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
```

1 Code fourni

Pour récupérer le code de base du projet, cliquez sur le lien ci-dessous, et acceptez le devoir. (`Attention`, vous devez avoir un compte Github, voir TP Git pour plus d'info.)

`https://classroom.github.com/a/ENcTu-VG`

Après avoir accepté le devoir, clonez le nouveau dépôt Git dans votre répertoire *home* ou dans un autre répertoire de votre choix.

Le dépôt que vous venez de cloner contient une base de code et quelques tests pour faciliter le développement du projet. Prenez le temps d'analyser le contenu de cette base de code. Vous y trouverez :

- 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 sous-répertoire `tests/` contenant des tests pour votre projet.
- un script `test_archive.sh` qui vous permet d'initialiser et de tester votre base de code (à exécuter via `make testarchive`, au moins avant la remise du projet.)

Préparation de la base de code Commencez par initialiser le projet en tapant la commande `make` une première fois. Un script s'exécute et vous demande votre nom. Renseignez votre nom et vérifiez qu'il est correcte avant de continuer, si il est correcte tapez `'O'` (pour Oui), et attendez que le script finisse de s'exécuter.

Compiler, exécuter et tester. À tout moment, vous pouvez compiler le projet avec la commande `make` et l'exécuter en tapant `./minishell`. Vous pouvez également exécuter la batterie de tests disponible dans le répertoire `tests/` avec la commande `make test`. Dans sa version initiale, le programme `minishell.c` n'est capable d'exécuter aucune commande et ne passe aucun test. À vous de compléter le fichier `minishell.c` pour faire en sorte que votre minishell fonctionne et passe tous les tests.

Fonctionnement des tests. Chaque test est constitué de deux fichiers : un fichier *test*`.ms` et un fichier *test*`.ms.sol`. Le fichier *test*`.ms` contient une ou plusieurs commandes qui doivent pouvoir être exécutées par votre minishell. Le code de retour et le texte attendu sur la sortie standard après l'exécution des commandes sont stockés dans le deuxième fichier *test*`.ms.sol`. Pour que votre minishell passe un test, il faut qu'il retourne le bon code d'erreur, et produise le bon texte sur la sortie standard. 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 a bien affiché "salut" sur la sortie standard. Notez que le tests ne sont pas nécessairement complets. Vous êtes encouragés à rajouter des tests si vous pensez que c'est nécessaire.

Plus d'infos ... 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.

2 Spécifications simplifiées du *minishell*

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 mots suivants sont les arguments de la commande à exécuter. 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 *pipe* (i.e. la 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 dans un processus distinct.

Qu'il s'agisse d'une commande simple ou d'une séquence, toutes les commandes se terminent 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.

3 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 spécifie éventuellement 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`.

4 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.

Extensions Si vous avez implémenté et testé tous les éléments obligatoires du projet, vous pouvez implémenter les extensions non obligatoires de votre choix. Voici quelques extensions que vous pouvez implémenter.

- Implémentation de l'instruction `'cd'`, elle doit permettre de faire en sorte que le code suivant fonctionne:

```
./minishell
mini-shell>ls
README          analex.h   minishell.c
...
mini-shell>cd tests
mini-shell>ls
01_echo.ms      02_ls.ms.sol         04_file_redir.ms
...
```

  Notez que l'instruction `cd` n'est pas une commande externe (comme `ls`), mais un mot-clé traité directement par bash. En fonction de vos choix d'implémentation, vous pouvez être amené à modifier le module `analex.h/analex.c`, mais **Attention**, votre minishell doit rester compatible avec les spécifications du projet de base, et les tests existants.
- Capturer le signal `SIGINT` pour que votre minishell ne se termine pas lorsque vous tentez de tuer un processus qui s'exécute à l'interieur de votre minishell avec Ctrl - C. (Voir `man 2 signal`.)
- Un système de variable d'environnement similaire à celui de bash.
- Toutes les autres fonctionnalités de bash qui ne sont pas dans votre minishell, complétion automatique, pile, fg/bg, Ctrl-R, structures de contrôle `if then else`. ...

5 Modalités de remise du projet (à lire attentivement)

- Le projet est à effectuer **individuellement**. (Pas d'exception.)
- Votre projet sera récupéré automatiquement depuis votre dépôt *Github classroom* **Dimanche 16 Mai 2020 à 20h.** La récupération des projets est **automatique**, aucun retard n'est possible.
- Vous devez avoir effectué au moins 16 *commits* sur votre dépôt Git. Les commits devront être répartis dans le temps, et refléter les différentes étapes de l'avancement de votre projet. A minima, vous devez faire un commit à chaque fois que vous passez un nouveau test de

la commande `make test`. **Attention cette règle n'est pas optionnelle**. Les projets qui ne respectent pas cette règle seront fortement pénalisés.

- Faites le projet sérieusement, il vous servira pour l'examen. **N'échangez pas de code source avec les autres étudiants**. 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 et en commentaire dans votre code. En cas de doute, contactez un enseignant responsable.
- Vous devez remettre un petit rapport d'environ deux pages au format PDF. Le fichier PDF doit s'appeler `rapport.pdf`, et se trouver dans votre dépôt. 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 connaissons.

checklist Avant la date limite:

- Vérifiez que vous avez ajouté tous les changements, dans tous les fichiers sources de votre projet (avec `git add`), que vous avez enregistré ces changements (avec `git commit`) et que vous avez bien synchronisé votre dépôt local avec votre dépôt Github (avec `git push`). Testez régulièrement votre archive avec la commande `make testarchive`. Vérifiez que tout est bien sur Github en utilisant l'interface web de Github. Clonez votre dépôt Github dans un nouveau répertoire et vérifiez que tout fonctionne.
- Vérifiez que votre archive contient bien tous les fichiers source dont on peut avoir besoin pour compiler votre projet. Pour vérifiez, n'hésitez pas à faire un nouveau clone à partir de votre dépôt Github.
- Vérifiez que `make` fonctionne et génère au moins un exécutable qui s'appelle minishell.
- Vérifiez que la commande make test fonctionne
- Vérifiez que votre répertoire contient le un fichier `rapport.pdf` qui fonctionne et qui n'est pas vide.

À part ça, bonne chance :)