Utilisation des systèmes UNIX

Benjamin Negrevergne

PSL University – Paris Dauphine – Équipes *MILES*

Objectif du projet

Implémenter un programme *comme* bash

```
1  $ ./minishell
2  mini-shell>ls
3  AUTHORS  README         analex.h  minishell.c     no_cheat.sh
4  ...
5  mini-shell>wc -l README
6  65 README
7  mini-shell>ls > tmp
8  mini-shell>wc -l tmp
9  26 tmp
10 mini-shell>ls | wc -l
11 26
```

Informations générales

À remettre:

- Code source fonctionnel + tests
- Petit rapport (2 pages)

Modalités:

- Pas de projets en groupe
- Séances de TP réservées pour faire le projet
- 30% de la note de l'UE, note projet = max(projet, exam)
- L'examen portera en partie sur le projet

Fraude

⚠ la fraude est punie sévèrement.

Qu'est ce que la fraude?

- Copier/recopier le code d'un(e) autre étudiant(e)
- Copier/recopier le code d'un projet similaire
- Copier/recopier des bouts de code trouvés sur internet sans mentionner la source
- Faire faire le projet a quelqu'un d'autre

Mesures anti-fraude

- Modalités de notations dissuasives
- Séances de TP réservées au travail sur le projet
- Remise de l'historique des modifications
- Source code unique
- Logging
- Détecteur de plagiat de code

Base de code

- un fichier `AUTHOR` qu'il faudra compléter avec votre nom **avant toute chose.**

Base de code

- un fichier `AUTHOR` qu'il faudra compléter avec votre nom **avant toute chose.**
- un fichier `Makefile` qui permet de compiler votre projet lancer les tests

Base de code

- un fichier `AUTHOR` qu'il faudra compléter avec votre nom **avant toute chose.**
- un fichier `Makefile` qui permet de compiler votre projet lancer les tests
- des fichiers sources (complets) pour faire **l'analyse lexicale** `analex.h analex.c testlex.c`.

Base de code

- un fichier `AUTHOR` qu'il faudra compléter avec votre nom **avant toute chose.**
- un fichier `Makefile` qui permet de compiler votre projet lancer les tests
- des fichiers sources (complets) pour faire **l'analyse lexicale** `analex.h analex.c testlex.c`.
- un fichier source (à compléter) gérer la création des processus et l'exécution des commandes:
  `minishell.c`.

Base de code

- un fichier `AUTHOR` qu'il faudra compléter avec votre nom **avant toute chose.**
- un fichier `Makefile` qui permet de compiler votre projet lancer les tests
- des fichiers sources (complets) pour faire **l'analyse lexicale** `analex.h analex.c testlex.c`.
- un fichier source (à compléter) gérer la création des processus et l'exécution des commandes:
  `minishell.c`.
- un script `runtest.sh` et un sous-répertoire `tests/` contenant des tests pour votre projet. (exécutés via `make test`)

le module `analex`

Permet d'effectuer **l'analyse lexicale** c'est-à-dire, de découper une commande en **tokens** (= unité lexicale avec un type et une valeur)

le module `analex`

Permet d'effectuer **l'analyse lexicale** c'est-à-dire, de découper une commande en **tokens** (= unité lexicale avec un type et une valeur)

Exemple:

```
1  ls > tmp
```

- Représentation 1: chaîne de 10 caractères
- Représentation 2: séquence de 3 tokens:
  - Token 1: mot '`ls`'
  - Token 2: opérateur '>'
  - Token 3: mot '`tmp`'

L'analyse lexicale permet de passer de la représentation 1 à la représentation 2

le module `analex` (suite)

- Une fonction: `TOKEN getToken(char *word)`
  - Lit le prochain token sur l'entrée standard
  - Retourne le type de ce token
  - Renseigne la valeur de ce token dans la variable `word` passée en paramètre
- Une énumération: `TOKEN`, contient la liste des type possibles

```
1  typedef enum {
2    T_WORD, /* un mot */
3    T_BAR,  /* | */
4    T_SEMI, /* ; */
5    T_AMPER, /* & */
6    T_LT,   /* < */
7    T_GT,   /* > */
8    T_GTGT, /* >> */
9    T_NL,   /* retour-chariot */
10   T_EOF   /* ctrl-d */
11 } TOKEN;
```

testlex.c

La fonction `main` de `testlex.c` permet de tester `getToken`

```
while ( (t = getToken(w)) != T_EOF ) {
  printf(" Token : %s", libToken[t]) ;
  if (t==T_WORD)
    printf(", valeur: %s\n", w);
  else
    printf ("\n");
}
```

testlex.c

La fonction `main` de `testlex.c` permet de tester `getToken`

```
while ( (t = getToken(w)) != T_EOF ) {
  printf(" Token : %s", libToken[t]) ;
  if (t==T_WORD)
    printf(", valeur: %s\n", w);
  else
    printf ("\n");
}
```

```
$ ./testlex
ls > tmp
 Token : T_WORD, valeur: ls
 Token : T_GT
 Token : T_WORD, valeur: tmp
 Token : T_NL
```

testlex.c

La fonction `main` de `testlex.c` permet de tester `getToken`

```
while ( (t = getToken(w)) != T_EOF ) {
  printf(" Token : %s", libToken[t]) ;
  if (t==T_WORD)
    printf(", valeur: %s\n", w);
  else
    printf ("\n");
}
```

```
$ ./testlex
ls > tmp
 Token : T_WORD, valeur: ls
 Token : T_GT
 Token : T_WORD, valeur: tmp
 Token : T_NL
```

- À étudier !

Le fichier source minishell.c

- Deux fonction principales : `commande()` et `exécuter()`

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

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

Le fichier source minishell.c

- Deux fonction principales : `commande()` et `exécuter()`

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

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

- À compléter!

Fonctionnalités à implémenter

À minima

- Commande simple
```
ls
```
- Commande avec redirection de fichier
```
ls > tmp ls >> tmp wc -l < tmp
```
- Pipes
```
ls | tmp
```
- Exécution en arrière plan
```
xcalc &
```
- Séquences de commandes
```
echo "salut1" ; echo "salut2"
```

Tests

Un test par fonctionnalité

tests/01_echo.ms

```
echo salut
```

tests/01_echo.ms.sol

```
0     # code de retour attendu
salut # affichage attendu sur la sortie standard
```

Tests

Un test par fonctionnalité

tests/01_echo.ms

```
echo salut
```

tests/01_echo.ms.sol

```
0     # code de retour attendu
salut # affichage attendu sur la
      sortie standard
```

```
make test
./runtest.sh tests/*.ms
##############################
EXECUTING TEST: tests/01_echo.ms.
##############################
Test file:
    1: echo salut
Test results:
- Return value: minishell has returned the expected return value (0)
- Expected output file:
    1: salut
- Actual output file:
- Diff between the expected and the actual file:
salut                                                   <

- Expected output is incorrect.
Result: TEST HAS FAILED.
```

Outils

- build system : *make*
- questionnaire de source: *git* (github/gitclassroom)
- debugger : *gdb*
- analyse de code : *valgrind*

voir `demo/`

Git

Outil de versionnage de code source

- Garder l'historique des modification
- Synchroniser et fusionner le travail de plusieurs collaborateurs
- Maintenir plusieurs version simultanées du code

Git

Outil de versionnage de code source

- Garder l'historique des modification
- Synchroniser et fusionner le travail de plusieurs collaborateurs
- Maintenir plusieurs version simultanées du code

- Git s'utilise en ligne de commandes (avec la commande `git`)
- `gitk` outil de visualisation de l'historique
- `git gui` interface graphique pour git
- GitHub, GitLab, BitBucket: service d'hébergement de dépôts Git.

Git: utilisation locale

Les commandes suivantes permettent d'apporter des modification à votre dépot lcoal. (Ne nécessite pas de connexion internet)

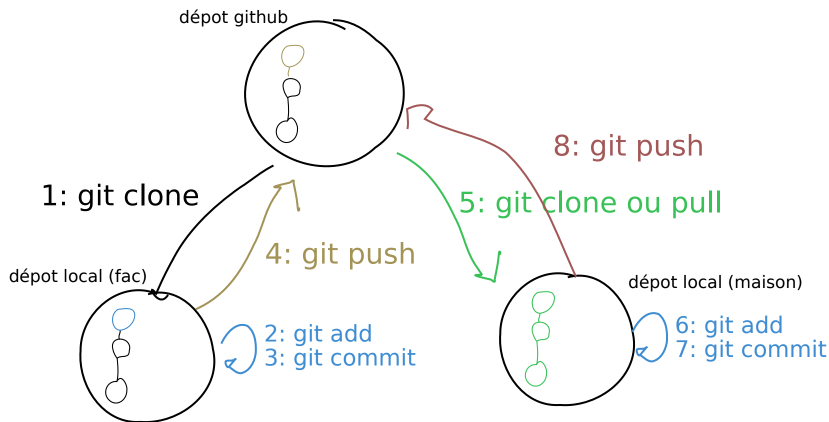
- `git init`: transformer le répertoire en cours en nouveau dépot git (inutile pour le projet)
- `git add <fichier>`: ajouter les changements apportés à `fichier` à l'*index*
- `git commit`: enregistrer tous les changements ajoutés a l'*index* (avec `git add`), et spécifier une brève description pour cet ensemble de changements
- `git restore <fichier>`: remet un fichier dans l'état où il était lors du dernier commit

Git: synchroniser plusieurs dépots git

Les commandes suivantes permettent de synchroniser votre dépot local avec d'autres dépôts a distance. (nécessite une connexion internet)

- `git clone`: récupérer une copie d'un dépot git dans un nouveau répertoire.
- `git pull <remote>`: récupérer et fusionner les *commits* apportés au dépot `remote` (a faire systématiquement avant de commencer a travailler, et avant de faire un git push)
- `git push <remote> <branch>` : envoyer les commits apportés sur le dépot local vers le dépot distant.

Workflow



Quelques règles de base

- Faire de nouveaux commit aussi souvent que possible.
- Faire un `git push` pour diffuser le code quand il est fonctionnel, ou à la fin d'un session.

Si vous êtes perdus

- Clonnez votre dépot GitHub dans un nouveau repertoire
- Modifiez vos sources pour que dépot soit dans l'état que vous souhaitez
- Vérifiez que tout est fonctionne (et que vous n'avez pas écrasés des changements)
- Commitez les changements
- Vérifiez encore
- git push!