonctions ;
- l'absence de fuites mémoire.

Vous devrez déposer une archive compressée contenant vos programmes source et un fichier texte README dans le dépôt MOODLE prévu à cet effet avant la date indiquée. Cette archive aura pour nom les deux noms du binôme, mis bout à bout et séparés par le caractère underscore (exemple : «Dupont_Durand.tar.gz»).

Le fichier README doit contenir :
- une conclusion/bilan sur le produit final (ce qui est fait, testé, non fait)
- une présentation des améliorations possibles mais non réalisées
- un bilan par rapport au travail en binôme