L2 MIDO 2020 — Programmation Unix
Sur une idée d'E. Lazard et D. Galland.

Version du 03 Avril 2020 : dernière version disponible ici.

Projet mini-shell

Il s'agit de réaliser une version simplifiée du programme `sh`, un des interpréteurs de commande d'UNIX. On pourra se référer au manuel de `sh` (ou d'un autre shell comme `bash`) pour ses caractéristiques complètes. Le sous-ensemble demandé consiste au minimum à gérer le lancement des commandes (au premier plan ou en arrière-plan), les redirections et les tubes entre commandes.

Exemple de commandes devant être traitées :

```
ls
ls > toto &
ls | grep toto
ls | grep toto > tutu
ls -l ; ls
```

Contenu de l'archive

Une archive contenant une base de code et des tests est disponible sur le site web du cours. Téléchargez l'archive et décompressez la dans votre répertoire `home` (ou ailleurs).

L'archive contient :

- un fichier `Makefile` pour compiler le projet et lancer les tests ;

- un module `analex.h analex.c testlex.c` pour vous faciliter le travail d'analyse lexicale des commandes (voir ci-après);

- un fichier `minishell.c` que vous devez compléter ;

- un script `runtest.sh` et un repertoire tests contenant des tests pour votre projet.

Vous pouvez compiler votre projet avec la commande `make` et l'exécuter en tapant `./minishell`. Vous pouvez également lancer la batterie de tests disponibles dans le réperoire `tests/` avec la commande `make test`. Les tests sont des séries de commandes enregistrées dans les fichiers `*.ms` et le résultat attendu de l'exécution de ces commandes est stocké dans les fichiers `*.ms.sol` correspondants. Un test réussit si la série de commande produit le résultat attendu lorsqu'on l'exécute avec votre minishell. Par exemple, le test `01_echo.ms` exécute la commande `echo salut`, et vérifie que votre minishell à bien retourné le code d'erreur `0` et affiché `salut` sur la sortie standard.

Dans sa version actuelle, le programme minishell n'est capable d'exécuter aucune commande et ne résoud aucun test. À vous de faire en sorte que votre programme fonctionne et passe tous les tests. Notez que le tests ne sont pas nécéssairements complets, vous êtes encouragés à rajouter des tests si vous pensez que c'est nécéssaire.

Le contenu de l'archive est décrit plus précisement dans le fichier `README`, lisez le fichier `README` en détail pour comprendre l'utilisation de l'archive, et pensez à mettre votre nom dans la section « auteur(s) » du fichier.

Spécifications simplifiées du shell

Une commande simple consiste en une suite de mots séparés par un ou plusieurs séparateurs (blancs, tabulations ou caractères spéciaux). Le premier mot spécifie le nom de la commande à exécuter, les autres mots sont passés comme arguments à la commande. Rappel : le nom de la commande est lui-même passé comme argument d'indice 0.

Une commande pipelinée est une séquence d'une ou plusieurs commandes séparées par le caractère barre verticale `|` : la sortie standard de chaque commande, à l'exception de la dernière, est connectée par un tube à l'entrée standard de la commande suivante. Chaque commande est exécutée comme un processus distinct.

Qu'il s'agisse d'une commande simple ou d'une séquence, la commande se termine par un retour-chariot, par un caractère `&` suivi d'un retour-chariot, ou par un caractère `;`. Dans le premier cas, le shell attend la fin de la dernière commande ; dans le deuxième, il lance la commande en arrière-plan et reprend immédiatement la main ; dans le dernier cas, le caractère `;` sépare deux commandes s'exécutant à la suite l'une de l'autre.

Avant l'exécution d'une commande, son entrée standard ou sa sortie standard peut être redirigée :
`< fichier` : redirection de l'entrée de la commande depuis le fichier ;
`> fichier` : redirection de la sortie dans le fichier avec troncature ;
`>> fichier` : redirection de la sortie dans le fichier en ajout dans celui-ci.

Il n'est pas demandé de gérer les mécanismes de protection ou d'expansion des caractères.

Analyse d'une commande

Afin de vous aider à décoder ce que l'utilisateur tape au clavier, un analyseur lexical vous est fourni (`analex.c`). Il lit l'entrée standard et découpe le texte tapé au clavier en différents mots et caractères spéciaux qu'on appelle des *tokens*. Chaque appel à la fonction `getToken(word)` retourne le type du prochain token (mot, retour chariot, bare verticale, etc.) et specifie eventuellement le contenu du token (c'est-à-dire le mot lui-même) dans la variable `word` (uniquement si le token est un mot). (L'ensemble de tous les tokens possibles est disponible sous forme d'une énumération dans le fichier `analex.h`.)

Par exemple, la commande `ls -la` ⏎ est composée de trois tokens : deux mots distincts et un caractère spécial (le retour chariot ou *new line* en anglais). Un premier appel à la fonction `getToken(word)` retournera donc le type du premier token, c'est-à-dire `T_WORD`, et remplira le tableau de caractères `word` passé en paramètre avec le chaîne « `ls` ». Un deuxieme appel retournera à nouveau le token `T_WORD`, et remplira le tableau word avec la chaîne « `-la` » et un troisième appel retournera le token `T_NL` (`NL` pour *newline*), sans modifier le contenu de `word`.

Pour bien comprendre le fonctionnement de l'analyseur lexical, commencez par compiler `testlex.c` avec la commande `make testlex`, puis exécutez `./testlex`, et tapez `ls -la`.

```
$./testlex
ls -la
 Token : T_WORD, valeur: ls
 Token : T_WORD, valeur: -la
 Token : T_NL
```

Essayez également d'autres commandes et analysez le code de `testlex.c` `analex.c` et `analex.h`.

Exécution d'une commande

On traitera d'abord le cas d'une commande simple, par exemple `echo` ou `ls -l`, puis on ajoutera l'exécution en arrière-plan, les redirections et enfin on terminera avec les tubes et le caractère `;`

Commande simple

Pour exécuter une commande simple (comme la commande `echo salut`), votre minishell doit analyser le texte tapé au clavier à l'aide de l'analyseur lexical pour y extraire le nom de l'exécutable (c'est-à-dire, le premier token : `echo`) ainsi que les différents arguments (ici un seul argument : `salut`). Puis, lorsque l'analyseur lexical détecte le retour chariot, votre minishell doit créer un nouveau processus à l'aide de `fork()`, programmer l'exécution de la commande à l'intérieur du processus fils à l'aide de `execvp()`, et mettre le processus parent en attente de la fin de son processus fils grâce à `wait()`, ou `waitpid()`. Lorsque la commande à fini de s'exécuter dans le processus fils, le processus parent doit être prêt à exécuter la commande suivante.

Pour une commande simple, l'algorithme de la fonction `commande()` est le suivant :

```
TOKEN commande() {
 Répéter
    getToken(...);
    Suivant le type du token
        si T_WORD
            stocker une copie dans un tableau tabArgs;
        si T_NL
            terminer le tableau tabArgs par un pointeur NULL;
            pid = executer(tabArgs);
            return T_NL;
        si T_EOF
            return T_EOF;
}
```

La fonction `executer(...)` crée un nouveau processus fils, exécute la commande dans ce fils et renvoie dans le père le numéro du processus fils.

```
pid_t executer(...) {
    fork();
        fils : execvp(...);
        père: return num. fils;
}
```

La fonction `main()` boucle sur l'appel à la fonction `commande()` :

```
main() {
 Répéter
    token = commande();
    si token == T_EOF
        break;
}
```

Traitement de l'attente

Si on ne rencontre pas le token `T_AMPER` (&), on doit, dans la fonction `main()`, attendre la fin du fils lancé avant de redonner la main à l'utilisateur.

```
main() {
 Répéter
    token = commande();
    si token == T_EOF
        break;
    si commande lancée,
        waitpid(pid du fils);
}
```

Traitement des redirections

On modifie la fonction `commande()` en lui ajoutant deux arguments, de telle sorte qu'elle reçoive de `main()` les descripteurs de l'entrée et de la sortie standard (initialement 0 et 1) : `commande(int entree, int sortie)`

(Cela va permettre de gérer plus facilement les tubes dans la section suivante.) Lorsque, par exemple, on détecte dans la fonction `commande()` le token `T_GT` on ouvre le fichier donné par le token suivant :

```
    case T_GT:
        getToken(...); pour obtenir le nom de fichier
        sortie = open(...);
```

Les descripteurs `entree` et `sortie` sont transmis à la fonction `executer(entree, sortie, argv)`, et dans le fils, avant de faire `exec()`, on effectue si nécessaire les redirections.

Traitement du tube

On veut exécuter une commande comme `ls -l | grep toto`. Cette commande sera traitée de gauche à droite. On commence comme précédemment par décoder la partie gauche en stockant les arguments dans le tableau `argv[]` et lorsque l'on rencontre le token `T_BAR`, on crée un tube et on appelle la fonction `executer()` pour exécuter la partie gauche avec redirection de la sortie dans le tube en écriture. Pour traiter la partie droite, on appelle récursivement la fonction `commande()` avec comme entrée le tube en lecture.

Il faudra être attentif à bien fermer dans le père et les deux fils, les extrémités non utilisées du tube.

Traitement du point-virgule

Un `;` sépare deux commandes. Il suffit de le traiter comme un retour-chariot : après traitement de la première commande, dans la fonction `main()` l'appel à `commande()` traitera la partie droite.

Modalités de remise du projet (à lire attentivement)

- Le projet devra être rendu le **Mardi 19 Mai 2020 à 20h**, via MyCourse. Vous devez envoyer une archive au format tar.gz contenant l'intégralité de votre projet, y compris un petit rapport de deux pages environ (plus de détails ci-dessous).

- Le projet est à effectuer individuellement ou par groupe de deux. Les groupes de deux seront évalués plus sévèrement. Si vous souhaitez travailler en groupe, c'est à vous de trouver votre binome.

- Faites le projet sérieusement, ça vous servira pour l'examen s'il est maintenu, sinon le projet contribuera à la note finale de l'UE. Donc **n'échangez pas de code source avec les autres groupes. Un détecteur de plagiat de code sera exécuté sur l'ensemble des projets** (attention, ça fonctionne assez bien). Vous pouvez éventuellement utiliser un petit morceau de code que vous n'avez pas écrit vous même (par exemple, morceau de fonction récupéré sur Stack Overflow) à condition de le signaler dans le rapport. En cas de doute, contactez un enseignant responsable.

- Vous devez remettre un petit rapport d'environ deux pages (ou plus) au format PDF. Le fichier PDF doit s'appeler rapport_*prenom_nom*.pdf (ou rapport_*prenom_nom_prenom_nom*.pdf), et se trouver directement dans le répertoire projet de

votre archive. Le rapport doit détailler les fonctionnalités que vous avez réussit à implémenter, parler des problèmes qui se sont posés, discuter des solutions et des choix d'implémentation, et décrire vos éventuelles sources d'inspiration. Évitez de rappeller l'objectif du projet, nous le conaissons.

- L'archive doit contenir un répertoire projet_minishell_*prenom_nom* (ou projet_*prenom_nom_prenom_nom*). Rappel : vous pouvez renommer un répertoire avec la commande mv *ancien_nom nouveau_nom* et créer une archive tar.gz avec la commande `tar -czvf` *`nom_archive.tar.gz repertoire_à_archiver/`*

```
mv projet_minishell projet_minishell_prenom_nom
tar -czvf projet_minishell_prenom_nom.tar.gz projet_minishell_prenom_nom
```

Avant d'envoyer votre archivez, vérifiez les choses suivantes :
- que votre archive contient bien tous les fichiers source dont on peut avoir besoin pour compiler votre projet
- que make fonctionne et génère au moins un exécutable qui s'appelle minishell.
- que la commande make test fonctionne
- que votre répertoire contient le rapport

Pour être sûr, vérifiez que votre archive fonctionne et que votre projet compile sur un autre ordinateur que le votre. Un projet qui ne marche que sur votre ordinateur, n'est pas un projet qui marche.

Attention : N'oubliez pas d'éléments importants dans l'archive et surtout, ne tentez pas de simuler un oubli accidentel pour envoyer l'archive complète le lendemain (ça arrive très souvent). Tout projet incomplet ou remis en retard sera automatiquement pénalisé.

À part ça, bonne chance :)