MENDY Elie
19004664

Chapitre 10 - Le Projet final

Table des matières

1 Consigne

modifier le shell élémentaire vu au long du cours en y intégrant toutes les modifications vues séparemment dans les différents chapitres.

- Les commandes cd et exit
- Les redirections d'entrées–sorties
- Les pipes
- Les procédures en arrière plan
- Ajouter l'auto-complétion des noms de fichiers(*)

L'archive devra contenir

- les fichiers avec le code source de votre shell
- une makefile complète qui permettra de construire l'exécutable
- La documentation des commandes et accessible par MAN

2 Motivations et Explication de la démarche

J'ai souhaité que le shell rendu dans ce projet s'approche au plus près du shell **bash** dans son comportement. Dans sa finalité, j'ai donc souhaité qu'il soit rendu possible à un utilisateur de lancer des lignes de commandes équivalentes comprenant des redirections et des chainages de commande. Il sera également possible de lancer des processus en arrière-plan et d'autocompléter les noms des fichiers présents dans le répertoire de travail courant.
Je détaillerai donc détaillerons, dans la partie suivante, le développement de chaque fonctionnalité ajouté au mini-shell présenté dans le cours.

La partie quatre présentera le jeu de test pour chaque fonctionnalité ainsi que les résultats obtenus.

Le code source du programme sera donné dans son intégralité dans la partie cinq.

La mise en forme du code a été entièrement révisé et les commentaires ajoutés de manière à décrire au mieux les sections qui le composent.

3 Développement des fonctionnalités

3.1 Les commandes cd et exit

Etant donnée que l'appel système **chdir** doit se faire dans le processus parent, j'ai pris l'initiative de réserver une section *"Commandes internes"* dans le programme du Shell qui nous est donné dans le cours.

Ainsi la fonction **moncd()** sera déclarée à la suite de la fonction main().

Dans la fonction **main()**, une fois que l'on s'est assuré que la ligne de commande comporte au moins un mot, on contrôlera si le premier mot de cette ligne est **"cd"**.

Si le premier mot de la ligne de commande est bien **"cd"** :

- Si aucun chemin n'est donné :
  On changera le répertoire courant par le Home de l'utilisateur.
- Sinon si le nouveau chemin est donné et qu'il est valide :
  Il remplacera le répertoire courant.
  *(via la fonction **chdir()**).*
- Sinon le chemin est erroné :
  On renverra une erreur.

On complètera la section "Commandes Internes" du programme par la fonction exit() qui se chargera de renvoyer une indication de sortie ***"Bye"*** avant de fermer le shell enfant.

```
1       //* ------ COMMANDES INTERNES ------
2       // [commande perso] moncd()
3       if (strcmp(mot[0], "cd") == 0)
4       {
5         moncd(nbMot, mot);
6         continue;
7       }
8
9       // [commande perso] exit()
10      if (strcmp(mot[0], "exit") == 0)
11      {
12        exit(1); // on sort
13      }
```

3.2 Les redirections d'entrées–sorties

3.2.1 Mécanisme d'identification des chevrons

Dans un premier temps, il a fallu coder le mécanisme qui permet de détecter la présence des chevrons sur la ligne de commande du mini-shell.
Ce mécanisme repose sur la déclaration d'un toggle pour chaque chevron à identifier (des entiers à 0 qui passeront à 1 si détection du chevron correspondant lors du parcours de la ligne de commande).

La liste des toggles et de leurs chevrons respectifs est la suivante :

in	<
out	>
outEnd	>>
err	2 >
errEnd	2 >>
andOne	1 > &2
andTwo	2 > &1
and	& >
andEnd	& >>

Une boucle se charge de parcourrir le vecteur **mot[]**, d'identifier les chevrons et de remplir un nouveau vecteur **ldc[]** sur lequel sera lancé l'exécution **execv**.

Les chaines de caractères représentant les directions d'entrées et sorties redirigées seront stockées dans deux nouvelles variables **input**, **output** et **error**.

3.2.2 mécanisme de redirection

Une fois les chevrons identifiés il s'agit maintenant de mettre en place les redirections correspondantes.
De ce fait pour chaque chevron, une clause if effectuera les redirections necessaires à l'aide de la fonction **dup2()**.
Exemple de redirection pour les chervons suivants detectés >> et 2 > &1 :

- les toggles actifs dans ce cas seront **outEnd** et **andTwo**
- On déclarera un file descriptor **fd**

```
1 fd = open(output , O_WRONLY|O_CREAT|O_APPEND, 0666)
2
```

- On effectuera les deux redirections suivantes :

```
1 dup2(fd , STDOUT_FILENO)
2 dup2(fd , STDERR_FILENO)
3
```

- On fermera le file descriptor

On s'assurera de la réussite des appels systemes en les testants.

Les deux mécanismes présentées ci-dessus on été regrouppés dans la fonction **parsingChrvrons()**

3.3 Les pipes

L'objectif du développement de cette fonctionnalité a été de se rapprocher au plus près de l'utilisation des pipes du shell bash.

source : https://stackoverflow.com/questions/8082932/connecting-n-commands-with-pipes-in-a-shell

L'adaptation du code de la source cité ici a permis cette implémentation.

Ci dessous la liste des changement apporté au mécanisme pour adapté au mini-shell

- Encapsulation du mécanisme de d'identification et de redirection lié aux chevrons présents sur une ldc. Cette encapsulation se trouve présentement dans la fonction **parsingChevrons()**.
  Cette fonction sera appelé sur chaque ligne de commande avant son execution.

- Utilisation de la fonction **execvp** permettant de simplifier le code dans sa partie de traitement du chemin d'accès aux différentes commandes.

- Suppression du mécanisme de récupération des différents répertoires du path et de la variable **pathname**.

- Implémentation d'une structure **command**.

- Initialisation d'un vecteur de commandes : **cmd**.

3.4 Les procédures en arrière plan

Il m'a été possible d'ajouter cette fonctionnalité en partant des deux principes suivants :

- Si un processus parent n'attend pas la fin d'un processus enfant, il continuera son execution
- Rediriger la sortie du processus enfant vers "dev/null" me permet de ne pas l'afficher en console

J'ai donc implémenté une variable **background** de type ***int*** instanciée à 0. Cette variable serait mise à 1 si le caractère '' serait identifié en fin de ligne.
(le caractère serait supprimé en fin de ligne de manière à jouer la commande sans encombre).

```
1       //* ------ GESTION LANCEMENT BACKGROUND ------
2       // toggle pour le lancement d'une commande en background
3       int background = 0;
4       if (ligne[strlen(ligne) - 1] == '&')
5       {
6         // modification du toggle
7         background = 1;
8         // supression du caractere '&' à la fin de la LDC
9         ligne[strlen(ligne) - 2] = '\0';
10      }
```

extrait de la section 'DEROULEMENT DU MAIN'

Il m'a fallu ajouter une condition au processus parent avant de lancer l'attente du processus enfant. (selon la valeur de la variable **background**).

```
1       //! PROCESSUS PARENT
2       else if (tmp != 0) // tmp == PID de l'enfant
3       {
4         // attente du la fin du processus
5         // uniquement si lancement au premier plan
6         if (!background)
7           waitpid(tmp, NULL, 0);
8         continue; // relance du prompt
9       }
```

On passe la variable **background** à la fonction **forkPipes()** pour qu'elle la transmette à la fonction **parsingChevrons()**.

```
//* ------ EXECUTION DE LA COMMANDE ------
  //? si pipeline sur la ldc cette execution correspond
  //? a celle de la dernière commande apres le pipeline

  parsingChevrons(cmd[i].argv, background);
  return execvp(commande[0], (char *const *)commande);
```

extrait de la fonction forkPipes()

La redirection vers **'dev/null'** a été implémenté dans la fonction **parsingChevrons()** en passant en argument la valeur de la variable **background**.

```
//* ------  MISE EN PLACE DE OU DES REDIRECTION(S) ------
  //? & lancement en arrière plan
  if (background)
  {
    // redirection vers '/dev/null'
    int fd;
    if ((fd = open("/dev/null", O_WRONLY | O_CREAT | O_TRUNC,
   0666)) < 0)
    {
      perror("Stdout : impossible d'ouvrir le fichier\n");
      exit(1);
    }
    if (dup2(fd, STDOUT_FILENO) < 0 || dup2(fd, STDERR_FILENO) <
    0)
    {
      perror("Fils : erreur lors de la duplication du
   descripteur ");
      exit(1);
    }
    assert(close(fd) >= 0);
    return;
  }
```

extrait de la fonction parsingChevron()

3.5 Ajouter l'autocomplétion des noms de fichiers(*)

Mon implémentation de l'autocomplétion des noms de fichiers se basesur l'utilisation de la librairie **lreadline** et de sa fonction **readline()**

Page du manuel de cette fonction consultable en ligne : https://man7.org/linux/man-pages/man3/readline.3.html

Cette fonction embarque une possibilité de générer une autocomplétion des noms de fichiers.
Il faudra donc télécharger la librairie et l'installer grâce à cette commande :

```
sudo apt install libreadline-dev
```

J'ai donc défini une fonction **saisirLDC()** pour remplacer le mécanisme de saisie utilisateur en place Cette fonction définira un prompt et enregistrera la saisie utilisateur dans la variable globale **ligne**

```
/*
Nom         : usage
Objectif    : logger de message d'erreurs pour l'utilisateur
*/
int saisirLDC(char *str)
{
  char *buffer;
  // affichage du prompt
  // methode permettant l'autocompletion des noms de fichiers
  buffer = readline("\n>>> ");
  add_history(buffer);
  strcpy(str, buffer);

  return 1;
}
```

la boucle du main sera écrite de la manière suivante :

```
for (; saisirLDC(ligne);)
```

4 Expérimentations :test des fonctionnalités

4.1 Les commandes cd et exit

Le raisonnement pour controler le bon fonctionnement des commandes internes **cd** et **exit** est le suivant :

- On lance un shell enfant.
- On s'informe du repertoire courant grâce à la commande **"pwd"**.
- On créer un répertoire **"new_repertoire"**.
- On tente de se déplacer via la commande **"cd new_repertoire"**.
- On s'informe à nouveau du repertoire courant grâce à la commande **"pwd"**.
- La commande fonctionne dans le cas ou le repertoire courant est bien notre nouveau répertoire.
- Quitter proprement le shell par la commande **exit**.

```
elie@NZXT:~/workspace/licence/l2/os/chap10$ menshell

>>> pwd
/media/elie/ARCHIVES/Elie/workspace/licence/l2/os/chap10

>>> ls -l
total 49
-rwxrwxrwx 1 elie elie  6245 janv.  7 19:46 chap10.c
-rwxrwxrwx 1 elie elie   320 janv. 23 16:05 Makefile
-rwxrwxrwx 1 elie elie 22608 janv. 23 16:06 menshell
-rwxrwxrwx 1 elie elie 16172 janv. 17 23:50 menshell.c
-rwxrwxrwx 1 elie elie   245 janv.  7 20:45 sys.h

>>> mkdir new_repertoire

>>> cd new_repertoire/

>>> pwd
/media/elie/ARCHIVES/Elie/workspace/licence/l2/os/chap10/new_repertoire

>>> cd

>>> pwd
/home/elie

>>> exit
elie@NZXT:~/workspace/licence/l2/os/chap10$
```

4.2 Les redirections d'entrées–sorties

Pour s'assurer du bon fonctionnement des redirections nous lancerons la batterie de test suivante :

```
>>> echo "le test 0 à réussi"
"le test 0 à réussi"

>>> echo "le test 1 à reussi" > test1-2

>>> echo "le test 2 à resusi" >> test1-2

>>> cat test1-2
"le test 1 à reussi"
"le test 2 à resusi"
```

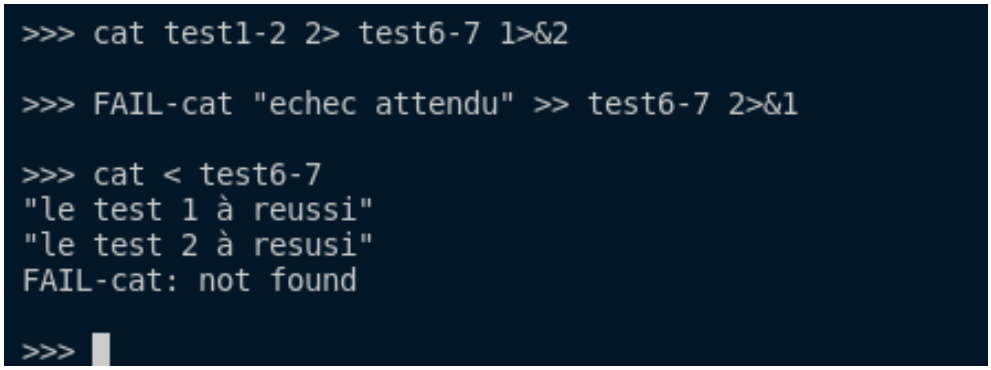
Test des chevrons > et >>

```
>>> FAIL-test3  2> FAIL-test3-4

>>> FAIL-test4  2>> FAIL-test3-4

>>> cat FAIL-test3-4
FAIL-test3: not found
FAIL-test4: not found

>>>
```

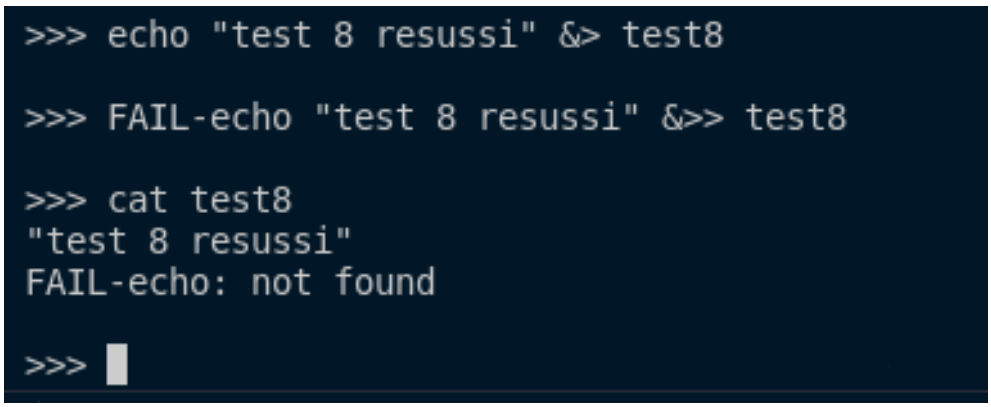
Test des chevrons 2 > et 2 >>

```
>>> cat < FAIL-test3-4 >> test5

>>> cat test5
FAIL-test3: not found
FAIL-test4: not found

>>>
```

Test du chevron chevrons < suivi de >>



Test des chevrons $1 > \&2$ *et* $2 > \&1$



Test des chevrons $\& >$ *et* $\& >>$

4.3 Les pipes

Explications de la démarche :

- On lancera la commande **ls -l** en redirigeant sa sortie dans un fichier **test**.
- Sur la même ligne de commande :
  - On découpera le contenu de ce fichier de manière à récupérer seulement le masque des fichiers avec la commande **cut**.
    *(dans notre mini-shell la ligne de commande est découpée en fonction des caractères " ", la selection d'un délimiteur pour l'option **-d** de la commande **cut** aura donc été le **1** de manière à isoler le masque de chaque fichier dans notre répertoire courant).*
  - On triera ces masques avec la commande **sort**.
  - On eliminera les doublons avec la commande **uniq**.
    La sortie de cette fonction sera redirigée dans un fichier **tmp**
- On affichiera le contenu du fichier tmp avec **cat** en redirigeant la sortie dans un fichier **final**
- Il ne nous restera plus qu'à afficher le contenu du fichier **final**.
- Sans oublier de supprimer les fichiers créés dans ce répertoire pour ne pas le poluer.

Liste des commandes :

- **ls -l > test**
- **cat < test | cut -d1 -f1 | sort | uniq > tmp**
- **cat tmp | du > final**
- **cat final**
- **rm test tmp final**

Lancement du test :
Le jeu de test est contenu dans le fichier **runTest**.
Une fois le shell lancé, il suffit de l'appeler pour lancer la démarche citée ci-dessus.

```
>>> runTest

AFFICHAGE DU FICHIER TEST
total 51
-rwxrwxrwx 1 elie elie  6245 janv.  7 19:46 chap10.c
-rwxrwxrwx 1 elie elie    44 janv. 23 16:15 FAIL-test3-4
-rwxrwxrwx 1 elie elie   320 janv. 23 16:05 Makefile
-rwxrwxrwx 1 elie elie 22608 janv. 23 16:34 menshell
-rwxrwxrwx 1 elie elie 16172 janv. 23 16:34 menshell.c
drwxrwxrwx 1 elie elie     0 janv. 23 16:06 new_repertoire
-rwxrwxrwx 1 elie elie   613 janv. 23 16:44 runTest
-rwxrwxrwx 1 elie elie   245 janv.  7 20:45 sys.h
-rwxrwxrwx 1 elie elie     0 janv. 23 16:52 test

AFFICHAGE DU FICHIER TEMPORAIRE
drwxrwxrwx
-rwxrwxrwx
total 5

AFFICHAGE DU FICHIER FINAL
0       ./new_repertoire
55      .

SUPPRESSION DES FICHIERS (test, tmp et final)


>>>
```

Lancement du fichier ***runTest***

4.4 Les procédures en arrière-plan

Explications de la démarche : Nous dériverons le test inclu dans le fichier **runTest** en ajoutant un caractère **&** pour valider le bon fonctionnement des procédures en arrière-plan.

Liste des commandes :

- **ls -l** -> execution en console
- **ls -l &** -> execution en arrière-plan
- **ls -l > test &** -> redirection + execution en arrière-plan
- **cat test** -> affichage du résultat
- **cat < test | cut -d1 -f1 | sort | uniq** -> pipes + redirection + exécution en console
- **cat < test | cut -d1 -f1 | sort | uniq > result &** -> pipes + redirection + exécution en arrière-plan
- **cat result** -> affichage du résultat

```
>>> ls -l
total 52
-rwxrwxrwx 1 elie elie  6245 janv.  7 19:46 chap10.c
-rwxrwxrwx 1 elie elie    44 janv. 23 16:15 FAIL-test3-4
-rwxrwxrwx 1 elie elie   320 janv. 23 16:05 Makefile
-rwxrwxrwx 1 elie elie 22608 janv. 23 18:53 menshell
-rwxrwxrwx 1 elie elie 16160 janv. 23 18:53 menshell.c
-rwxrwxrwx 1 elie elie    20 janv. 23 18:58 result
-rwxrwxrwx 1 elie elie   613 janv. 23 16:44 runTest
-rwxrwxrwx 1 elie elie   245 janv.  7 20:45 sys.h
-rwxrwxrwx 1 elie elie   431 janv. 23 18:58 test

>>> ls -l &

>>> ls -l > test &

>>> cat test
total 51
-rwxrwxrwx 1 elie elie  6245 janv.  7 19:46 chap10.c
-rwxrwxrwx 1 elie elie    44 janv. 23 16:15 FAIL-test3-4
-rwxrwxrwx 1 elie elie   320 janv. 23 16:05 Makefile
-rwxrwxrwx 1 elie elie 22608 janv. 23 18:53 menshell
-rwxrwxrwx 1 elie elie 16160 janv. 23 18:53 menshell.c
-rwxrwxrwx 1 elie elie    20 janv. 23 18:58 result
-rwxrwxrwx 1 elie elie   613 janv. 23 16:44 runTest
-rwxrwxrwx 1 elie elie   245 janv.  7 20:45 sys.h
-rwxrwxrwx 1 elie elie     0 janv. 23 18:59 test

>>> cat < test | cut -d1 -f1 | sort | uniq
-rwxrwxrwx
total 5

>>> cat < test | cut -d1 -f1 | sort | uniq > result &

>>> cat result
-rwxrwxrwx
total 5

>>>
```

*Lancement du fichier **runTest***

4.5 Ajouter l'autocomplétion des noms de fichiers(*)

Explications de la démarche :

Nous listerons les dossiers du répertoire courant dans un premier temps.
Nous commencerons à écrire le nom d'un fichier dont le début est similaire à un autre.
Nous appuierons sur <tab> pour observer les propositions du shell.

Une fois cette expérience validée nous tenterons de lancer l'autocomplétion des répertoires dans un chemin (ici nous tenterons de simuler un déplacement dans l'arborescence du serveur nginx).

Pour terminer nous tenterons de lancer l'autocomplétion dans le répertoire /**var**/**log**/ qui devrait contenir une grande quantité de fichiers.
L'autocomplétion nous donnera la possibilité de ne pas les afficher par une question (y/n).

Liste des commandes :

- **ls**
- **mensh<tab>** -> autocomplétion du nom de fichier jusqu'à dernière similitude
- **cd /etc/nginx/<tab>** -> affichage du nom de tous les sous-répertoires
- **cd /var/log/<tab>** -> affichage d'une confirmation d'affichage (en raison du nombre important de fichiers dans le répertoire)

```
>>> ls
chap10.c  Makefile  menshell  menshell.c  runTest  sys.h

>>> menshell
menshell    menshell.c

>>> cd /etc/nginx/
conf.d/            modules-available/ sites-enabled/
fastcgi.conf       modules-enabled/   snippets/
fastcgi_params     nginx.conf         uwsgi_params
koi-utf            proxy_params       win-utf
koi-win            scgi_params
mime.types         sites-available/

>>> cd /var/log/
Display all 105 possibilities? (y or n)
```

*Lancement du fichier **runTest***

5 Code source complet du programme

fichier sys.h :

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <string.h>
```

fichier menshell.c :

```c

#include "sys.h"

//*************************************************************
//* DEFINITION DES PRIMITIVES
//*************************************************************

//* ------ CONSTANTES ------

enum
{
  MaxLigne = 1024,          // longueur max d'une ligne de commandes
  MaxMot = MaxLigne / 2, // nb max de mot dans la ligne
  MaxCommandes = 100,     // nb max de commandes par ligne
  MaxDirs = 100,          // nb max de repertoire dans PATH
  MaxPathLength = 512,    // longueur max d'un nom de fichier
};

//* ------- STRUCTURE COMMANDE -------

typedef struct command
{
  char *argv[MaxLigne];
} * command;
```

```
32
33 //* ------ VARIABLES GLOBALES ------
34
35 // [ligne] acceuillera la ligne de commande lue après le prompte
36 char ligne[MaxLigne];
37
38 // [copyLigne] acceuillera copie de la ligne de commande
39 char copyLigne[MaxLigne];
40
41 // [mot] contiendra un découpage de la ligne par mots
42 // sera utilisée pour le prétraitement et le lancement des commandes perso
43 char *mot[MaxMot];
44
45 // [commande] vecteur de chaines utilisé pour l'execution d'une commande
46 // est remplie par le parsing() des chevrons sur une ldc
47 char *commande[MaxMot];
48
49 // [lstCommandes] vecteur de chaines qui contiendra une
50 // liste de commandes (sous forme de chaines)
51 // sera remplie par le découpage() de la copie de la ldc par pipes "|"
52 char *lstCommandes[MaxCommandes];
53
54 // [lstCommandesDecoupees] vecteur qui contiendra une liste
55 // de commandes (sous forme de vecteur de chaines)
56 // sera remplie par le découpage() de chaque commande présentes dans le
    vecteur lstCommandes[]
57 // le decoupage de chaque commande se fera selon le separateur donné dnas
    le cour " \t\n"
58 char *lstCommandesDecoupees[MaxCommandes][MaxLigne];
59
60 // entiers utilisés commes indices pour itérer sur les différentes
    structures
61 int nbMot, nbCommandes, i, tmp, process;
62
63 // stockage des noms des fichiers de redirections
64 char
65     input[128] = {0},  // redirection de l'entrée standard
66     output[128] = {0}, // redirection de la sortie standard
67     error[128] = {0};  // redirection des erreurs
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
```

```
84
85
86
87 //* ------ PROTOTYPES DES FONCTIONS ------
88
89 // logger de message pour l'utilisateur
90 void usage(char *);
91
92 // commande perso - changement de directory
93 int moncd(int, char *[]);
94
95 // decoupage d'une ligne de commande - remplissage d'un vecteur de mots
96 int decouper(char *, char *, char *[], int background);
97
98 // identification des chevrons sur une ligne de commande
99 void parsingChevrons(char *[], int);
100
101 // lancer l'arborescence de processus enfant(s)
102 int forkPipes(int n, struct command *cmd, int background);
103
104 // redirections en cas de pipe(s) + execution commande
105 int spawnProc(int in, int out, struct command *cmd);
106
107 // lancer une saisie utilisateur
108 int saisirLDC(char *str);
109
110 // lancer le vidage du buffer
111 void viderBuffer();
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
```

```
139
140
141 //*************************************************************
142 //* EXECUTION DU MAIN
143 //*************************************************************
144
145 int main(int argc, char *argv[])
146 {
147
148   //* ------ BOUCLE D'EVALUATION ------
149
150   /* lecture et traitement de chaque ligne de commande  */
151   // for (printf(PROMPT); fgets(ligne, sizeof ligne, stdin) != 0; printf(
      PROMPT))
152   for (; saisirLDC(ligne);)
153   {
154     //* ------ GESTION LANCEMENT BACKGROUND ------
155     // toggle pour le lancement d'une commande en background
156     int background = 0;
157     if (ligne[strlen(ligne) - 1] == '&')
158     {
159       // modification du toggle
160       background = 1;
161       // supression du caractere '&' à la fin de la LDC
162       ligne[strlen(ligne) - 2] = '\0';
163     }
164
165     //* ------ COPIE + DECOUPAGE (de la ldc) ------
166     // copie de la ligne de commande (necessaire pour la redécouper par la
       suite)
167     strcpy(copyLigne, ligne);
168
169     // découpage de la ligne + récupération du nombre de mot
170     nbMot = decouper(ligne, " \t\n", mot, MaxMot);
171
172     //* ------ PRE-TRAITEMENT ------
173     // si ligne vide
174     if (mot[0] == 0)
175     {
176       continue;
177     }
178
179     // [commande perso] moncd()
180     if (strcmp(mot[0], "cd") == 0)
181     {
182       moncd(nbMot, mot);
183       continue;
184     }
185
186     // [commande perso] exit()
187     if (strcmp(mot[0], "exit") == 0)
188     {
189       exit(1); // on sort
190     }
191
```

```
192
193
194     //* ------ LANCEMENT PROCESSUS ENFANT ------
195     tmp = fork();
196
197     //! ERREUR FORK
198     if (tmp < 0) // == -1 : Erreur
199     {
200       perror("fork");
201       continue; // relance du prompt
202     }
203
204     //! PROCESSUS PARENT
205     else if (tmp != 0) // tmp == PID de l'enfant
206     {
207       // attente du la fin du processus
208       // uniquement si lancement au premier plan
209       if (!background)
210         waitpid(tmp, NULL, 0);
211       continue; // relance du prompt
212     }
213
214     //! PROCESSUS ENFANT
215     nbCommandes = decouper(copyLigne, "|", lstCommandes, MaxCommandes);
216
217     //* ------ INITIALISATION D'UNE STRUCTURE COMMANDE ------
218     struct command cmd[MaxCommandes];
219
220     //* ------ REMPLISSAGE DE LA STRUCTURE COMMANDE ------
221     for (int c = 0; c < nbCommandes; c++)
222     {
223       decouper(lstCommandes[c], " \t\n", lstCommandesDecoupees[c],
    MaxCommandes);
224
225       for (int i = 0; lstCommandesDecoupees[c][i]; i++)
226       {
227         cmd[c].argv[i] = lstCommandesDecoupees[c][i];
228       }
229     }
230
231     //* ------ EXECUTION DES COMMANDDES DE LA LDC ------
232     //? mecanisme d'execution -->  voir fonctions fork_pipes() et
    spawn_proc()
233     forkPipes(nbCommandes, cmd, background);
234
235     //* ------ COMMANDE NON TROUVEE ------
236     fprintf(stderr, "%s: not found\n", commande[0]);
237     exit(1);
238   }
239
240   //* ------ SORTIE DU SHELL ------
241   printf("Bye\n");
242   return 0;
243 }
244
```

```
245
246
247 //*************************************************************
248 //* DEFINITION DE FONCTIONS
249 //*************************************************************
250
251
252 /*
253 Nom       : usage
254 Objectif    : logger de message d'erreurs pour l'utilisateur
255 */
256 int saisirLDC(char *str)
257 {
258   char *buffer;
259   // affichage du prompt
260   // methode permettant l'autocompletion des noms de fichiers
261   buffer = readline("\n>>> ");
262   add_history(buffer);
263   strcpy(str, buffer);
264
265   return 1;
266 }
267
268 /*
269 Nom       : viderBuffer
270 */
271 void viderBuffer()
272 {
273   while (getchar() != '\n')
274     ;
275 }
276
277 /*
278 Nom       : usage
279 Objectif    : logger de message d'erreurs pour l'utilisateur
280 */
281 void usage(char *message)
282 {
283   fprintf(stderr, "%s\n", message);
284   exit(1);
285 }
286
287 /*
288 Nom       : decouper()
289 Objectif    : decoupe une chaine en mots
290 */
291 int decouper(char *ligne, char *separ, char *mot[], int maxmot)
292 {
293   int i; // compteur de mots
294
295   // decoupe de la ligne + remplissage du vecteur 'mot'
296   mot[0] = strtok(ligne, separ);
297   for (i = 1; mot[i - 1] != 0; i++)
298   {
299     if (i == maxmot)
```

```
300       {
301         fprintf(stderr, "Err. decouper() : trop de mots\n");
302         mot[i - 1] = 0;
303         break;
304       }
305       mot[i] = strtok(NULL, separ);
306     }
307 
308     // retour du nombre de mots parsés
309     return i - 1;
310 }
311 
312 /*
313 Nom         : moncd()
314 Objectif    : changement de répertoire
315 */
316 int moncd(int ac, char *av[])
317 {
318     char *dir;
319     int t;
320 
321     //* ----- TRAITEMENT DES ARGS -----
322 
323     // si aucun arg donné à ma cmd : on retourne dans le home
324     if (ac < 2)
325     {
326       dir = getenv("HOME"); // récupèration de la valeur de $home
327       if (dir == 0)
328       {
329         dir = "/tmp";
330       }
331     }
332 
333     // si
334     else if (ac > 2)
335     {
336       fprintf(stderr, "usage: %s [dir]\n", av[0]);
337       return 1;
338       // sinon, on va dans le repertoire donné
339     }
340     else
341     {
342       dir = av[1];
343     }
344 
345     /* FAIRE LE BOULOT */
346     t = chdir(dir);
347     if (t < 0)
348     { // test de l'appel systeme chdir
349       perror(dir);
350     }
351     return 0;
352 }
353 
354 /*
```

```
355 Nom        : parsingChevrons()
356 Objectif    :   identification des chevrons sur une ligne de commande
357                effectuer les redirections impliqués par les chevrons
358 */
359 void parsingChevrons(char *ligne[], int background)
360 {
361
362   //* ------  REINITIALISATION DES TOOGLES (chevrons) ------
363
364   // toggle d'indication de présence
365   // de chevron(s) sur une ligne de commande
366   int chevron = 0;
367
368   // définition de toggles (un pour chaque chevron)
369   int
370     in = 0,       //      <
371     out = 0,      //      >
372     outEnd = 0,   //      >>
373     err = 0,      //      2>
374     errEnd = 0,   //      2>>
375     andOne = 0,   //      2&>1
376     andTwo = 0,   //      1&>1
377     and = 0,      //      &>
378     andEnd = 0;   //      &>>
379
380   //* ------  PARSING DE LA LDC ------
381
382   // pour chaque chevron identifié sur la ldc :
383   // - activation du toggle correspondant
384   // - sauvegarde de l'emplacement de la redirection (input/output/error)
385   for (int i = 0; ligne[i] != NULL; i++)
386   {
387     if (!strcmp(ligne[i], "<"))
388     {
389       chevron = 1;
390       strcpy(input, ligne[i + 1]);
391       in = 1;
392     }
393
394     if (!strcmp(ligne[i], ">"))
395     {
396       chevron = 1;
397       strcpy(output, ligne[i + 1]);
398       out = 1;
399     }
400
401     if (strcmp(ligne[i], "2>") == 0)
402     {
403       chevron = 1;
404       strcpy(error, ligne[i + 1]);
405       err = 1;
406     }
407
408     if (strcmp(ligne[i], ">>") == 0)
409     {
```

```
410         chevron = 1;
411         strcpy(output, ligne[i + 1]);
412         outEnd = 1;
413       }
414
415       if (strcmp(ligne[i], "2>>") == 0)
416       {
417         chevron = 1;
418         strcpy(error, ligne[i + 1]);
419         errEnd = 1;
420       }
421
422       if (strcmp(ligne[i], "1>&2") == 0)
423       {
424         chevron = 1;
425         andOne = 1;
426       }
427
428       if (strcmp(ligne[i], "2>&1") == 0)
429       {
430         chevron = 1;
431         andTwo = 1;
432       }
433
434       if (strcmp(ligne[i], "&>") == 0)
435       {
436         chevron = 1;
437         strcpy(output, ligne[i + 1]);
438         and = 1;
439       }
440
441       if (strcmp(ligne[i], "&>>") == 0)
442       {
443         chevron = 1;
444         strcpy(output, ligne[i + 1]);
445         andEnd = 1;
446       }
447
448       // construction du vecteur ldc contenant tous les mots qui précèdent le
      1er chevron
449       if (!chevron)
450       {
451         commande[i] = strdup(ligne[i]);
452       }
453     }
454
455     //* ------  MISE EN PLACE DE OU DES REDIRECTION(S) ------
456     //? & lancement en arrière plan
457     if (background)
458     {
459       // redirection vers '/dev/null'
460       int fd;
461       if ((fd = open("/dev/null", O_WRONLY | O_CREAT | O_TRUNC, 0666)) < 0)
462       {
463         perror("Stdout : impossible d'ouvrir le fichier\n");
```

```
464         exit(1);
465       }
466
467       if (dup2(fd, STDOUT_FILENO) < 0 || dup2(fd, STDERR_FILENO) < 0)
468       {
469         perror("Fils : erreur lors de la duplication du descripteur ");
470         exit(1);
471       }
472
473       assert(close(fd) >= 0);
474     }
475
476     //? < - redirection stdin
477     if (in)
478     {
479       int fd0;
480       if ((fd0 = open(input, O_RDONLY, 0)) < 0)
481       {
482         perror("Stdin : impossible de lire le fichier\n");
483         exit(0);
484       }
485       if (dup2(fd0, STDIN_FILENO) < 0)
486       {
487         perror("Fils : erreur lors de la duplication du descripteur ");
488         exit(1);
489       }
490       assert(close(fd0) >= 0);
491     }
492
493     //? > - redirection stdout
494     if (out)
495     {
496
497       int fd1;
498       if ((fd1 = open(output, O_WRONLY | O_CREAT | O_TRUNC, 0666)) < 0)
499       {
500         perror("Stdout : impossible d'ouvrir le fichier\n");
501         exit(1);
502       }
503
504       if (dup2(fd1, STDOUT_FILENO) < 0)
505       {
506         perror("Fils : erreur lors de la duplication du descripteur ");
507         exit(1);
508       }
509
510       /* 2>&1 - redirection de stderr en association avec stdout */
511       if (andTwo)
512       {
513         if (dup2(fd1, STDERR_FILENO) < 0)
514         {
515           perror("Fils : erreur lors de la duplication du descripteur ");
516           exit(1);
517         }
518       }
```

```
519
520       assert(close(fd1) >= 0);
521     }
522
523     //? >> - redirection stdout en fin de fichier
524     if (outEnd)
525     {
526       int fd;
527       if ((fd = open(output, O_WRONLY | O_CREAT | O_APPEND, 0666)) < 0)
528       {
529         perror("Stdout : impossible d'ouvrir le fichier\n");
530         exit(0);
531       }
532       if (dup2(fd, STDOUT_FILENO) < 0)
533       {
534         perror("Fils : erreur lors de la duplication du descripteur ");
535         exit(1);
536       }
537
538       /* 2>&1 - redirection de stderr en association avec stdout */
539       if (andTwo)
540       {
541         if (dup2(fd, STDERR_FILENO) < 0)
542         {
543           perror("Fils : erreur lors de la duplication du descripteur ");
544           exit(1);
545         }
546       }
547
548       assert(close(fd) >= 0);
549     }
550
551
552
553
554
555
556
557     //? 2> - redirection stderr
558     if (err)
559     {
560
561       int fd2;
562       if ((fd2 = open(error, O_WRONLY | O_CREAT | O_TRUNC, 0666)) < 0)
563       {
564         perror("Stderr : impossible d'ouvrir le fichier\n");
565         exit(0);
566       }
567
568       if (dup2(fd2, STDERR_FILENO) < 0)
569       {
570         perror("Fils : erreur lors de la duplication du descripteur ");
571         exit(1);
572       }
573
```

```
574       /* 1>&2 - redirection de stdout en association avec stderr */
575       if (andOne)
576       {
577         if (dup2(fd2, STDOUT_FILENO) < 0)
578         {
579           perror("Fils : erreur lors de la duplication du descripteur ");
580           exit(1);
581         }
582       }
583
584       assert(close(fd2) >= 0);
585     }
586
587     //? 2>> - redirection stderr en fin de fichier
588     if (errEnd)
589     {
590
591       int fd2;
592       if ((fd2 = open(error, O_WRONLY | O_CREAT | O_APPEND, 0666)) < 0)
593       {
594         perror("Stderr : impossible d'ouvrir le fichier\n");
595         exit(0);
596       }
597
598       if (dup2(fd2, STDERR_FILENO) < 0)
599       {
600         perror("Fils : erreur lors de la duplication du descripteur ");
601         exit(1);
602       }
603
604       /* 1>&2 - redirection de stdout en association avec stderr */
605       if (andOne)
606       {
607         if (dup2(fd2, STDOUT_FILENO) < 0)
608         {
609           perror("Fils : erreur lors de la duplication du descripteur ");
610           exit(1);
611         }
612       }
613
614       assert(close(fd2) >= 0);
615     }
616
617     //? &> - redirection stdin et stdout au même endroit
618     if (and)
619     {
620       int fd;
621       if ((fd = open(output, O_WRONLY | O_CREAT | O_TRUNC, 0666)) < 0)
622       {
623         perror("Stdout : impossible d'ouvrir le fichier\n");
624         exit(0);
625       }
626       if (dup2(fd, STDOUT_FILENO) < 0 || dup2(fd, STDERR_FILENO) < 0)
627       {
628         perror("Fils : erreur lors de la duplication du descripteur ");
```

```
629         exit(1);
630       }
631       assert(close(fd) >= 0);
632     }
633
634     //? &>> - redirection stdin et stdout au même endroit en fin de fichier
635     if (andEnd)
636     {
637       int fd;
638       if ((fd = open(output, O_WRONLY | O_CREAT | O_APPEND, 0666)) < 0)
639       {
640         perror("Stdout : impossible d'ouvrir le fichier\n");
641         exit(0);
642       }
643       if (dup2(fd, STDOUT_FILENO) < 0 || dup2(fd, STDERR_FILENO) < 0)
644       {
645         perror("Fils : erreur lors de la duplication du descripteur ");
646         exit(1);
647       }
648
649       assert(close(fd) >= 0);
650     }
651   }
652
653   /*
654   Nom         : spawnProc()
655   Objectif    : effectuer les redirections en cas de pipes
656                 execution de la commande
657   */
658   int spawnProc(int in, int out, struct command *cmd)
659   {
660     //! initialisation d'un pid
661     pid_t pid;
662
663     //! lancement d'un processus enfant
664     if ((pid = fork()) == 0)
665
666     //! execution dans processus enfant
667     {
668
669       //* ------ REDIRECTION ENTREE STANDARD ------
670
671       if (in != 0)
672       {
673         dup2(in, 0);
674         close(in);
675       }
676
677       //* ------ REDIRECTION SORTIE STANDARD ------
678
679       if (out != 1)
680       {
681         dup2(out, 1);
682         close(out);
683       }
```

```
684
685       //* ------ EXECUTION DE LA COMMANDE ------
686
687       parsingChevrons(cmd->argv, 0);
688       return execvp(commande[0], (char *const *)commande);
689     }
690
691     //! si parent: retourne le pid
692     return pid;
693   }
694
695   /*
696   Nom         : forkPipes()
697   Objectif    : lancer l'arborescence de processus enfant
698                 correspondant aux commandes présentes sur la ldc (séparées par
        des pipes " | ")
699   */
700   int forkPipes(int n, struct command *cmd, int background)
701   {
702     int i, in, fd[2];
703
704     //? Le premier processus doit obtenir son entrée à partir du descripteur
        de fichier original 0.
705     in = 0;
706
707     //? boucle d'execution de chaque commande sauf la dernière étape du
        pipeline.
708     for (i = 0; i < n - 1; ++i)
709     {
710       pipe(fd);
711
712       //? report de 'in' (de l'iteration précédente) dans fd[1] le file
        descriptor d'ecriture du pipe l'itération précédente
713       spawnProc(in, fd[1], cmd + i);
714
715       //? fermeture du file descriptor d'ecriture
716       close(fd[1]);
717
718       //? capture de la valeur de fd[0] --> le processus enfant lira à partir
        de ce file descriptor
719       in = fd[0];
720     }
721
722     //? définition de stdin comme étant l'entrée de lecture du pipe précédent
723     if (in != 0)
724       dup2(in, 0);
725
726     //* ------ EXECUTION DE LA COMMANDE ------
727     //? si pipeline sur la ldc cette execution correspond
728     //? a celle de la dernière commande apres le pipeline
729     parsingChevrons(cmd[i].argv, background);
730     return execvp(commande[0], (char *const *)commande);
731   }
```

6 Difficultés rencontrées

À titre personnel, je tiens à mentionner qu'il a été particulièrement difficile pour moi de découvrir les notions de bas niveau du système Linux.

Par conséquent, l'implémentation des redirections et les chainages de commandes s'est avéré compliqué car mon but a été de permettre un chainage multiple et la gestion de redirection de plusieurs flux via une même commande.

La deuxième difficulté aura été d'implémenter l'autocomplétion jusqu'à ce que je découvre la librairie qui gère cette partie.

7 Prolongement possible :

Les points que j'identifie comme vecteurs d'amélioration de ce projet sont les suivants :

- Rendre l'interface utilisateur plus attrayante
  - Coloration syntaxique
  - Refonte graphique du prompt (comme sur le shell de la distribution Kali Linux)
- Une implémentation de l'autocomplétion des noms de commande et de leurs paramètres (comme sur le shell **fish**)
- Une gestion de l'historique des commandes
  - Stockage des commandes tapées dans un fichier
  - Implémentation d'une commande **history**
  - Possibilité de remonter dans les commandes tapées avec la flèche haut du clavier
  - Proposer une autocomplétion grisée de la commande en cours d'écriture

8 Conclusion

Ce projet est un des plus instructifs qui m'a été donné de réaliser. Il m'aura permis de prendre en main le système d'exploitation que j'utilise au quotidien, comprendre les mécanismes relatifs aux processus, à leur exécution et la communication entre eux. De plus, Ce projet m'aura permis de prendre conscience du fonctionnement et de l'efficacité d'un shell, de l'interaction homme-machine qui en dépend et des améliorations qui sont possibles à apporter à ce type d'interface.