

Avrile Floro (n° étudiant : 22000086)

Systèmes d'exploitation

Le 25 février 2024



Table des matières

1 Chapitre 10 - Projet Final	3
1.1 Objectifs et motivations du projet	3
1.2 État de la question et méthodologie retenue	3
1.3 Budget, temps alloué, coût	4
1.4 Le Makefile	5
1.5 Les fichiers d'en-tête	6
1.6 Le fichier projet_co-main.c	8
1.7 Le fichier cn-decouper.c	11
1.8 Les commandes cd et exit	12
1.8.1 La commande cd	12
1.8.2 La commande exit	22
1.9 les procédures en arrière-plan	25
1.10 Les redirections d'entrées-sorties	28
1.10.1 La redirection simple en sortie >	31
1.10.2 La redirection double en sortie »	34
1.10.3 La redirection simple de stderr en sortie 2>	36
1.10.4 La redirection double de stderr en sortie 2»	39
1.10.5 La redirection de la sortie et de la sortie d'erreur vers un fichier >&	41
1.10.6 La redirection de l'entrée simple <	45
1.11 Les redirections d'entrées-sorties au sein des pipes	48
1.11.1 La redirection simple en sortie > avec les pipes	50
1.11.2 La redirection double en sortie » avec les pipes	53
1.11.3 La redirection simple de stderr en sortie 2> avec les pipes	55
1.11.4 La redirection double de stderr en sortie 2» avec les pipes	57
1.11.5 La redirection de la sortie et de la sortie d'erreur vers un fichier >& avec les pipes	59
1.11.6 La redirection de l'entrée simple < avec les pipes	62
1.12 Les pipes	64
1.13 L'auto-complétion des noms de fichiers	74
1.14 Le manuel monman	78
1.15 Difficultés rencontrées	83
1.16 Conclusions et perspectives	83
Annexes	84
.1 Le Makefile en intégralité	84
.2 Le fichier "sys.h"	85
.3 Le fichier "f_head.h"	86
.4 Le fichier "projet_co-main.c" en intégralité	88
.5 Le fichier "cn-decouper" en intégralité	91
.6 Le fichier "pipe.c" en intégralité	92
.7 Le fichier "redir_in_pipe.c" en intégralité	97
.8 Le fichier "moncd.c" en intégralité	100
.9 Le fichier "and_monexit.c" en intégralité	101
.10 Le fichier "redirection.c" en intégralité	102
.11 Le fichier "monman.c" en intégralité	106
.12 L'évaluation du cours	107

Programmer avec Linux

Chapitre 10 - Projet Final

Avrile Floro

Étudiante n°22000086

1 Chapitre 10 - Projet Final

10.1 Sujet du Projet

Pour terminer le cours, je vous demande de modifier le shell élémentaire vu au long du cours en y intégrant toutes les modifications vues séparément dans les différents chapitres.

Les points à ajouter au code sont donc :

Eléments que doit comporter votre Shell

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

Dans l'Archive

- 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.

1.1 Objectifs et motivations du projet

Dans le cadre de ce cours de Systèmes d'Exploitation, nous avons, au fur et à mesure du cours, développé et modifié un shell élémentaire.

L'objectif est de pouvoir développer notre propre shell simple et d'y intégrer certaines fonctionnalités basiques telles que : les commandes cd et exit, les procédures en arrière-plan, les redirections d'entrées-sorties, les pipes, l'auto-complétion des noms de fichiers et un manuel d'utilisation man.

1.2 État de la question et méthodologie retenue

Le but de ce projet est le développement d'un shell disposant de certaines des fonctionnalités du shell Bash. Nous nous sommes inspirée des comportements du shell Bash.

Dans le cadre du cours, nous avons développé un shell écrit en C. Dans un souci de cohérence, nous avons décidé de conserver ce langage de programmation. Ainsi, nous avons codé notre shell en C. Par ailleurs, pour faciliter la maintenance et la compréhension du projet, nous avons décidé d'adopter une approche modulaire. C'est-à-dire que nous avons divisé notre programme en plusieurs modules indépendants, chaque module correspondant à une ou plusieurs caractéristiques.

1.3 Budget, temps alloué, coût

Le budget pour ce projet a été nul puisque nous disposions déjà de l'ensemble du matériel informatique nécessaire et que les logiciels utilisés sont libres et gratuits. Le temps alloué pour le projet est d'environ 3-3,5 semaines à 5 jours de travail par semaine. En revanche, la charge de travail pour le projet a été répartie sur l'année puisque les fonctionnalités du shell ont été développées progressivement. De plus, lorsque nous avons commencé à rédiger le projet final, ces fonctionnalités étaient déjà, en très grande partie, intégrées ensemble.

Nous allons présenter les différents éléments du programme, tout en mettant en valeur les fonctionnalités attendues.

1.4 Le Makefile

Nous avons préparé un Makefile qui peut être exécuté avec la commande `make`. Les directives de compilation du Makefile sont `gcc -Wall projet_co-main.c and_monexit.c redirection.c redir_in_pipe.c cn-decouper.c moncd.c pipe.c man.c -lreadline -o projet`. Voici l'intégralité du Makefile :

```
1 # Nom ..... : Makefile
2 # Rôle ..... : Compile projet
3 # Auteur ..... : Avrile Floro
4 # Version ..... : V0.1 du 25/10/23
5 # Licence ..... : réalisé dans le cadre du cours de SE
6 # Usage : Pour exécuter : make (depuis le répertoire)
7
8 all:
9     gcc -Wall projet_co-main.c and_monexit.c redirection.c redir_in_pipe.c cn-decouper.c moncd.c
10    pipe.c man.c -lreadline -o projet
11
12 clean:
13     rm -f projet
```

```
avrile@vmubuntu:~/Desktop/projet$ make
gcc -Wall projet_co-main.c and_monexit.c redirection.c redir_in_pipe.c cn-decouper.c moncd.c pipe.c
man.c -lreadline -o projet
avrile@vmubuntu:~/Desktop/projet$ ./projet
? pwd
/home/avrile/Desktop/projet
? monexit
Bye
avrile@vmubuntu:~/Desktop/projet$ █
```

FIGURE 1 – On exécute le Makefile avec la commande "make". Le programme compile sans erreurs. On lance le programme avec la commande "./projet".

1.5 Les fichiers d'en-tête

Pour faciliter la lecture des modules du programme, nous avons décidé d'ajouter deux fichiers d'en-tête.

Le premier fichier d'en-tête, `sys.h`, contient les bibliothèques utilisées par le programme. Voici son intégralité :

```
1 /*# Nom ..... : sys.h
2 # Rôle ..... : les bibliothèques couramment utilisées
3 # Auteur ..... : Avrile Floro
4 # Version ..... : V0.1 du 30/01/2024
5 # Licence ..... : réalisé dans le cadre du cours de SE */
6
7
8 #include <assert.h>
9 #include <string.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <unistd.h>
13 #include <sys/stat.h>
14 #include <sys/types.h>
15 #include <fcntl.h>
16 #include <sys/wait.h>
17 #include <stdbool.h>
18 #include <errno.h>
19 #include <readline/readline.h>
20 #include <readline/history.h>
```

Le second fichier d'en-tête, `f_head.h`, contient les déclarations des fonctions. Voici son intégralité :

```
1 /* ****
2 # Nom ..... : f_head.h
3 # Rôle ..... : Les déclarations de fonctions
4 # Auteur ..... : Avrile Floro
5 # Version ..... : V0.1 du 02/02/2024
6 # Licence ..... : réalisé dans le cadre du cours de SE
7 *****/
8
9 enum {
10 MaxLigne = 1024, // longueur max d'une ligne de commandes
11 MaxMot = MaxLigne / 2, // nbre max de mot dans la ligne
12 MaxDirs = 100, // nbre max de repertoire dans PATH MaxPathLength = 512,
13 MaxPathLength = 512, // longueur max d'un nom de fichier
14 MaxPipes = 3, // nb de pipes maximum
15 };
16
17
18 int decouper(char *, char *, char **, int);
19
20 /* & */
21 int arriere_plan(int* nb_total_mots, char* mot[], bool* attend_enfant);
22
23 /* MONCD */
24 int appel_mon_cd(char *dir[], char *mot[], int* nb_total_mots);
25 int mon_cd(char *dir[]);
26
27 /* MONEXIT */
28 int mon_exit(char *mot[]);
29
30 int redirection(char** mot, int* nb_total_mots, bool* redir_sortie, bool* stderr_sortie, bool*
31     redir_s_entree, int* old_entree, int* old_sortie, int* old_erreur, bool* redir, bool*
32     reset);
33
34 int gestion_pipes (int nb_pipe_total, int nb_total_mots, char* mot[], char* mots_pipe[] [MaxMot
35     ], int mes_pipes[] [2], char* dirs[], char pathname[], int pathname_size, bool
36     attend_enfant, bool* ilya_pipe);
37
38 int cb_pipe(char* mot[], int nb_total_mots, bool* ilya_pipe);
39
40 int monman(char* mot[]);
```

```

38 int div_tab(int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot], int mes_pipes[][2], int
* nb_mots_lastpipe);
39
40 int un_pipe(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe);
41
42 int deux_pipes(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe);
43
44 int redirection_dans_pipe(char* mots_pipe[][MaxMot], int ligne);
45
46 int mon_exit_pipe(char *mot[]);
47
48
49
50
51 /* CRÉATION REDIRECTIONS*/
52 /* > */
53 int simple_sortie(char** mot, int* nb_total_mots, mode_t mode, bool* redir_sortie, int i) ;
54
55 /* >> */
56 int double_sortie (char** mot, int* nb_total_mots, bool* redir_sortie, mode_t mode, int i) ;
57
58 /* 2> */
59 int stderr_simple_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
int i);
60
61 /* 2>> */
62 int stderr_double_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
int i);
63
64 /* >& */
65 int stderr_stdout_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, bool*
redir_sortie, mode_t mode, int i);
66
67 /* < */
68 int simple_entree(char** mot, int* nb_total_mots, bool* redir_s_entree, mode_t mode, int i) ;
69
70
71 /* RÉINITIALISATION*/
72
73 /* réinitialisation après > et >> */
74 int reset_sortie(bool* redir_sortie, int* old_sortie);
75
76 /* réinitialisation après 2> et 2>> */
77 int reset_stderr(bool* stderr_sortie, int* old_stderr);
78
79 /* réinitialisation après < */
80 int reset_stdin(bool* redir_s_entree, int* old_entree);
81
82
83 /* CRÉATION REDIRECTIONS POUR LES PIPES */
84 /* > */
85 int simple_sortie_pipe(char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i) ;
86
87 /* >> */
88 int double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i) ;
89
90 /* 2> */
91 int stderr_simple_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i);
92
93 /* 2>> */
94 int stderr_double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i);
95
96 /* >& */
97 int stderr_stdout_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i);
98
99 /* < */
100 int simple_entree_pipe(char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i) ;
101
102
103 /* MANUEL */
104 int monman(char* mot[]);
105
106 /* calcul nb mots dans pipes */
107 int nb_mots(char* mots_pipe[][512], int ligne);

```

1.6 Le fichier projet_co-main.c

Le fichier `projet_co-main.c` est le cœur de notre shell. Il inclut plusieurs fonctionnalités comme la gestion des processus en arrière-plan, les commandes spéciales (`moncd`, `monexit`, `monman`), les redirections, les pipes et l'auto-complétion des noms de fichiers.

Dans un premier temps, les fichiers d'en-tête (headers) sont ajoutés au programme. On définit également notre prompt ? .

Au sein de la fonction `main`, on déclare plusieurs variables qui vont être utilisées pour gérer les commandes entrées par l'utilisateur, les redirections et les pipes.

Les valeurs de `stdin`, `stdout` et `stderr` sont sauvegardées pour permettre leur réinitialisation ultérieure. Le chemin d'accès est découpé en répertoires grâce à la fonction `decouper`.

```
1  /* ****
2 # Nom ..... : projet_co-main.c
3 # Rôle ..... : Main du shell simple du cours
4 #           + gestion du "&" pour lancer un processus en arrière plan
5 #           + gestion de la commande "moncd"
6 #           + gestion de la commande "monexit"
7 #           + gestion des redirections: >, >>, 2>, 2>>, >&, <
8 #           + gestion des pipes |, ||
9 #           + manuel "monman" [monman monman, monman moncd, monman monexit]
10 #           + auto-complétion des noms de fichiers (appui sur tab)
11 # Auteur ..... : Avrile Floro
12 # Version ..... : V0.1 du 30/01/2024
13 # Licence ..... : réalisé dans le cadre du cours de SE
14 # Compilation : gcc -Wall -g projet_co-main.c
15 #           cn-decouper.c moncd.c redirection.c man.c -lreadline -o projet
16 #           un makefile a été préaré > make
17 # Usage : Pour exécuter : ./projet
18 #           puis "monexit" pour quitter
19 #*****
20
21 #include "sys.h"
22 #include "f_head.h"
23
24
25 #define PROMPT "? "
26
27 int
28 main(int argc, char * argv[]) {
29     char* ligne; /* pointeur vers ligne pour Readline */
30     char pathname[MaxPathLength]; char * mot[MaxMot];
31     char * dirs [MaxDirs];
32     int i , tmp;
33     int mon_cdrom, out; // pour moncd et monexit
34
35     bool redir = false ; /* si on veut créer une redirection */
36     bool reset = false ; /* si on veut réinitialiser les entrées-sorties */
37
38     bool redir_sortie = false ; /* > et >> et >& */
39     bool stderr_sortie = false ; /* 2> et 2 >> et >& */
40     bool redir_s_entree = false ; /* < */
41
42     int old_entree = dup(0); /* copie de stdin initiale */
43     int old_sortie = dup(1); /* copie de stdout initiale */
44     int old_erreur = dup(2); /* copie la sortie stderr initiale */
45
46     int mes_pipes[MaxPipes][2]; /* tableaux pour les FD des pipes */
47
48     bool ilya_pipe = false ;
49
50     char * mots_pipe[MaxPipes][MaxMot]; /* sous-tab de mots pour c/ côté de pipe*/
51
52     /* Découper UNE COPIE de PATH en répertoires */
53     decouper(strdup(getenv("PATH")), ":", dirs, MaxDirs);
```

La boucle principale du prompt attend les entrées de l'utilisateur et utilise `readline`. Chaque entrée, qui correspond à une ligne, est traitée.

Au sein de la boucle, on gère les processus en arrière-plan (ça correspond à un & en fin de ligne). On vérifie

également si la ligne de commande contient des pipes, des redirections et des commandes telles que `moncd`, `monexit` ou `monman`. S'il y a une commande de ce type, alors elle est traitée sans créer de processus enfant.

Avant d'exécuter une commande, on configure les redirections si cela est nécessaire. Si des pipes sont détectés, on appelle la fonction en charge de la gestion des pipes.

```
55  /* BOUCLE DU PROMPT */
56  while((ligne = readline(PROMPT)) != NULL){
57
58      bool attend_enfant = true;
59      /* réinitiali pour c/ boucle: par défaut, on attend le processus enfant */
60
61      int nb_total_mots = decouper(ligne, " \t\n", mot, MaxMot);
62      /* on récupère le nb total de mot */
63
64      /* ARRIÉRE-PLAN & */
65      arriere_plan(&nb_total_mots, mot, &attend_enfant);
66
67      int nb_pipe_total = cb_pipe(mot, nb_total_mots, &ilya_pipe);
68      /* renvoie nb total de pipes | + permet passage ilya_pipe à vrai */
69
70      /* LES REDIRECTIONS */
71      if (!ilya_pipe){
72          redir = true ;
73          redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &redir_s_entree, &
74          old_entree, &old_sortie, &old_erreur,&redir, &reset);
75      }
76
77      /* LIGNE VIDE */
78      if (mot[0] == 0)
79          continue;
80
81      /* MONCD */
82      if ((!ilya_pipe)&&((mon_cdrom = appel_mon_cd(dirs, mot, &nb_total_mots)) == 0 ||
83      mon_cdrom == 1))
84          continue;
85
86      /* MON_EXIT */
87      if ((!ilya_pipe)&&(out = mon_exit(mot)) == -1)
88          perror("mon_exit");
89
90      /* MONMAN */
91      if ((!ilya_pipe)&&(strcmp(mot[0], "monman") == 0)){
92          monman(mot);
93      }
94
95      /* LES PIPES */
96      else if (ilya_pipe){
97          gestion_pipes(nb_pipe_total, nb_total_mots, mot, mots_pipe, mes_pipes, dirs,
98          pathname, sizeof(pathname), attend_enfant, &ilya_pipe) ;
99      }
```

Lorsqu'une commande n'utilise pas de pipe ou n'est pas une commande personnalisée, alors un processus enfant est créé via un fork. Le processus enfant va tenter d'exécuter la commande avec un `execv`. Si ça échoue, une erreur apparaît.

Après l'exécution d'une commande, les valeurs initiales de `stdin`, `stdout` et `stderr` sont réinitialisées. On s'assure ainsi que les redirections n'affectent pas les commandes futures.

Finalement, si l'utilisateur a entré `monexit` alors le shell termine sa boucle et s'arrête avec un message.

```
99      /* FORK INITIAL SI PAS DE PIPE */
100     else {
101
102         tmp = fork(); /* lancement du processus enfant */
103
104         if (tmp<0){ /* erreur tmp */
105             perror("fork") ;
106             continue;
107         }
108     }
```

```

109     /* PROCESSUS PARENT */
110    else if (tmp > 0){
111        /* si on attend le processus enfant */
112        if (attend_enfant == true){
113            while (wait(NULL) > 0)
114                ;
115        }
116        /* fin du processus enfant */
117
118        /* RÉINITIALISATION STDIN, STDOUT, STDERR */
119        reset = true;
120        redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &
121 redir_s_entree, &old_entree, &old_sortie, &old_erreur,&redir, &reset);
122
123        /* si on n'attend pas le processus enfant,
124         ou que le processus enfant est déjà terminé on continue */
125        continue;
126    }
127
128    /* PROCESSUS ENFANT */
129    else { /* tmp = 0 */
130
131        for(i = 0; dirs[i] != 0; i++){
132            sprintf(pathname, sizeof pathname, "%s/%s", dirs[i], mot[0]);
133            execv(pathname, mot); /* exécution du programme */
134        }
135
136        /* aucun exec n'a fonctionné */
137        fprintf(stderr, "%s: notfound\n",mot[0]);
138        exit(1) ;
139    }
140
141    printf ("Bye\n");
142    return 0;
143
144}

```

1.7 Le fichier cn-decouper.c

Le fichier `cn-decouper.c` est tiré du cours. Il s'agit d'un wrapper autour de la fonction `strtok`. La fonction `decouper` permet de découper une chaîne de caractères (`ligne`) en plusieurs `mots` en utilisant un caractère de séparation `separ`. Ces sous-chaînes sont stockées dans `mot[]` et la fonction renvoie le nombre total de mots dans `ligne`.

En raison de la construction du programme, `i` correspond au nombre total de mots + 1. Avant de nous rendre compte de cela, nous avions adapté notre programme (sans vraiment en comprendre la raison) à ce fait, de sorte que notre programme était fonctionnel en dépit de ce retour. Nous avons cependant décidé de corriger la sortie de `decouper` et d'ajuster notre programme en conséquence, notamment car dans le cadre d'un travail en équipe, cela permettrait de maintenir plus facilement le programme.

Pour que le retour de la fonction `decouper` corresponde au nombre total de mots, nous soustrayons 1 à `i`.

```
1  /* ****
2 # Nom ..... : cn-decouper.c
3 # Rôle ..... : découper ligne en token
4 # Auteur ..... : tiré du cours
5 # Licence ..... : réalisé dans le cadre du cours de SE
6 # Compilation : compilé avec les autres programmes
7 #*****
8
9 /* cn=decouper.c
10 Un wrapper autour de strtok
11 */
12
13 # include <stdio.h>
14 # include <string.h>
15
16 /* decouper == decouper une chaîne en mots */
17
18 int decouper(char * ligne, char * separ, char * mot[], int maxmot) {
19     int i;
20
21     mot[0] = strtok(ligne, separ);
22     for(i = 1; mot[i - 1] != 0; i++) {
23         if (i == maxmot) {
24             fprintf(stderr, "Erreur dans la fonction decouper: trop de mots\n");
25             mot[i - 1] = 0;
26             break;
27         }
28         mot[i] = strtok(NULL, separ);
29     }
30     return i-1; /* = nb total mots */
31 }
32
33 # ifdef TEST
34
35 int
36 main(int ac, char * av[]) {
37     char *elem[10];
38     int i;
39
40     if (ac<3){
41         fprintf(stderr, "usage:%sphrasesepar\n",av[0]);
42         return 1;
43     }
44
45     printf("On decoupe '%s' suivantes '%s' :\n", av[1], av[2]);
46     decouper(av[1],av[2], elem,10);
47
48     for(i =0; elem[i] != 0; i++)
49         printf("%d:%s\n",i, elem[i]);
50
51     return 0;
52 }
53
54 # endif
```

1.8 Les commandes cd et exit

- Les commandes cd et exit

1.8.1 La commande cd

Comme expliqué précédemment, le programme est divisé en modules. Pour la commande `cd`, que nous avons en réalité appelée `moncd`, les deux fichiers les plus importants sont `projet_co-main.c` et `moncd.c`.

Le fichier `projet_co-main.c`

Le fichier `projet_co-main.c` est le coeur du shell. La fonction `main` se trouve dans le fichier `projet_co-main.c`. C'est cette fonction qui contient la boucle du prompt. Dans le prompt, il existe une division entre les fonctions qui sont appelées avant et après le fork. Toutes les fonctions que nous avons personnalisées, et notamment `moncd`, sont appelées avant le `fork()` car leur utilisation n'est pas compatible avec `execv`. En outre, il est important que les modifications effectuées par `moncd` aient lieu dans le processus parent et non pas dans le processus enfant.

Après l'appel du prompt, on vérifie plusieurs éléments (les redirections, la ligne vide, les commandes en arrière-plan). On récupère également le `nb_total_mots` grâce à la fonction `découper`. On le passera en argument à la fonction `appel_mon_cd`.

Finalement, on appelle la fonction `appel_mon_cd`, en assignant sa valeur de retour à l'int `mon_cdrom`, qui a été déclaré précédemment. Pour que la fonction `moncd` soit exécutée, il faut que le booléen `ilya_pipe` soit faux. En effet, si le premier mot de la ligne de commande est `moncd` mais qu'il y a un pipe sur la ligne de commande, le traitement de la commande `moncd` doit être différent (notamment car la commande opérera alors au sein d'un processus enfant).

```
25 #define PROMPT "? "
26
27 int
28 main(int argc, char * argv[]){
33     int mon_cdrom, out; // pour moncd et monexit
55     /* BOUCLE DU PROMPT */
56     while((ligne = readline(PROMPT)) != NULL){
57
58         bool attend_enfant = true;
59         /* réinitialise pour c/ boucle: par défaut, on attend le processus enfant */
60
61         int nb_total_mots = decouper(ligne, "\t\n", mot, MaxMot);
62         /* on récupère le nb total de mot */
63
64         /* ARRIÈRE-PLAN & */
65         arriere_plan(&nb_total_mots, mot, &attend_enfant);
66
67         int nb_pipe_total = cb_pipe(mot, nb_total_mots, &ilya_pipe);
68         /* renvoie nb total de pipes | + permet passage ilya_pipe à vrai */
69
70         /* LES REDIRECTIONS */
71         if (!ilya_pipe){
72             redir = true ;
73             redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &redir_s_entree, &
74             old_entree, &old_sortie, &old_erreur,&redir, &reset);
75         }
76
77         /* LIGNE VIDE */
78         if (mot[0] == 0)
79             continue;
80
81         /* MONCD */
```

```

81     if (((!iloya_pipe)&&((mon_cdrom = appel_mon_cd(dirs, mot, &nb_total_mots)) == 0 ||  
82         mon_cdrom == 1))  
83         continue;

```

Le fichier moncd.c

La mise en oeuvre de la commande `moncd` est possible grâce au fichier `moncd.c`. Ce fichier contient deux fonctions : la fonction `appel_mon_cd` (qui vérifie si la première commande de la ligne est égale à `moncd`) et la fonction `moncd` (qui se charge de changer de répertoire courant).

Nous avons été obligée de reprendre à la fin de notre projet cette fonction, car elle ne fonctionnait pas de manière optimale. C'est dans ce cadre, que nous nous sommes rendue compte que le `nb_total_mots` renvoyé par la fonction `découper` était incrémenté de 1 par rapport à la réalité. C'est à ce moment-là que nous avons décidé de corriger la sortie de la fonction `découper` et d'adapter en conséquence notre programme.

La fonction `appel_mon_cd` vérifie si le premier mot de la ligne est `moncd`. Si c'est le cas, elle vérifie si la commande `moncd` est suivie de plus d'un argument. Si la commande est suivie de deux arguments ou plus, alors la fonction renvoie un `usage` car il ne s'agit pas de l'utilisation attendue. En revanche, si la commande `moncd` est suivie d'un seul argument, alors on appelle `moncd` en lui passant ce dernier en argument. Par ailleurs, si la commande `moncd` n'est suivie d'aucun argument, alors on appelle la fonction `mon_cd` en lui passant un argument `NULL`.

```

1  /* ****  
2 # Nom ..... : moncd.c  
3 # Rôle ..... : la fonction mon_cd permet de changer de répertoire  
4 # Auteur ..... : Avrile Floro  
5 # Version ..... : V0.1 du 01/09/2023  
6 # Licence ..... : réalisé dans le cadre du cours de SE  
7 # Compilation : compiler avec les autres programmes (selon l'appelant)  
8 #*****  
9  
10 #include "sys.h"  
11 #include "f_head.h"  
12  
13 int appel_mon_cd(char *dir[], char *mot[], int* nb_total_mots){  
14     if (strcmp(mot[0], "moncd") == 0){ // si le mot[0] est moncd  
15         if ((*nb_total_mots > 2)&&(mot[2]!=NULL)){ // s'il y a plus de 2 mots  
16             fprintf(stderr, "Usage: moncd [dir]\n");  
17             return 1;  
18         }  
19         if (*nb_total_mots == 2){  
20             mon_cd(&mot[1]); // on lance la fonction mon_cd  
21             return 0;  
22         }  
23     }  
24     else { // si mot[1] n'existe pas  
25         mon_cd(NULL); // on lance la fonction mon_cd  
26         return 0;  
27     }  
28 }  
29  
30     return 2; /* pour ne pas trigger le "continue" dans main */  
31 }

```

La fonction `mon_cd` est la fonction qui est en charge du changement de répertoire courant. Pour ce faire, elle vérifie si son argument `dir` est nul ou non. Nous avons dû ajouter une double vérification car nous obtenions une erreur de segmentation : nous vérifions tout d'abord que le pointeur `dir` n'est pas nul et ensuite qu'il pointe également vers une chaîne de caractères valide.

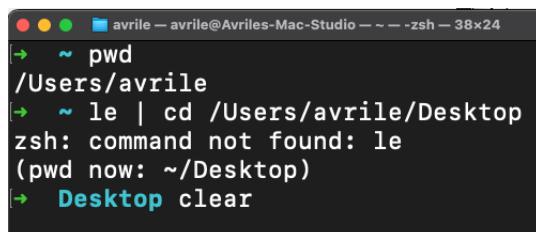
Si la fonction `moncd` a reçu un argument, c'est-à-dire si `dir` n'est pas nul, on change de répertoire grâce à la fonction `chdir`, à laquelle on passe `dir` en argument. On vérifie la bonne exécution de `chdir`. En revanche, si l'argument passé à la fonction `mon_cd` est nul, on récupère le chemin d'accès du répertoire `HOME` avec `getenv("HOME")`, que l'on place dans une variable. On vérifie la bonne exécution de la commande. Puis, on appelle la fonction `chdir` en lui passant la variable contenant le chemin d'accès du répertoire `HOME` en argument.

Tout au long de la fonction, on vérifie que les différentes commandes s'effectuent sans erreurs.

```
34 int mon_cd(char *dir[]){
35     int t;
36
37     if (dir != NULL && *dir != NULL){ // s'il y a un path en argument
38         t = (chdir(*dir)); // on change de répertoire
39         if (t < 0){ // si erreur
40             perror("chdir"); // affiche msg erreur
41             return 1;
42         }
43     }
44
45     else { // s'il n'y a pas de path en argument
46         char * h = getenv("HOME"); // on récupère le path du répertoire HOME
47
48         if (h == NULL){ // si erreur
49             perror("Pas de répertoire HOME"); // affiche msg erreur
50             return 1;
51         }
52
53     else{
54         t = chdir(h) ; // on change de répertoire vers HOME
55
56         if (t < 0){ // si erreur
57             perror("Pas de répertoire HOME"); // affiche msg erreur
58             return 1;
59         }
60     }
61
62     return 0;
63 }
```

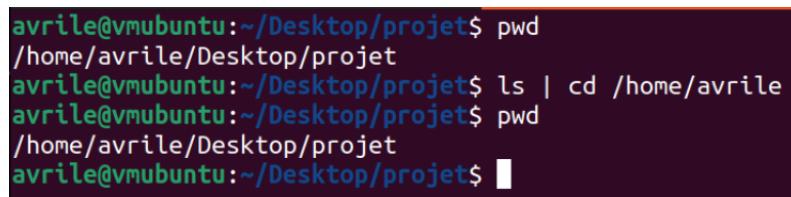
L'utilisation de la commande moncd avec les pipes

Nous avons constaté une différence de comportement entre le shell Zsh sous Mac et le shell Bash sous Linux. Lorsque nous utilisons la commande cd au sein d'un pipe sous zsh, le répertoire courant du processus parent est modifié. En revanche, lorsque nous effectuons la même opération sous Linux (et Bash), le répertoire courant du processus parent n'est pas modifié. Nous avons décidé d'adopter le comportement de Bash en l'espèce.



The screenshot shows a terminal window titled "avrile — avrile@Avriles-Mac-Studio --zsh - 38x24". The user runs "pwd" to show they are in their home directory, "/Users/avrile". Then, they run "ls | cd /Users/avrile/Desktop". The output shows an error: "zsh: command not found: le". This indicates that the current working directory has been changed to "/Users/avrile/Desktop", as the command was run in a child process. Finally, the user runs "clear" to clear the screen.

FIGURE 2 – Lorsque nous utilisons la commande "cd" au sein du shell Zsh dans le cadre d'un pipe, le répertoire courant du processus parent est modifié.



The screenshot shows a terminal window titled "avrile@vmubuntu:~/Desktop/projet\$". The user runs "pwd" to show they are in their project directory, "/home/avrile/Desktop/projet". Then, they run "ls | cd /home/avrile". The output shows they are now in their home directory, "/home/avrile". Finally, the user runs "pwd" again to verify they are still in the project directory, which they are.

FIGURE 3 – Lorsque nous utilisons la commande "cd" au sein du shell Bash sous Linux dans le cadre d'un pipe, le répertoire courant du processus parent n'est pas modifié.

Comme précisé dans la partie précédente, la fonction `appel_mon_cd` prend en argument le `nb_total_mots`. Dans le cadre des pipes, le `nb_total_mots` s'entend comme le nombre total de mots dans un pipe.

Le fonctionnement de `appel_mon_cd` demeure le même. C'est-à-dire que si la commande `moncd` reçoit plus d'un argument, alors la fonction `appel_mon_cd` renvoie un `usage`. Si un seul argument est passé, il est réutilisé lors de l'appel à la fonction `mon_cd`. Enfin, s'il n'y a aucun argument passé à la commande, alors la fonction `mon_cd` est appelée avec un argument `NULL`. La fonction `mon_cd` est elle aussi identique.

Récupérer le nombre total de mots dans les pipes s'est avéré plus délicat que nous ne le pensions. Au final, nous avons adopté une double technique.

Récupérer le nombre de mots dans les lignes non terminales du tableau bidimensionnel

Au sein du fichier `pipe.c` (en charge de la gestion des pipes), nous avons créé une fonction `nb_mots` qui parcourt les éléments d'un tableau et compte les éléments non-nuls. Cette fonction s'est avérée efficace, dans le cadre des pipes, pour compter le nombre d'éléments dans les tableaux pour les lignes non finales. C'est-à-dire que nous l'utilisons, lorsqu'il y a un pipe, pour le processus enfant n°1. Lorsqu'il y a deux pipes, nous l'utilisons pour les processus enfants n°1 et 2. La fonction `nb_mots` renvoie le nombre d'éléments pour une ligne donnée du tableau bidimensionnel.

```
341 int nb_mots(char* mots_pipe[][MaxMot], int ligne){
342     int nb_elmt = 0;
343     for (int i = 0; i < MaxMot; i++){ /* parcourt la ligne */
344         if (mots_pipe[ligne][i] != NULL) /* si non nul */
345             nb_elmt++; /* incrémente nb_elmt */
346     }
347     return nb_elmt; /* renvoie nb élémnt */
348 }
```

Au sein des fonctions `un_pipe` et `deux_pipes` (qui gèrent les pipes), on récupère pour les lignes non terminales, le nombre total de mots grâce à l'appel de la fonction `nb_mots`, dont on associe le résultat à une variable (`nb_elt_tab1` ou `nb_elt_tab2`, selon). On passera cette variable en argument à `appel_mon_cd`.

```
1 /* ****
2 # Nom ..... : pipe.c
3 # Rôle ..... : Gestion des pipes : | et | |
4 # Auteur ..... : Avrile Floro
5 # Version ..... : V0.1 du 26/10/2023
6 # Licence ..... : réalisé dans le cadre du cours de SE
7 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
8 ****
79 int un_pipe(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe){
92     /* PROCESSUS ENFANT 1 */
93     if (tmp ==0) {
98         int nb_elt_tab1 = nb_mots(mots_pipe, 0); /* compte les mots du pipe */
107         if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
108             appel_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
109             exit(0); /* sinon on reste bloqué on attend */
110     }
```

Récupérer le nombre de mots dans les lignes terminales du tableau bidimensionnel

La technique illustrée précédemment ne fonctionne pas pour calculer le nombre de mots dans les lignes terminales. Nos tableaux font 512 mots. La dernière ligne correspondait alors à la taille restante après la soustraction des mots compris dans le ou les pipes précédents. On aurait pu retravailler sur la fonction `nb_mots` pour qu'elle fonctionne et parvienne effectivement à discriminer selon que les cases sont remplies ou non. Nous avons choisi

une méthode différente, qui nous a semblé plus simple.

Pour pouvoir récupérer le nombre de mots dans les lignes terminales de nos tableaux bidimensionnels, nous avons commencé par déclarer un `int* nb_mots_lastpipe`. Nous l'avons déclaré au sein de la fonction `gestion_pipes` qui est la fonction centrale de création et de gestion des pipes. Cette fonction est contenue dans le fichier `pipe.c`.

Nous avons ensuite passé l'`int* nb_mots_lastpipe` en argument de toutes les fonctions en lien avec pipes, c'est-à-dire `div_tab`, `un_pipe` et `deux_pipes`.

```

29 int gestion_pipes (int nb_pipe_total, int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot]
30   , int mes_pipes[][2], char* dirs[], char pathname[], int pathname_size, bool
31   attend_enfant, bool* ilya_pipe){
32
33   int* nb_mots_lastpipe = malloc(sizeof(int));
34   *nb_mots_lastpipe = 0; /* pour calculer longueur pipes */
35
36   if (nb_pipe_total > 0) /* qu'il y ait | ou || */
37     div_tab(nb_total_mots, mot, mots_pipe, mes_pipes, nb_mots_lastpipe);
38     /* on remplit les sous tableau de chaque commande pour les pipes */
39
40   if (nb_pipe_total ==1){ /* s'il y a un || */
41     int v = un_pipe(mes_pipes, pathname, pathname_size, dirs, mots_pipe, attend_enfant,
42     ilya_pipe, nb_mots_lastpipe);
43     if (v == 0 || v == -1)
44       return 0;
45   }
46
47   if (nb_pipe_total==2){ /* s'il y a deux || */
48     int v = deux_pipes(mes_pipes, pathname, pathname_size, dirs, mots_pipe, attend_enfant,
49     ilya_pipe, nb_mots_lastpipe);
50     if (v == 0 || v == -1)
51       return 0;
52   }
53   return 0;
54 }
```

Au sein de la fonction `div_tab`, on va compter pour chaque pipe, le nombre de mots total. La valeur retenue sera celle du dernier pipe car elle a écrasé les valeurs précédentes au sein de la variable `nb_mots_lastpipe`.

```

53 int div_tab(int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot], int mes_pipes[][2], int
54   * nb_mots_lastpipe){
55
56   /* permet de remplir les sous-tableaux [lignes] correspondant aux commandes des pipes */
57
58   int ligne = 0; /* on commence au 1er sous-tableau */
59   int mot_actuel=0; /* on commence au premier mot */
60   for (int i = 0 ; i < nb_total_mots ; i++){
61
62     if (strcmp(mot[i], "|") == 0){ /* si le caractère est un | */
63       mots_pipe[ligne][mot_actuel] = NULL; /* on le remplace par NULL */
64       mot_actuel = 0; /* on recommencera à compter à 0 dans le prochain sous-tableau */
65       (*nb_mots_lastpipe)=0; /* init nb laspipe */
66       ligne++; /* changement de sous-tableau car rencontré un pipe */
67     }
68
69     else { /* si le caractère est pas un | */
70       mots_pipe[ligne][mot_actuel] = mot[i]; /* j'écris le mot dans le tableau */
71       mot_actuel++; /* je parcours les mots */
72       (*nb_mots_lastpipe)++; /* on incrémente */
73     }
74   }
75   mots_pipe[ligne][mot_actuel] = NULL; /* on termine le tableau avec NULL */
76
77   return 0;
78 }
```

Finalement, au sein des fonctions `un_pipe` et `deux_pipes` (fichier `pipe.c`), on réutilise la valeur de `nb_mots_lastpipe` (qui a été passée en argument) et on la passe à la fonction `appel_mon_cd`. On utilise la valeur de `nb_mots_lastpipe`

uniquement pour la dernière ligne (la ligne finale) de nos tableaux bidimensionnels.

```
291 /* PROCESSUS ENFANT 3 */
292 if (tmp3 == 0) {
293     dup2(mes_pipes[1][0], 0); /* redirection entrée depuis pipe2 */
294     close(mes_pipes[1][0]); /* fermeture pipe2 */
295     close(mes_pipes[1][1]);
296
297     redirection_dans_pipe(mots_pipe, 2);
298
299     if (strcmp(mots_pipe[2][0], "monman") == 0){ /* si monman */
300         monman(mots_pipe[2]);
301         exit(0); /* sinon on reste bloqué on attend */
302     }
303
304     if (strcmp(mots_pipe[2][0], "moncd") == 0){ /* si moncd */
305         appel_mon_cd(dirs, mots_pipe[2], nb_mots_lastpipe);
306         exit(0); /* sinon on reste bloqué on attend */
307     }
308
309     if (strcmp(mots_pipe[2][0], "monexit") == 0){ /* si monexit */
310         mon_exit_pipe(mots_pipe[2]);
311         exit(0); /* sinon on reste bloqué on attend */
312     }
313
314     else{
315         for(int i = 0; dirs[i] != 0; i++){
316             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[2][0]);
317             execv(pathname, mots_pipe[2]);
318         }
319
320         fprintf(stderr, "%s: notfound enfant 3\n", mots_pipe[2][0]);
321         exit(1);
322     }
323 }
```

Remarques générales sur l'implémentation de moncd avec les pipes

Au sein de `un_pipe` et `deux_pipes`, la fonction `appel_mon_cd` est appelée avant le `execv`. Ainsi, en cas d'exécution de la fonction `appel_mon_cd`, il n'y aura pas de `execv` puisqu'on aura utilisé `exit(0)`.

L'utilisation de `exit(0)` est nécessaire afin de ne pas rester bloquée. Nous n'utilisions initialement pas `exit(0)` mais le processus enfant ne se terminait pas. ChatGPT nous a proposé de rajouter `exit(0)` à la fin du `if`, et effectivement, cela a débloqué la situation (source : <https://chat.openai.com/>).



FIGURE 4 – C'est ChatGPT qui nous a conseillé de rajouter un `exit(0)` afin de terminer le processus enfant (source : <https://chat.openai.com/>).

Par ailleurs, nous n'appelons la fonction `appel_mon_cd` que si le premier mot de la ligne de commande est `moncd`. Sinon, le `fork()` s'exécute. Nous avons reproduit le même schéma pour chacun des pipes de nos deux fonctions (`un_pipe` et `deux_pipes`).

L'utilisation de `moncd` avec les pipes ne modifie pas le répertoire courant du processus parent car au sein des pipes, on se trouve dans un processus enfant. Néanmoins, il était nécessaire d'intégrer la fonction `appel_mon_cd` aux fonctions de gestion des pipes afin qu'il n'y ait pas d'erreur en cas d'utilisation de la commande `moncd` avec les pipes. Il s'agit du comportement que nous souhaitions imiter, d'après le shell `Bash`.

```

1  /* ****
2 # Nom ..... : pipe.c
3 # Rôle ..... : Gestion des pipes : | et |
4 # Auteur ..... : Avrile Floro
5 # Version ..... : V0.1 du 26/10/2023
6 # Licence ..... : réalisé dans le cadre du cours de SE
7 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
8 ****
92
93     /* PROCESSUS ENFANT 1 */
94    if (tmp ==0) {
95        dup2(mes_pipes[0][1], 1); /* redirection sortie vers pipe */
96        close(mes_pipes[0][0]); /* fermeture pipe */
97        close(mes_pipes[0][1]);
98
99        int nb_elt_tab1 = nb_mots(mots_pipe, 0); /* compte les mots du pipe */
100
101       redirection_dans_pipe(mots_pipe, 0);
102
103       if (strcmp(mots_pipe[0][0], "monman") == 0){ /*si monman*/
104           monman(mots_pipe[0]);
105           exit(0); /* sinon on reste bloqué on attend */
106       }
107
108       if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
109           appell_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
110           exit(0); /* sinon on reste bloqué on attend */
111       }
112
113       if (strcmp(mots_pipe[0][0], "monexit") == 0){ /* si monexit*/
114           mon_exit_pipe(mots_pipe[0]);
115           exit(0); /* sinon on reste bloqué on attend */
116       }
117
118       else{
119           for(int i = 0; dirs[i] != 0; i++){
120               snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
121               execv(pathname, mots_pipe[0]);
122           }
123
124           fprintf(stderr, "%s: not found enfant 1\n", mots_pipe[0][0]);
125           exit(1) ;
126       }
127
128     /* PROCESSUS PARENT > FORK 2 */
129    if ((tmp2=fork()) <0){
130        perror("fork");
131        return -1;
132    }

```

De cette façon, notre shell se comporte de façon identique au shell `Bash`. Nous illustrons cela en utilisant la commande `moncd` dans le cadre de pipe(s). On constate que cela ne renvoie pas d'erreur, néanmoins le répertoire courant du processus parent n'est pas modifié.

```

avrile@vmubuntu:~/Desktop/projet$ ./projet
? pwd
/home/avrile/Desktop/projet
? ls | moncd | wc
    0      0      0
? ls | wc | moncd /home/avrile
? pwd
/home/avrile/Desktop/projet
? ls | moncd /home | wc
    0      0      0
? pwd
/home/avrile/Desktop/projet
? moncd /existepas
chdir: No such file or directory
? moncd | ls
'&'          f_head.h      man.c      proj
and_monexit.c LATEX_PROJET  moncd.c   proj
cn-decouper.c Makefile       pipe.c     redi
? █

```

FIGURE 5 – Notre shell se comporte de la même manière que le shell Bash sous Linux lorsque nous utilisons la commande "moncd" avec des pipes.

Les tests

Nous lançons notre programme avec la commande `./projet`. Nous vérifions notre répertoire courant avec la commande `pwd`, qui nous indique que nous nous trouvons dans le répertoire `moncd /home/avrile/Desktop/projet`. Nous lançons la commande `moncd` (suivi de `pwd` pour vérifier le succès de la commande), et nous constatons que nous nous trouvons dorénavant dans le répertoire `/home/avrile`. Nous continuons de nous déplacer grâce à la commande `/home/avrile/Desktop`. Nous constatons une nouvelle fois, grâce à la commande `pwd`, que notre commande a fonctionné.

Nous avons ensuite illustré deux erreurs. Tout d'abord, lorsque nous passons deux arguments à la commande `moncd`, elle retourne un `Usage`, comme attendu. Par ailleurs, si nous passons en argument un répertoire qui n'existe pas, la fonction `chdir` renvoie une erreur le précisant. Finalement, nous retournons au répertoire initial avec la commande `moncd /home/avrile/Desktop/projet`.

```

avrile@vmubuntu:~/Desktop/projet$ ./projet
? pwd
/home/avrile/Desktop/projet
? moncd
? pwd
/home/avrile
? moncd /home/avrile/Desktop
? pwd
/home/avrile/Desktop
? moncd un deux
Usage: moncd [dir]
? moncd /existe pas
Usage: moncd [dir]
? moncd /existe_pas
chdir: No such file or directory
? moncd /home/avrile/Desktop/projet
? pwd
/home/avrile/Desktop/projet
?

```

FIGURE 6 – Nous illustrons le fonctionnement de la commande "moncd" grâce à des tests.

Afin d'illustrer le fonctionnement de la commande `moncd` avec les pipes, nous réutilisons l'image jointe précédemment. Nous allons la commenter.

On lance la commande `pwd` et on obtient notre répertoire courant. On lance ensuite une commande avec deux pipes en plaçant `moncd` en processus enfant n°2. Puis, nous lançons, dans le cadre d'une commande à deux pipes, la commande `moncd` accompagnée d'un chemin d'accès. La commande s'exécute sans erreur mais le répertoire courant du processus parent n'est pas modifié. C'est le comportement attendu. La commande renvoie une erreur lorsque le répertoire indiqué n'existe pas. Par ailleurs, nous lançons finalement la commande `moncd` suivie d'un pipe et le comportement est bien celui attendu.

```

avrile@vmubuntu:~/Desktop/projet$ ./projet
? pwd
/home/avrile/Desktop/projet
? ls | moncd | wc
      0      0      0
? ls | wc | moncd /home/avrile
? pwd
/home/avrile/Desktop/projet
? ls | moncd /home | wc
      0      0      0
? pwd
/home/avrile/Desktop/projet
? moncd /existepas
chdir: No such file or directory
? moncd | ls
'&'          f_head.h      man.c      proj
and_monexit.c LATEX_PROJET  moncd.c   proj
cn-decouper.c Makefile      pipe.c    redi
? █

```

FIGURE 7 – Nous illustrons le fonctionnement de la commande "moncd" lorsqu'elle est utilisée avec des pipes.

Comme nous avions initialement identifié un comportement indésirable sur cette commande (qui a été corrigé suite aux modifications apportées à notre code et expliquées en détail précédemment), nous avons fait des tests additionnels. Nous constatons que la commande `moncd` se comporte de façon correcte dans tous les cas de figure.

```
avrile@vmubuntu:~/Desktop/projet$ ./projet
? ls | ls | moncd
? ls | ls | moncd lsls ls
Usage: moncd [dir]
? ls | moncd
? ls | moncd /home
? ls | moncd /home home
Usage: moncd [dir]
? pwd
/home/avrile/Desktop/projet
? moncd
? pwd
/home/avrile
? moncd /home/avrile
? pwd
/home/avrile
? moncd /home /home
Usage: moncd [dir]
? ls | moncd existepas
chdir: No such file or directory
? █
```

FIGURE 8 – Nous illustrons le fonctionnement de la commande "moncd" lorsqu'elle est utilisée avec des pipes et sans pipes.

1.8.2 La commande exit

Afin d'étudier la commande `monexit`, nous allons principalement nous intéresser au fichier `projet_co-main.c` et au fichier `and_monexit.c`.

L'implémentation de la commande `monexit` est très similaire à celle de la commande `moncd`, décrite précédemment. Au sein la fonction `main` (fichier `projet_co-main.c`), dans la boucle du prompt, nous appelons la fonction `mon_exit` avant le fork. En effet, si nous essayons de mettre fin au processus parent depuis un processus enfant, cela ne va pas marcher. C'est donc important d'appeler la fonction `monexit` au sein du processus parent et non pas au sein du processus enfant.

Pour que la fonction `mon_exit` soit effectivement lancée, il convient que le booléen `ilya_pipe` soit faux. En effet, lorsqu'il y a des pipes qui sont présents sur la ligne de commande, même si le premier mot de la commande est `monexit`, le comportement attendu est différent. C'est-à-dire que le processus parent ne sera pas fermé si `monexit` est le premier mot d'une ligne de commande contenant un pipe. C'est pour refléter cet état de fait que nous avons ajouté cette condition.

Par ailleurs, nous associons à `out` la valeur de retour de la fonction `monexit`. Normalement, si la fonction s'est déroulée correctement, le reste du code ne va pas s'effectuer puisqu'on aura fermé le processus parent dans lequel on se trouve.

```
27 int
28 main(int argc, char * argv[]){
29
30     int mon_cdrom, out; // pour moncd et monexit
31
32     /* BOUCLE DU PROMPT */
33     while((ligne = readline(PROMPT)) != NULL){
34
35         /* MON_EXIT */
36         if ((!ilya_pipe)&&(out = mon_exit(mot)) == -1))
37             perror("mon_exit");
```

La fonction `mon_exit` se trouve dans le fichier `and_monexit.c`. Au sein de la fonction, on vérifie si le premier mot de la ligne de commande est égal à `monexit`. Si c'est le cas, on quitte le shell sans erreur.

```
34 int mon_exit(char *mot[]){
35     if (strcmp(mot[0], "monexit") == 0){ // si le mot[0] est monexit
36         printf("Bye\n");
37         exit(0); // on quitte le shell sans erreur
38     }
39     return 0;
40 }
```

L'utilisation de la commande monexit avec les pipes

De la même façon que pour la commande `moncd`, nous avons constaté que les comportements du shell `Zsh` et du shell `Bash` étaient différents. Avec `Zsh`, sous Mac, lorsqu'on utilise la commande `exit` avec des pipes, le processus parent est clos. En revanche, sous Linux avec `Bash`, lorsqu'on utilise la commande `exit` avec des pipes, rien ne se passe. C'est-à-dire que le processus parent n'est pas clos. Nous avons reproduit ce comportement au sein de notre mini-shell.

```
avrile@vmubuntu:~$ ls | ls | exit
avrile@vmubuntu:~$
```

FIGURE 9 – Lors de l'utilisation de la commande "exit" avec le shell Bash, le processus parent n'est pas clos.

```
Last login: Wed Jan 31 20:51:57 from 192.168.66.4
avrile ~ ls | exit

Saving session...completed.

[Process completed]
```

FIGURE 10 – Lors de l'utilisation de la commande "exit" avec le shell Zsh, le processus parent est clos.

Afin de pouvoir utiliser la commande `monexit` avec les pipes, nous avons dû modifier le fichier `pipe.c`. Dans les fonctions en charge de la gestion de pipes, nous appelons (si nécessaire) la fonction `monexit_pipe` avant `execv`. C'est-à-dire que si la clause `if` contenant l'appel à la fonction `mon_exit_pipe` est réalisée, alors la clause `else` contenant `execv` ne s'exécutera pas.

Nous avons utilisé ce fonctionnement pour les deux fonctions de gestion des pipes, c'est-à-dire `un_pipe` et `deux_pipes`. Nous avons simplement mis à jour l'appel au tableau bidimensionnel.

```
92     /* PROCESSUS ENFANT 1 */
93     if (tmp == 0) {
94         dup2(mes_pipes[0][1], 1); /* redirection sortie vers pipe */
95         close(mes_pipes[0][0]); /* fermeture pipe */
96         close(mes_pipes[0][1]);
97
98         int nb_elt_tab1 = nb_mots(mots_pipe, 0); /* compte les mots du pipe */
99
100        redirection_dans_pipe(mots_pipe, 0);
101
102        if (strcmp(mots_pipe[0][0], "monman") == 0){ /*si monman*/
103            monman(mots_pipe[0]);
104            exit(0); /* sinon on reste bloqué on attend */
105        }
106
107        if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
108            appell_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
109            exit(0); /* sinon on reste bloqué on attend */
110        }
111
112        if (strcmp(mots_pipe[0][0], "monexit") == 0){ /* si monexit*/
113            mon_exit_pipe(mots_pipe[0]);
114            exit(0); /* sinon on reste bloqué on attend */
115        }
116
117        else{
118            for(int i = 0; dirs[i] != 0; i++){
119                sprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
120                execv(pathname, mots_pipe[0]);
121            }
122
123            fprintf(stderr, "%s: notfound enfant 1\n", mots_pipe[0][0]);
124            exit(1) ;
125        }
126    }
```

Afin que le processus parent ne soit pas clos lors de l'appel à la fonction `mon_exit_pipe` au sein des pipes, nous avons dû légèrement modifier la fonction utilisée.

Au sein du fichier `and_monexit.c`, nous avons créé une nouvelle fonction, `mon_exit_pipe`.

Lors de l'utilisation de la commande `monexit` avec des pipes, c'est la fonction `mon_exit_pipe` qui est utilisée. Cette fonction est similaire à la fonction `mon_exit` présentée précédemment à ceci près qu'elle ne quitte pas le shell avec un `exit(0)`. Cela permet d'obtenir le comportement souhaité : le processus parent n'est pas terminé

lors de l'utilisation de la commande `monexit` avec les pipes.

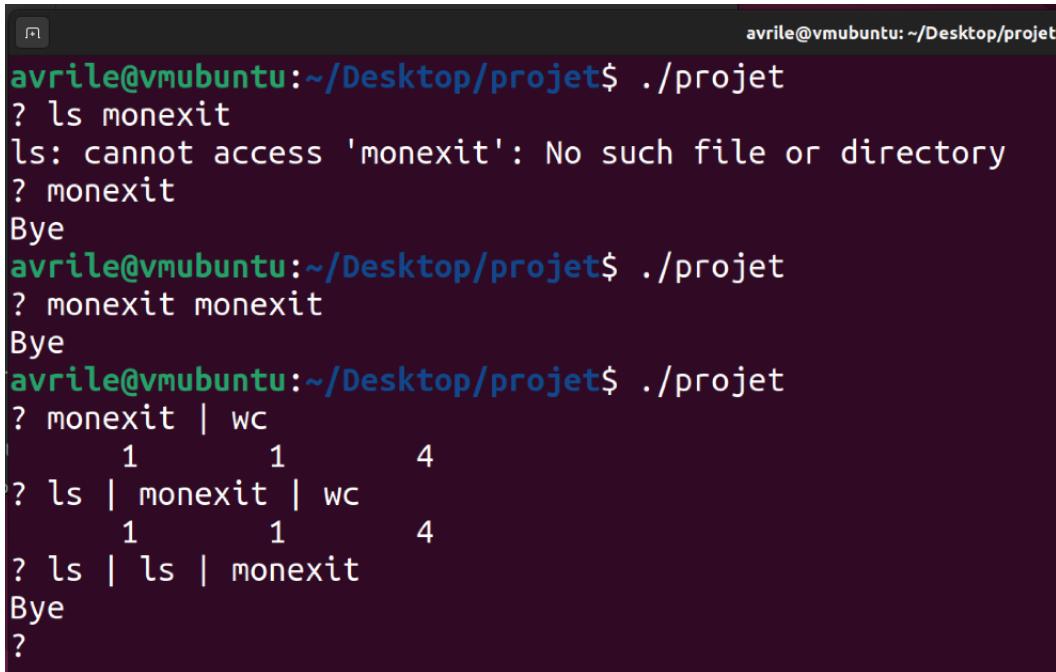
```
43 int mon_exit_pipe(char *mot[]){
44     if (strcmp(mot[0], "monexit") == 0){ // si le mot[0] est monexit
45         printf("Bye\n");
46     }
47     return 0;
48 }
```

Les tests

Pour illustrer le bon fonctionnement de notre commande `monexit`, nous allons effectuer des tests. Nous lançons le programme avec la commande `./projet`. Nous lançons la commande `ls monexit` et nous obtenons une erreur car la commande `ls` n'attend pas d'argument.

Pour être fonctionnelle, la commande `monexit` doit être la première commande. Finalement, nous entrons la commande `monexit` et cela entraîne la fermeture de notre shell. Nous relançons notre shell avec la commande `./projet`, et nous entrons la commande `monexit monexit`, cela fonctionne, le shell est fermé.

Nous relançons le shell et nous testons cette fois la commande `monexit` avec les pipes. Le comportement observé est cohérent avec celui du shell Bash, c'est-à-dire que cela ne cause pas d'erreurs lorsque nous utilisons la commande `monexit` avec des pipes. Néanmoins cela ne ferme pas le processus parent. Par ailleurs, lorsqu'on lance la commande `monexit` en premier sur la ligne de commande, lorsque la commande est suivie par des pipes, le processus parent n'est pas terminé.



```
avrile@vmubuntu:~/Desktop/projet$ ./projet
? ls monexit
ls: cannot access 'monexit': No such file or directory
? monexit
Bye
avrile@vmubuntu:~/Desktop/projet$ ./projet
? monexit monexit
Bye
avrile@vmubuntu:~/Desktop/projet$ ./projet
? monexit | wc
      1      1      4
? ls | monexit | wc
      1      1      4
? ls | ls | monexit
Bye
?
```

FIGURE 11 – Nous illustrons le fonctionnement de la commande "monexit", y compris avec des erreurs.

Conclusion pour les commandes `moncd` et `monexit` : Nous avons inclus les commandes `moncd` et `monexit` dans notre shell. Nous avons illustré leur fonctionnement et nous avons insisté sur leur utilisation avec les pipes. Par ailleurs, nous avons veillé à ce que le comportement des commandes soit conforme au shell Bash, notamment lors de l'utilisation des pipes.

1.9 les procédures en arrière-plan

- Les procédures en arrière plan

Nous avons implémenté le lancement des procédures en arrière-plan. Au sein de la fonction `main` (qui se trouve dans le fichier `projet_co-main.c`), nous déclarons un booléen `attend_enfant` qui, dans chaque boucle du prompt, est déclaré à vrai. C'est-à-dire que, par défaut, on attend que les processus soient terminés avant de continuer.

On récupère également le nombre total de mots, tel qu'il est renvoyé par la fonction `decouper`. On le stocke dans une variable.

Toujours dans la boucle du prompt, avant le `fork()`, on appelle la fonction `arriere_plan`. Cette fonction se trouve dans le fichier `and_monexit.c`.

```
25 #define PROMPT "? "
26
27 int
28 main(int argc, char * argv[]){
29
30     /* BOUCLE DU PROMPT */
31     while((ligne = readline(PROMPT)) != NULL){
32
33         bool attend_enfant = true;
34         /* réinitialise pour c/ boucle: par défaut, on attend le processus enfant */
35
36         int nb_total_mots = decouper(ligne, "\t\n", mot, MaxMot);
37         /* on récupère le nb total de mot */
38
39         /* ARRIÉRE-PLAN & */
40         arriere_plan(&nb_total_mots, mot, &attend_enfant);
41
42     }
43 }
```

La fonction `arriere_plan` vérifie si la ligne contient plus d'un mot et finalement si le dernier mot de la ligne est effectivement `&`. Si c'est le cas, la fonction `arriere_plan` remplace le `&` par `0`, passe le booléen `attend_enfant` à `faux` et met à jour le nombre total de mots pour refléter la suppression du dernier mot. C'est important de supprimer la trace du `&` dans le tableau de mots, sinon cela cause des erreurs lors du lancement de `execv` par le processus enfant.

```
16 int arriere_plan(int* nb_total_mots, char* mot[], bool* attend_enfant){
17
18     if ((*nb_total_mots > 1) && mot[*nb_total_mots-1] && (strcmp (mot[*nb_total_mots-1], "&") == 0)){
19         // s'il y a plus d'un mot et que le dernier mot est "&"
20         *attend_enfant = false;
21         /* pas besoin d'attendre l'enfant, passage à faux */
22
23         mot[*nb_total_mots-1] = 0; /* remplacement & par 0 */
24         /* pour ne pas être pris en compte par execv */
25
26         (*nb_total_mots)--; /* mise à jour du nb de mots */
27         return 0;
28     }
29
30     return 0;
31 }
```

Dès lors, puisque le booléen `attend_enfant` a été passé à `faux`, il convient de ne pas attendre les processus enfants. Cela est fait suite au `fork()`. Au sein du processus parent, on attend les processus enfants seulement si le booléen `attend_enfant` est vrai. Ce n'est pas le cas lorsqu'on est en présence d'un `&` et que le booléen `attend_enfant` a été passé à `faux`.

```
99     /* FORK INITIAL SI PAS DE PIPE */
100    else {
101
102        tmp = fork(); /* lancement du processus enfant */
```

```

103
104     if (tmp<0){ /* erreur tmp */
105         perror("fork") ;
106         continue;
107     }
108
109     /* PROCESSUS PARENT */
110     else if (tmp > 0){
111         /* si on attend le processus enfant */
112         if (attend_enfant == true){
113             while (wait(NULL) > 0)
114                 ;
115         }
116     /* fin du processus enfant */

```

Les tests

Le lancement des processus en arrière-plan est utilisable avec les redirections d'entrées-sorties, ainsi qu'avec les pipes. Il existe néanmoins une réserve pour l'utilisation du `&` avec les pipes : il est nécessaire que le `&` soit le dernier mot de la ligne de commande. Cela ne cause pas toujours une erreur si ce n'est pas le cas, mais la commande ne fonctionnera pas, c'est-à-dire qu'elle ne permettra pas l'exécution en arrière-plan. Pour permettre l'exécution des commandes en arrière-plan, le `&` doit être le dernier élément de la ligne de commande.

Nous lançons des tests pour illustrer le fonctionnement de la commande `&`. On lance le shell avec la commande `./projet`, puis on lance la commande `echo "bonjour" &`. La commande a fonctionné et le processus enfant n'a pas été attendu puisque le prompt apparaît avant le résultat de la commande `echo`. De même, lorsqu'on utilise la commande `ls | wc &`, le prompt apparaît avant le résultat de la commande `wc`. Cela signifie qu'effectivement les processus enfants n'ont pas été attendus. En outre, on a représenté qu'il était possible de lancer des processus en arrière-plan avec les pipes.

Nous avons utilisé les pipes pour illustrer une erreur de fonctionnement de la commande. Le `&` doit toujours être le dernier élément de la ligne de commande, sinon cela peut causer des erreurs. En effet, les commandes interprètent le `&` comme un argument inopportun, comme démontré par les commandes `moncd &` et `ls &`, lorsque ces dernières ne sont pas en dernière position de la ligne de commande.

```

avrile@vmubuntu:~/Desktop/projet$ ./projet
? echo "bonjour" &
? "bonjour"
ls | wc &
?      14      14      152
pwd &
? /home/avrile/Desktop/projet
ls & | wc
ls: cannot access '&': No such file or directory
      0      0      0
? moncd & | wc
chdir: No such file or directory
      0      0      0
?
?
```

FIGURE 12 – On illustre l'utilisation de notre commande `&`.

Le lancement des processus en arrière-plan est également compatible avec les redirections. Nous lançons le shell avec la commande `./projet`, puis nous lançons la commande `echo "bonjour" > testand.txt &`. Nous constatons qu'effectivement, un fichier `testand.txt` a été créé. Ce fichier contient le résultat de la commande `echo`.

FIGURE 13 – Le lancement des processus en arrière-plan est compatible avec l'utilisation des redirections.

Conclusion pour le lancement des processus en arrière-plan : Grâce à la commande &, nous sommes en mesure de lancer des processus en arrière-plan. Nous avons illustré l'utilisation de cette commande avec des exemples, y compris des erreurs. En outre, nous avons montré comment cette commande pouvait être utilisée conjointement avec les pipes et les redirections.

1.10 Les redirections d'entrées-sorties

– Les **redirections** d'entrées–sorties

Nous allons présenter les redirections suivantes : >, >>, 2>, 2>>, >&, et <.

Nous avons utilisé le manuel de Bash (https://www.gnu.org/software/bash/manual/html_node/Redirections.html) pour en apprendre plus sur les redirections.

La gestion des redirections

Au sein du fichier `projet_co-main.c`, dans la fonction `main`, nous déclarons de nombreux booléens et int, dont nous allons nous servir pour la gestion des redirections. Pour chacune des redirections, un booléen déclaré à `false` a été créé. Nous avons aussi gardé une trace de `stdin`, `stdout` et `stderr`. Deux booléens supplémentaires sont utilisés, `redirection` et `reset`.

```
27 int
28 main(int argc, char * argv[]) {
35     bool redir = false ; /* si on veut créer une redirection */
36     bool reset = false ; /* si on veut réinitialiser les entrées-sorties */
37
38     bool redir_sortie = false ; /* > et >> et >& */
39     bool stderr_sortie = false ; /* 2> et 2>> et >& */
40     bool redir_s_entree = false ; /* < */
41
42     int old_entree = dup(0); /* copie de stdin initiale */
43     int old_sortie = dup(1); /* copie de stdout initiale */
44     int old_erreur = dup(2); /* copie la sortie stderr initiale */
```

Les redirections au sein de la fonction main

Dans `main` (fichier `projet_co-main.c`), au sein du prompt principal et avant le `fork()`, s'il n'y a pas de pipe (car les redirections dans les pipes sont traitées à part), on passe le booléen `redir` à `true`. Puis, on appelle la fonction `redirection` afin de procéder aux redirections demandées par la commande, si elles existent.

```
27 int
28 main(int argc, char * argv[]) {
55     /* BOUCLE DU PROMPT */
56     while((ligne = readline(PROMPT)) != NULL){
70
71         /* LES REDIRECTIONS */
72         if (!ilya_pipe){
73             redir = true ;
74             redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &redir_s_entree, &
old_entree, &old_sortie, &old_erreur,&redir, &reset);
    }
```

Dans un second temps, c'est-à-dire après le fork, au sein du processus parent, on passe le booléen `reset` à vrai. Puis on rappelle la fonction `redirection` afin de procéder à la réinitialisation de `stdin`, `stdout` et `stderr`.

```
109     /* PROCESSUS PARENT */
110     else if (tmp > 0){
111         /* si on attend le processus enfant */
112         if (attend_enfant == true){
113             while (wait(NULL) > 0)
```

```

114
115
116
117
118
119
120
121
122
123
124
125

```

Les redirections au sein de la fonction redirection

La fonction `redirection` se trouve dans le fichier `redirection.c`. La fonction `redirection` est la fonction maîtresse pour la gestion des redirections. Selon les booléens se trouvant à vrai, elle est en charge de créer les redirections ou de réinitialiser les entrées et les sorties.

La fonction `redirection` est divisée en deux parties. La première partie, qui s'exécute lorsque le booléen `redirection` est vrai, permet de créer les redirections.

Grâce à une boucle parcourant tous les mots, chaque redirection est créée grâce à une fonction particulière. Une fois qu'une redirection a été créée, on `continue` afin de ne pas exécuter le code qui suit. (Sinon, cela cause un débordement de mémoire à cause de la modification de `nb_total_mots` à l'intérieur des sous-fonctions.) À la fin de la boucle, on passe le booléen `redirection` à faux, afin que, lors du prochain appel de la fonction, seule la partie relative à la réinitialisation s'exécute.

Contrairement aux autres fonctionnalités précédentes de notre shell, nous ne savons pas cette fois-ci à quelle place exactement de la ligne de commande peuvent se trouver les redirections. En effet, elles peuvent être utilisées avec d'autres fonctionnalités (par exemple `&`), ou après des commandes comprenant plusieurs arguments (par exemple `echo "bonjour"`). Pour cette raison, nous avons utilisé une boucle afin de parcourir tous les mots. Il convient de remarquer que cette boucle a pour limite `i < nb_total_mots -1` car chaque redirection est suivie d'un argument, raison pour laquelle la limite est établie à `-1`.

En outre, nous définissons les modes pour l'ensemble des fichiers. Les modes sont définis de façon à ce que tout le monde puisse lire, écrire et exécuter.

```

1  /* ****
2 # Nom ..... : redirection.c
3 # Rôle ..... : Fonction de gestion des redirections:
4 #           >, >>, 2>, 2>>, >&, <
5 # Auteur ..... : Avrile Floro
6 # Version ..... : V0.1 du 18/10/2023
7 # Licence .... : réalisé dans le cadre du cours de SE
8 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
9 #*****
10
11 #include "sys.h"
12 #include "f_head.h"
13
14 int redirection(char** mot, int* nb_total_mots, bool* redir_sortie, bool* stderr_sortie, bool*
15   redir_s_entree, int* old_entree, int* old_sortie, int* old_erreur, bool* redir, bool*
16   reset){
17
18   /* CRÉATION DES REDIRECTIONS */
19
20   if (*redir == true){
21
22     mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR | /* tout le monde */
23               S_IRGRP | S_IWGRP | S_IXGRP | /* peut lire, écrire */
24               S_IROTH | S_IWOTH | S_IXOTH ; /* et exécuter */
25
26     for (int i = 0; i < *nb_total_mots -1 ; i++) { /* on parcourt les mots */
27       /* redirection simple > */
28       simple_sortie (mot, nb_total_mots, mode, redir_sortie, i);
29       if (*redir_sortie)
30         continue;
31
32       /* redirection double >> */
33
34     }
35
36   }
37
38   /* fin du processus enfant */
39
40   /* RÉINITIALISATION STDIN, STDOUT, STDERR */
41   reset = true;
42   redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &
43   redir_s_entree, &old_entree, &old_sortie, &old_erreur,&redir, &reset);
44
45   /* si on n'attend pas le processus enfant,
46   ou que le processus enfant est déjà terminé on continue */
47   continue;
48 }

```

```

31     double_sortie (mot, nb_total_mots, redir_sortie, mode, i);
32     if (*redir_sortie)
33         continue;
34
35     /* redirection stderr simple sortie 2> */
36     stderr_simple_sortie(mot, nb_total_mots, stderr_sortie, mode, i);
37     if (*stderr_sortie)
38         continue;
39
40     /* redirection stderr double sortie 2>> */
41     stderr_double_sortie(mot, nb_total_mots, stderr_sortie, mode, i);
42     if (*stderr_sortie)
43         continue;
44
45     /* redirection stderr et stdout en sortie simple  >& */
46     stderr_stdout_sortie(mot, nb_total_mots, stderr_sortie, redir_sortie, mode, i);
47     if ((*stderr_sortie) && (*redir_sortie))
48         continue;
49
50     /* redirection simple < */
51     simple_entree (mot, nb_total_mots, redir_s_entree, mode, i);
52     if (*redir_s_entree)
53         continue;
54 }
55 *redir = false;
56 }
```

La deuxième partie de la fonction `redirection` concerne la réinitialisation des valeurs `stdin`, `stdout` et `stderr` après l'utilisation des redirections.

Pour cela, si la fonction est appelée alors que le booléen `reset` est vrai, la fonction va rétablir les valeurs d'origine, selon les autres booléens utilisés. Des fonctions correspondant à chaque cas de figure ont été préparées. Suite à la réinitialisation des valeurs, le booléen `reset` est passé à faux.

```

58     /* QUAND ON VEUT RÉINITIALISER LES ENTRÉES ET SORTIES */
59     else if (*reset == true) {
60
61         /* reset > et >> et >& stdout */
62         reset_sortie(redir_sortie, old_sortie);
63
64         /* reset 2> et 2>> et >&, stderr */
65         reset_stderr(stderr_sortie, old_erreur);
66
67         /* reset < , stdin */
68         reset_stdin(redir_s_entree, old_entree);
69
70         *reset = false;
71
72     return 0;
73 }
74 }
```

Pour chacune des redirections, nous présenterons la fonction correspondant à la création de la redirection, ainsi que la fonction correspondant à la réinitialisation des valeurs d'origine de `stdin`, `stdout` et `stderr`.

1.10.1 La redirection simple en sortie >

La création de la redirection

Cette commande permet de rediriger la sortie vers un fichier. Cette redirection va créer le fichier s'il n'existe pas et va écraser son contenu s'il existe déjà.

La création de la redirection est gérée par la fonction `simple_sortie`, appelée par `redirection`. Au sein de la fonction `simple_sortie`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot s'il est égal à `>`. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_redir_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection `>`, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT`, `O_TRUNC` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écrasement du contenu du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie (correspondant au fichier 1) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`. Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[i]=0` et `mot[i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`. Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis, nous passons le booléen correspondant à la redirection à vrai grâce à `*redir_sortie = true`.

```
78 /* CRÉATION DES REDIRECTIONS */
79
80 int simple_sortie (char** mot, int* nb_total_mots, mode_t mode, bool* redir_sortie, int i){
81
82     /* redirection simple en sortie > */
83     if (strcmp(mot[i], ">") == 0){
84         char* sortie = mot[i+1]; /* le fichier de sortie après le > */
85
86         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
87         /* write only, créé le fichier ou le remet à 0 si existe */
88         if (fd_redir_sortie < 0)
89             perror("fd_redir_sortie");
90
91         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */
92         if (t < 0)
93             perror("dup2 fd_redir_sortie");
94
95         mot[i] = 0; /* suppression du > */
96         (*nb_total_mots)-- ; /* mise à jour du nb de mots */
97         mot[i + 1] = 0; /* suppression du nom de fichier */
98         (*nb_total_mots)--; /* mise à jour du nb de mots */
99
100        close(fd_redir_sortie); /* fermeture */
101        *redir_sortie = true ;
102        /* passage à vrai du bool */
103    }
104
105 }
```

La réinitialisation des valeurs initiales

La réinitialisation des valeurs initiales est gérée par la fonction `reset_sortie`, appelée par `redirection`. Au sein de `reset_sortie`, nous cherchons si le booléen correspondant à `redir_sortie` est vrai. Si le booléen est vrai, alors on réinitialise `stdout` en utilisant la commande `dup2` et en utilisant le numéro de fichier faisant référence à la sortie originale, lequel a été initialisé au début du programme. Nous vérifions bien entendu la réussite de l'appel à la fonction.

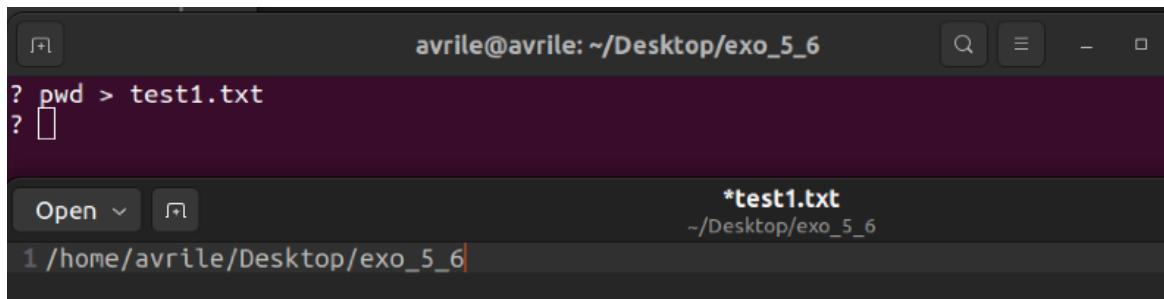
Finalement, le booléen qui était vrai est repassé à faux afin que le code puisse continuer de s'exécuter conve-

nablement s'il rencontre une autre instance de cette même redirection.

```
256 /* RÉINITIALISATION DES VALEURS INITIALES */  
257  
258 int reset_sortie(bool* redir_sortie, int* old_sortie){  
259     /* si > ou >> ou & (la sortie) */  
260     if (*redir_sortie == true) {  
261         int t = dup2(*old_sortie, 1); /* réinitialisation de stdout */  
262         if (t < 0)  
263             perror("dup old_sortie");  
264         else if (*redir_sortie)  
265             *redir_sortie = false ;  
266     }  
267     return 0;  
268 }
```

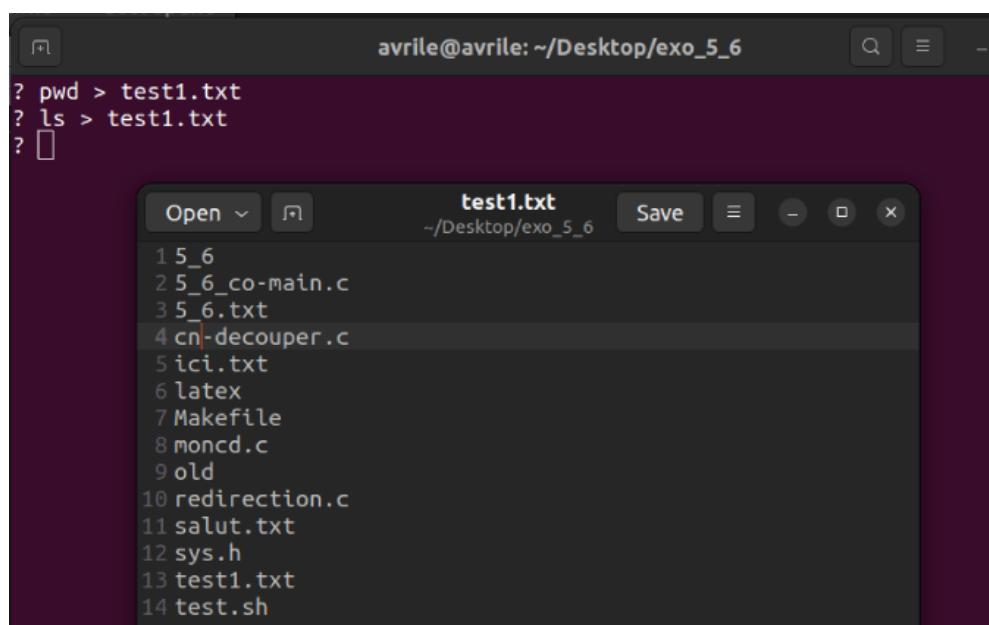
Les tests

Ci-après, voici des exemples d'utilisation de la redirection simple. Nous avons utilisé trois commandes différentes : `ls`, `pwd` et `echo`. Nous avons aussi illustré une erreur, lorsque la commande n'est pas reconnue. Dans ce cas-là, l'erreur apparaît sur le terminal car il ne s'agit pas de `stdout` mais de `stderr`.



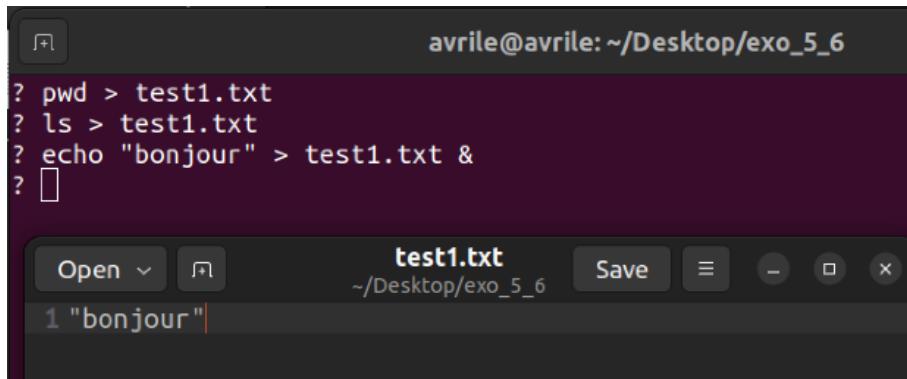
The terminal window title is "avrile@avrile: ~/Desktop/exo_5_6". The command entered is "? pwd > test1.txt". The terminal shows the command and its output, followed by a file browser window showing the contents of test1.txt. The file browser window title is "*test1.txt" and it shows the directory "/home/avrile/Desktop/exo_5_6".

FIGURE 14 – Nous redirigeons le résultat de la commande "pwd" dans le fichier "test1.txt".



The terminal window title is "avrile@avrile: ~/Desktop/exo_5_6". The commands entered are "? pwd > test1.txt", "? ls > test1.txt", and "? [empty]". The terminal shows the commands and their outputs. Below the terminal is a file browser window titled "test1.txt" showing the contents of the file, which includes the directory listing from the ls command.

FIGURE 15 – Nous redirigeons le résultat de la commande "ls" dans le fichier "test1.txt".



The terminal window shows the command history and the output of the command:

```
? pwd > test1.txt
? ls > test1.txt
? echo "bonjour" > test1.txt &
?
```

Below the terminal is a screenshot of a text editor window titled "test1.txt". The file contains the text "bonjour".

FIGURE 16 – Nous redirigeons le résultat de la commande echo "bonjour" dans le fichier "test1.txt".

```
? echo "bonjour" > test.txt
? echo "bonjour"
"bonjour"
?
```

FIGURE 17 – La réinitialisation de la sortie a fonctionné.

```
? echox "bonjour" > test1.txt
echox:notfound
?
```

FIGURE 18 – Nous tentons de rediriger le résultat de la commande "echox" dans le fichier "test1.txt" mais cela cause une erreur et donc la sortie s'affiche sur stderr.

1.10.2 La redirection double en sortie »

La création de la redirection

Cette commande permet de rediriger la sortie vers un fichier. Cette redirection va créer le fichier s'il n'existe pas et va ajouter le contenu à la fin du fichier s'il existe déjà. C'est la distinction principale avec la redirection simple en sortie. Ainsi, avec la redirection double en sortie, on ne supprime pas le contenu du fichier s'il existe déjà mais on écrit la suite à la fin du fichier.

La création de la redirection est gérée par la fonction `double_sortie`, appelée par `redirection`. Au sein de la fonction `double_sortie`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot s'il est égal à `>>`. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_redir_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection `>>`, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_APPEND` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'ajout du contenu à la fin du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie (correspondant au fichier 1) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`. Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[i]=0` et `mot[i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis, nous passons le booléen correspondant à la redirection à vrai grâce à `*redir_sortie = true`.

```
108 int double_sortie (char** mot, int* nb_total_mots, bool* redir_sortie, mode_t mode, int i){  
109  
110     /* redirection simple en sortie > */  
111     if (strcmp(mot[i], ">>") == 0){  
112  
113         char* sortie = mot[i+1]; /* le fichier de sortie après le > */  
114  
115         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode) ;  
116             /* write only, créé le fichier ou ajoute à la suite si existe */  
117             if (fd_redir_sortie < 0)  
118                 perror("fd_redir_sortie");  
119  
120         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */  
121         if (t < 0)  
122             perror("dup2 fd_redir_sortie");  
123  
124         mot[i] = 0; /* suppression du >> */  
125         (*nb_total_mots)--; /* mise à jour du nb de mots */  
126         mot[i + 1] = 0; /* suppression du nom de fichier */  
127         (*nb_total_mots)--; /* mise à jour du nb de mots */  
128  
129         close(fd_redir_sortie); /* fermeture */  
130         *redir_sortie = true ; /* passage à vrai du bool */  
131     }  
132     return 0 ;  
133 }
```

La réinitialisation des valeurs initiales

La réinitialisation des valeurs initiales est gérée par la fonction `reset_sortie`, appelée par `redirection` et que nous avons déjà présentée pour la fonction `simple_sortie`.

Au sein de `reset_sortie`, nous cherchons si le booléen correspondant à `redir_sortie` est vrai. Si le booléen est vrai, alors on réinitialise `stdout` en utilisant la commande `dup2` et en utilisant le numéro de fichier faisant référence à la sortie originale, lequel a été initialisé au début du programme. Nous vérifions bien entendu la

réussite de l'appel à la fonction.

Finalement, le booléen qui était vrai est repassé à faux afin que le code puisse continuer de s'exécuter convenablement s'il rencontre une autre instance de cette même redirection.

```
256 /* RÉINITIALISATION DES VALEURS INITIALES */  
257  
258 int reset_sortie(bool* redir_sortie, int* old_sortie){  
259     /* si > ou >> ou >& (la sortie)*/  
260     if (*redir_sortie == true) {  
261         int t = dup2(*old_sortie, 1); /* réinitialisation de stdout */  
262         if (t < 0)  
263             perror("dup old_sortie");  
264         else if (*redir_sortie)  
265             *redir_sortie = false ;  
266     }  
267     return 0;  
268 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection double en sortie. Nous avons utilisé trois commandes différentes : `ls`, `pwd` et `echo`. Nous avons aussi illustré une erreur, lorsque la commande n'est pas reconnue. Dans ce cas-là, l'erreur s'affiche sur `stderr` qui est le terminal. Nous constatons que les sorties des redirections successives sont bien ajoutées à la fin du fichier.

The screenshot shows a terminal window titled "avrile@avrile: ~/Desktop/exo_5_6". The user has run several commands to demonstrate redirection:

```
? ls >> test2.txt  
? pwd >> test2.txt &  
? echo "test2" >> test2.txt  
? echox "test2" >> test2.txt  
echox: notfound  
? echo "test2" >>  
"test2" >>  
? 
```

The output shows the contents of the file "test2.txt" at the bottom of the terminal window, which includes the outputs from all the redirection attempts.

FIGURE 19 – Nous redirigeons les commandes en sortie vers le fichier "test2.txt". Les sorties des commandes successives sont ajoutées à la fin du fichier.

```
? echo "bonjour" >> test.txt  
? echo "bonjour"  
"bonjour"  
?
```

FIGURE 20 – La réinitialisation de la sortie a fonctionné.

1.10.3 La redirection simple de stderr en sortie 2>

La création de la redirection

2> est une redirection de la sortie d'erreur `stderr` vers un fichier. Elle crée le fichier s'il n'existe pas et elle l'écrase s'il existe déjà.

La redirection simple de `stderr` en sortie est gérée par la fonction `stderr_simple_sortie`, appelée par `redirection`.

Au sein de la fonction `stderr_simple_sortie`, grâce à la fonction `strcmp`, nous vérifions, pour chaque mot, si l'est égal à `2>`. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant. Nous définissons également la variable `fd_stderr_s_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection `2>`, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_TRUNC` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écrasement du contenu du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie d'erreur standard `stderr` (correspondant au fichier 2) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`. Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[i]=0` et `mot[i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous passons le booléen correspondant à la redirection à vrai grâce à `*stderr_sortie = true`.

```
136 int stderr_simple_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
137     int i){
138 
139     /* redirection simple erreur en sortie 2> */
140     if (strcmp(mot[i], "2>") == 0){
141         char* sortie = mot[i+1]; /* le fichier de sortie après le 2> */
142 
143         int fd_stderr_s_redir = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
144             /* write only, créé le fichier ou le remet à 0 si existe */
145         if (fd_stderr_s_redir < 0)
146             perror("fd_stderr_s_redir");
147 
148         int t = dup2(fd_stderr_s_redir, 2); /* stderr redirigée vers l'arg du 2> */
149         if (t < 0)
150             perror("dup2 fd_stderr_sortie");
151 
152         mot[i] = 0;
153         (*nb_total_mots)--;
154         mot[i+1]=0;
155         (*nb_total_mots)--;
156 
157         close(fd_stderr_s_redir);
158         *stderr_sortie = true ;
159     }
160 }
161 }
```

La réinitialisation des valeurs initiales

La réinitialisation des valeurs initiales est gérée par la fonction `reset_stderr`, appelée par `redirection`.

Au sein de la fonction `reset_stderr`, appelée par `redirection`, nous cherchons si le booléen correspondant à `stderr_sortie` est vrai. Si c'est le cas, on réinitialise `stderr` en utilisant la commande `dup2` et en utilisant le numéro de fichier faisant référence à la sortie standard d'erreur originale, lequel a été initialisé au début du programme. Nous vérifions bien entendu la réussite de l'appel à la fonction.

Finalement, le booléen qui était vrai est repassé à faux afin que le code puisse continuer de s'exécuter convenablement s'il rencontre une autre instance de cette même redirection.

```
260 int reset_stderr(bool* stderr_sortie, int* old_erreur){  
261     /* si 2> ou 2>> ou >& (stderr) */  
262     if (*stderr_sortie == true) {  
263         int t = dup2(*old_erreur, 2); /* réinitialisation de stderr */  
264         if (t < 0)  
265             perror("dup old_sortie");  
266         else if (*stderr_sortie)  
267             *stderr_sortie = false ;  
268     }  
269     return 0;  
270 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection simple de stderr en sortie. Nous avons utilisé trois commandes différentes : `lss`, `echox` et `commande_inconnue`. Nous avons aussi illustré une commande qui n'inscrit rien dans le fichier puisqu'elle ne cause pas d'erreur, c'est la commande `ls`.
Nous constatons que lors de chaque redirection, le contenu précédent du fichier est écrasé.

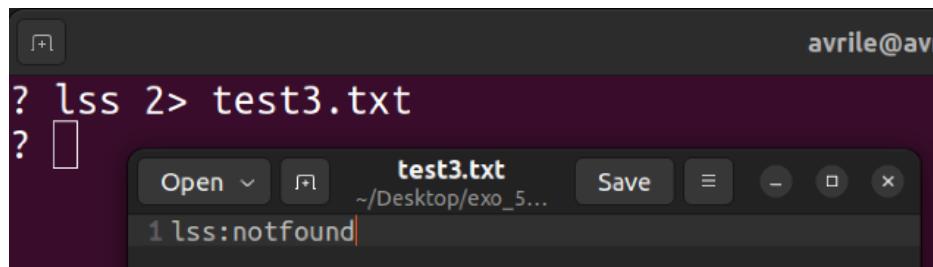


FIGURE 21 – Nous redirigeons stderr en sortie pour la commande "lss".

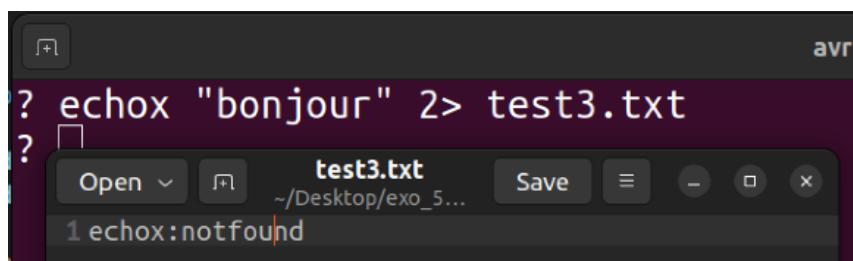


FIGURE 22 – Nous redirigeons stderr en sortie pour la commande "echox".

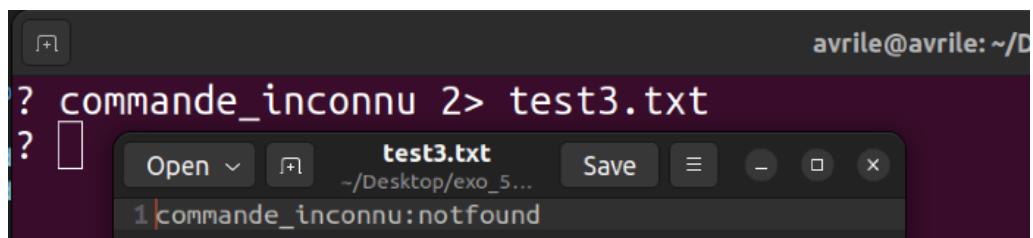
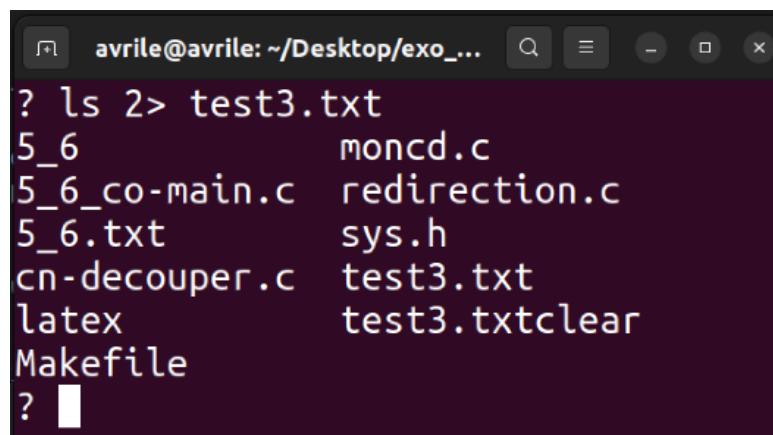


FIGURE 23 – Nous redirigeons stderr en sortie pour la commande "commande_inconnue".

```
? echox "bonjour" 2> test.txt  
? echox "bonjour"  
echox:notfound  
?
```

FIGURE 24 – La réinitialisation de la sortie d'erreur a fonctionné.



The screenshot shows a terminal window with the following details:

- Terminal title: `avrile@avrile: ~/Desktop/exo_...`
- Content:

```
? ls 2> test3.txt  
5_6          moncd.c  
5_6_co-main.c redirection.c  
5_6.txt      sys.h  
cn-decouper.c test3.txt  
latex        test3.txtclear  
Makefile  
?
```

FIGURE 25 – Nous redirigeons stderr en sortie pour la commande "ls", néanmoins cette commande ne cause pas d'erreur et le résultat de la commande s'affiche sur le terminal (qui correspond à stdout).

1.10.4 La redirection double de `stderr` en sortie 2»

La création de la redirection

`2>>` est une redirection de la sortie d'erreur `stderr` vers un fichier. Elle crée le fichier s'il n'existe pas et elle écrit à la fin s'il existe déjà. C'est la distinction principale avec la redirection simple de `stderr` en sortie.

La redirection double de `stderr` en sortie est gérée par la fonction `stderr_double_sortie`, appelée par `redirection`.

Au sein de la fonction `stderr_double_sortie`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot s'il est égal à `2>>`. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_stderr_d_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection `2>>`, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_APPEND` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écriture à la fin du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie d'erreur standard `stderr` (correspondant au fichier 2) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`.

Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[i]=0` et `mot[i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous passons le booléen correspondant à la redirection à vrai grâce à `*stderr_sortie = true`.

```
163 int stderr_double_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
164     int i){
165     /* redirection double erreur en sortie 2>> */
166     if (strcmp(mot[i], "2>>") == 0){
167         char* sortie = mot[i+1]; /* le fichier de sortie après le 2>> */
168
169         int fd_stderr_d_redir = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode) ;
170         /* write only, créé le fichier ou ajoute à la fin si existe */
171         if (fd_stderr_d_redir < 0)
172             perror("fd_stderr_d_redir");
173
174         int t = dup2(fd_stderr_d_redir, 2); /* stderr redirigée vers l'arg du 2>> */
175         if (t < 0)
176             perror("dup2 fd_stderr_sortie");
177
178         mot[i] = 0;
179         (*nb_total_mots)--;
180         mot[i+1]=0;
181         (*nb_total_mots)--;
182
183         close(fd_stderr_d_redir);
184         *stderr_sortie = true ;
185     }
186     return 0;
187 }
```

La réinitialisation des valeurs initiales

La réinitialisation des valeurs initiales est gérée par la fonction `reset_stderr`, appelée par `redirection`.

Au sein de la fonction `reset_stderr`, appelée par `redirection`, nous cherchons si le booléen correspondant à `stderr_sortie` est vrai. Si c'est le cas, on réinitialise `stderr` en utilisant la commande `dup2` et en utilisant le numéro de fichier faisant référence à la sortie standard d'erreur originale, lequel a été initialisé au début du

programme. Nous vérifions bien entendu la réussite de l'appel à la fonction.

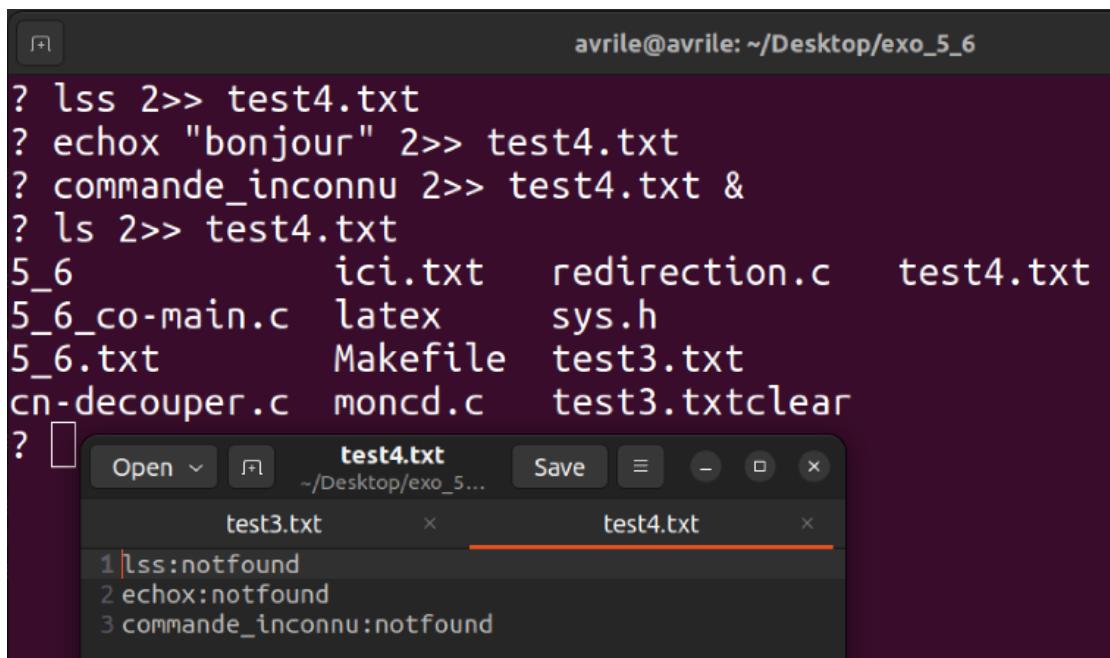
Finalement, le booléen qui était vrai est repassé à faux afin que le code puisse continuer de s'exécuter convenablement s'il rencontre une autre instance de cette même redirection.

```
260 int reset_stderr(bool* stderr_sortie, int* old_erreur){  
261     /* si 2> ou 2>> ou >& (stderr) */  
262     if (*stderr_sortie == true) {  
263         int t = dup2(*old_erreur, 2); /* réinitialisation de stderr */  
264         if (t < 0)  
265             perror("dup old_sortie");  
266         else if (*stderr_sortie)  
267             *stderr_sortie = false ;  
268     }  
269     return 0;  
270 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection double de stderr en sortie. Nous avons utilisé trois commandes différentes : `lss`, `echox` et `commande_inconnue`. Nous avons aussi illustré une commande qui n'inscrit rien dans le fichier puisqu'elle ne cause pas d'erreur, c'est la commande `ls`.

Nous constatons que, lors de chaque redirection, le contenu précédent du fichier est ajouté à la fin du fichier.



The terminal window shows the following command sequence:

```
? lss 2>> test4.txt  
? echox "bonjour" 2>> test4.txt  
? commande_inconnu 2>> test4.txt &  
? ls 2>> test4.txt
```

The output in the terminal is:

```
5_6           ici.txt    redirection.c    test4.txt  
5_6_co-main.c  latex      sys.h  
5_6.txt        Makefile    test3.txt  
cn-decouper.c   moncd.c    test3.txtclear
```

A file browser window titled "test4.txt" is open, showing the contents of the file:

```
test3.txt      test4.txt  
1 lss:notfound  
2 echox:notfound  
3 commande_inconnu:notfound
```

FIGURE 26 – Nous redirigeons stderr en sortie vers le fichier "test4.txt". Les sorties successives de stderr sont ajoutées à la fin du fichier.

```
? echox "bonjour" 2>> test.txt  
? echox "bonjour"  
echox:notfound  
?
```

FIGURE 27 – La réinitialisation de la sortie d'erreur a fonctionné.

1.10.5 La redirection de la sortie et de la sortie d'erreur vers un fichier >&

La création de la redirection

La redirection >&, qui est l'une des graphies permettant d'arriver à ce résultat (source : https://www.gnu.org/software/bash/manual/html_node/Redirections.html), redirige à la fois la sortie standard et la sortie d'erreur vers un fichier. Les deux flux sont fusionnés et traités de la même manière. Cette redirection va créer le fichier s'il n'existe pas et va écraser son contenu s'il existe déjà.

Nous définissons la redirection de la sortie et de la sortie d'erreur vers un fichier grâce à la fonction `stderr_stdout_sortie`, appelée par `redirection`.

Au sein de `stderr_stdout_sortie`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot, s'il est égal à >&. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant. Nous définissons également la variable `fd_s_sortie_and`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie.

Afin de respecter les spécifications de la redirection >&, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_TRUNC` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écrasement du contenu du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie d'erreur (correspondant au fichier 2) vers notre fichier. Nous effectuons la même opération avec la sortie `stdout` qui correspond au fichier 1. En effet, dans le cadre de cette redirection, à la fois la sortie standard et la sortie d'erreur sont redirigées vers le même fichier. Ensuite, nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`.

Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[i]=0` et `mot[i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous passons les booléens correspondants à la redirection à vrai grâce à `*stderr_sortie = true` et `*redir_sortie = true`.

```
189 int stderr_stdout_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, bool*
190     redir_sortie, mode_t mode, int i){
191
192     if (strcmp(mot[i], ">&") == 0){
193         char* sortie = mot[i+1]; /* le fichier de sortie après le > */
194
195         int fd_s_sortie_and = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
196             /* write only, créé le fichier ou le remet à 0 si existe */
197         if (fd_s_sortie_and < 0)
198             perror("fd_s_sortie_and");
199
200         int t = dup2(fd_s_sortie_and, 1); /* redir stout vers le fichier */
201         if (t < 0)
202             perror("dup2 stout fd_s_sortie_and");
203
204         int tt = dup2(fd_s_sortie_and, 2); /* redir stderr vers le fichier */
205         if (tt < 0)
206             perror("dup2 stderr fd_s_sortie_and");
207
208         mot[i] = 0; /* suppression du > */
209         (*nb_total_mots)--; /* mise à jour du nb de mots */
210         mot[i + 1] = 0; /* suppression du nom de fichier */
211         (*nb_total_mots)--; /* mise à jour du nb de mots */
212
213         close(fd_s_sortie_and); /* fermeture */
214         *stderr_sortie = true ; /* passage à vrai du bool */
215         *redir_sortie = true ; /* passage à vrai du bool */
216     }
217
218     return 0;
219 }
```

La réinitialisation des valeurs initiales

Afin de réinitialiser les sorties standard et d'erreur, nous allons utiliser deux fonctions : `reset_sortie` (pour la réinitialisation de `stdout`) et `reset_stderr` (pour la réinitialisation de `stderr`).

Au sein de la fonction `reset_sortie`, appelée par `redirection`, nous cherchons si le booléen correspondant à `redir_sortie` est vrai. Si c'est le cas, on réinitialise `stdout` en utilisant la commande `dup2` et en utilisant le numéro de fichier faisant référence à la sortie standard originale, lequel a été initialisé au début du programme. Nous vérifions bien entendu la réussite de l'appel à la fonction.
Finalement, le booléen qui était vrai est repassé à faux afin que le code puisse continuer de s'exécuter convenablement s'il rencontre une autre instance de cette même redirection.

```
246 /* RÉINITIALISATION DES VALEURS INITIALES */
247
248 int reset_sortie(bool* redir_sortie, int* old_sortie){
249     /* si > ou >> ou >& (la sortie)*/
250     if (*redir_sortie == true) {
251         int t = dup2(*old_sortie, 1); /* réinitialisation de stdout */
252         if (t < 0)
253             perror("dup old_sortie");
254         else if (*redir_sortie)
255             *redir_sortie = false ;
256     }
257     return 0;
258 }
```

Au sein de la fonction `reset_stderr`, appelée par `redirection`, nous cherchons si le booléen correspondant à `stderr_sortie` est vrai. Si c'est le cas, on réinitialise `stderr` en utilisant la commande `dup2` et en utilisant le numéro de fichier faisant référence à la sortie standard d'erreur originale, lequel a été initialisé au début du programme. Nous vérifions bien entendu la réussite de l'appel à la fonction.

Finalement, le booléen qui était vrai est repassé à faux afin que le code puisse continuer de s'exécuter convenablement s'il rencontre une autre instance de cette même redirection.

```
260 int reset_stderr(bool* stderr_sortie, int* old_erreur){
261     /* si 2> ou 2>> ou >& (stderr)*/
262     if (*stderr_sortie == true) {
263         int t = dup2(*old_erreur, 2); /* réinitialisation de stderr */
264         if (t < 0)
265             perror("dup old_sortie");
266         else if (*stderr_sortie)
267             *stderr_sortie = false ;
268     }
269     return 0;
270 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection des sorties standard et d'erreur vers un fichier (`>&`).

Nous avons utilisé trois commandes différentes : `echo`, `echox` et `ls`. Nous avons aussi illustré une commande qui ne fonctionne pas, lorsque l'on écrit la redirection `> &`, c'est-à-dire avec un espace. Il convient d'écrire la commande de redirection sans espace.

Nous constatons que, lors de chaque redirection, le contenu du fichier est écrasé (dans le cas où le fichier existait déjà).

```
avrile@avrile: ~/Desktop/exo_5...$ echo "x" >& test5.txt
? 
```

FIGURE 28 – Nous redirigeons les sorties standard et d’erreur vers un fichier. Nous illustrons avec le passage d’une commande “echo”.

```
avrile@avrile: ~/Desktop/exo_5...$ echo "x" >& test5.txt
? echo "x" >& test5.txt
? echox "x" >& test5.txt
? 
```

FIGURE 29 – Nous redirigeons les sorties standard et d’erreur vers un fichier. Nous illustrons avec le passage d’une commande “echox” (qui cause une erreur).

```
avrile@avrile: ~/Desktop/exo_5...$ echo "x" >& test5.txt
? echo "x" >& test5.txt
? echox "x" >& test5.txt
? ls >& test5.txt
? 
```

Line Number	File Name
1	5_6
2	5_6_co-main.c
3	5_6.txt
4	cn-decouper.c
5	ici.txt
6	latex
7	Makefile
8	moncd.c
9	redirection.c
10	sys.h
11	test3.txt
12	test3.txtclear
13	test4.txt
14	test5.txt

FIGURE 30 – Nous redirigeons les sorties standard et d’erreur vers un fichier. Nous illustrons avec le passage d’une commande “ls”.

```
? echox "bonjour" >& test.txt
? echo "bonjour" >& test.txt
? echo "bonjour"
"bonjour"
? echox "bonjour"
echox:notfound
?
```

FIGURE 31 – La réinitialisation de la sortie standard et de la sortie d'erreur a fonctionné.

```
? echoxx > & test5.txt
echoxx:notfound
?
```

FIGURE 32 – Nous redirigeons les sorties standard et d'erreur vers un fichier. Nous illustrons avec le passage d'une commande "echox" qui cause une erreur. En revanche, nous n'avons pas bien orthographié la commande de redirection car nous y avons ajouté un espace. Ça ne marche pas.

1.10.6 La redirection de l'entrée simple <

La création de la redirection

La redirection < redirige l'entrée standard (`stdin`) d'une commande depuis un fichier plutôt que depuis le clavier.

Nous définissons la redirection de l'entrée vers un fichier grâce à la fonction `simple_entree`, appelée par `redirection`. Au sein de `simple_entree`, grâce à la fonction `strcmp`, nous vérifions pour chacun des mots s'il est égal à <. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_redir_s_entree`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection <, nous avons ajouté le drapeau `O_RDONLY` (qui correspond à la lecture du fichier). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger l'entrée standard vers notre fichier. Ensuite, nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`.

Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[i]=0` et `mot[i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous passons le booléen correspondant à la redirection à vrai grâce à `*redir_s_entree = true`.

```
220 int simple_entree (char** mot, int* nb_total_mots, bool* redir_s_entree, mode_t mode, int i){  
221  
222     /* redirection simple en entrée < */  
223     if (strcmp(mot[i], "<") == 0){  
224         char* entree = mot[i+1]; /* le fichier entrée après le < */  
225  
226         int fd_redir_s_entree = open(entree, O_RDONLY, mode) ;  
227             /* read only */  
228         if (fd_redir_s_entree < 0)  
229             perror("fd_redir_s_entree");  
230  
231         int t = dup2(fd_redir_s_entree, 0); /* entrée redirigée vers l'arg du < */  
232         if (t < 0)  
233             perror("dup2 fd_redir_s_entree");  
234         else {  
235             mot[i] = 0; /* suppression du < */  
236             (*nb_total_mots)--;  
237             mot[i + 1] = 0; /* suppression du nom de fichier */  
238             (*nb_total_mots)--;  
239             close(fd_redir_s_entree);  
240             *redir_s_entree = true ; /* passage à vrai du bool */  
241         }  
242     }  
243     return 0;  
244 }
```

La réinitialisation des valeurs initiales

Afin de réinitialiser l'entrée `stdin`, nous utilisons la fonction `reset_stdin`, appelée par `redirection`. Au sein de `reset_stdin`, nous cherchons si le booléen correspondant à `redir_s_entree` est vrai.

Si c'est le cas, alors on réinitialise `stdin` en utilisant la commande `dup2` et en utilisant le numéro de fichier faisant référence à l'entrée `stdin` originale, lequel a été initialisé au début du programme. Nous vérifions bien entendu la réussite de l'appel à la fonction.

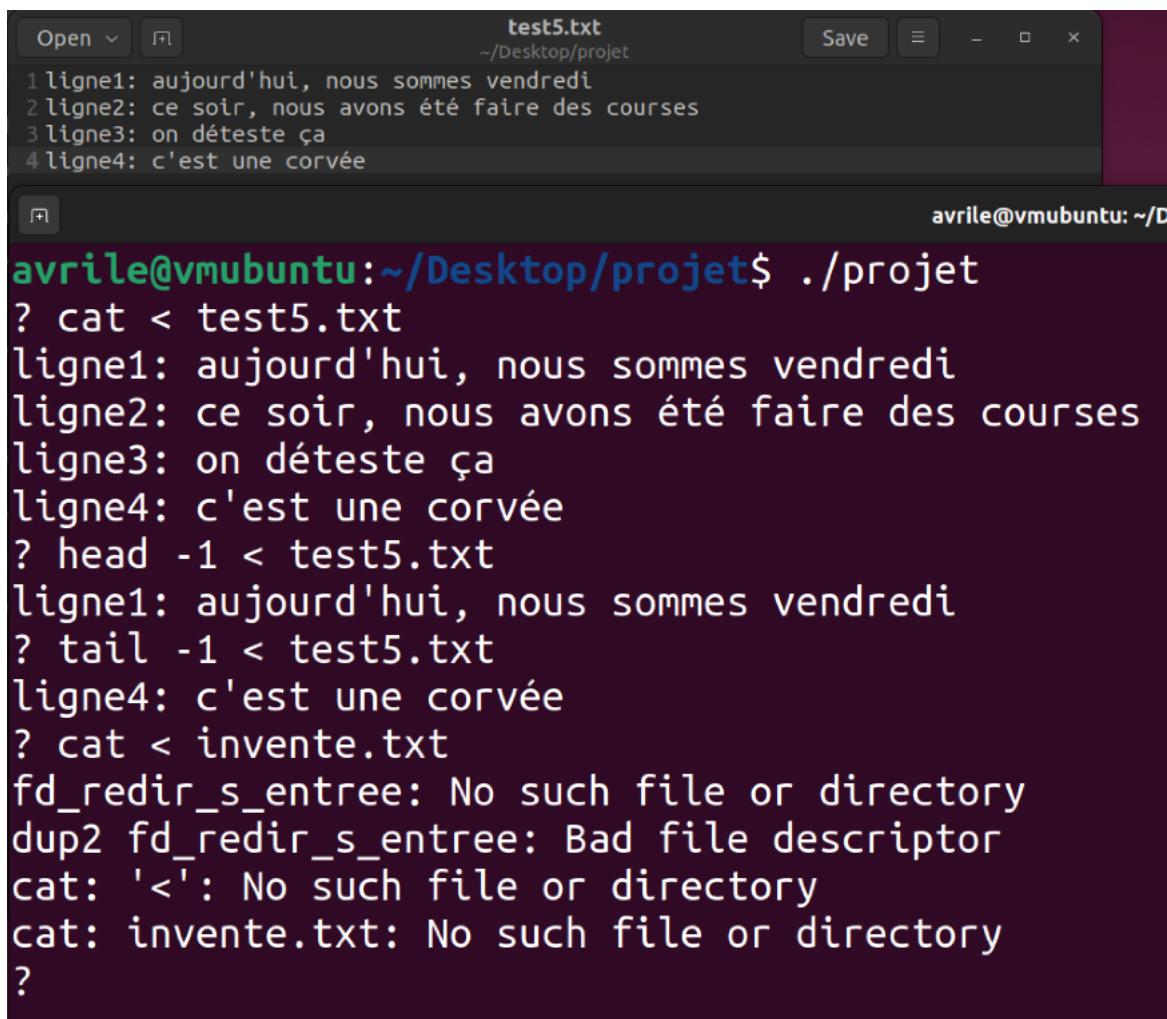
Finalement, le booléen qui était vrai est repassé à faux afin que le code puisse continuer de s'exécuter convenablement s'il rencontre une autre instance de cette même redirection.

```

273 int reset_stdin(bool* redir_s_entree, int* old_entree){
274     /* si < */
275     if (*redir_s_entree == true){
276         int t = dup2(*old_entree,0); /* réinit de stdin */
277         if (t < 0)
278             perror("dup old_entree");
279         *redir_s_entree = false ;
280     }
281     return 0;
282 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection de l'entrée simple. Nous avons utilisé des commandes différentes : `cat`, `head` et `tail`. Nous avons aussi illustré une commande qui ne fonctionne pas, lorsque l'on tente de rediriger l'entrée vers un fichier qui n'existe pas.



The screenshot shows a terminal window with a dark background. At the top, there is a file viewer window titled "test5.txt" showing four lines of text: "ligne1: aujourd'hui, nous sommes vendredi", "ligne2: ce soir, nous avons été faire des courses", "ligne3: on déteste ça", and "ligne4: c'est une corvée". Below this, the terminal prompt is "avrile@vmubuntu: ~/Desktop/projet\$". The user then enters several commands:

- ? cat < test5.txt
- ligne1: aujourd'hui, nous sommes vendredi
- ligne2: ce soir, nous avons été faire des courses
- ligne3: on déteste ça
- ligne4: c'est une corvée
- ? head -1 < test5.txt
- ligne1: aujourd'hui, nous sommes vendredi
- ? tail -1 < test5.txt
- ligne4: c'est une corvée
- ? cat < invente.txt
- fd_redir_s_entree: No such file or directory
- dup2 fd_redir_s_entree: Bad file descriptor
- cat: '<': No such file or directory
- cat: invente.txt: No such file or directory
- ?

FIGURE 33 – Nous redirigeons l'entrée standard vers le fichier "test5.txt" et nous appelons les commandes "cat", "tail", et "head". Si le fichier n'existe pas, cela cause une erreur.

Conclusion pour les redirections d'entrées-sorties : La fonction `redirection`, qui se trouve au sein du fichier `redirection.c`, est la fonction en charge de la gestion des redirections. Elle est appelée par la fonction `main`. En son sein, elle est divisée en sous-fonctions qui correspondent à la création des redirections et aux réinitialisations postérieures des entrées-sorties.

Nous avons présenté six redirections : `>`, `>>`, `2>`, `2>>`, `>&` et `<`. Pour chaque redirection, nous avons présenté les fonctions particulières (pour la création de la redirection et la réinitialisation des entrées-sorties) et nous avons illustré par trois exemples et une erreur.

1.11 Les redirections d'entrées-sorties au sein des pipes

Afin de pouvoir utiliser les redirections d'entrées-sorties avec les pipes, nous avons dû modifier notre fichier `redirection.c`. Pour ce faire, nous avons créé un nouveau fichier `redir_in_pipe.c`. En effet, pour l'implémentation des pipes, nous avons dû modifier notre tableau de chaînes de caractères pour le rendre bidimensionnel. Chaque ligne du tableau correspond au contenu d'un pipe. Ainsi, pour pouvoir continuer à utiliser les redirections, nous avons dû adapter nos fonctions à ce nouvel état de fait, c'est-à-dire à la transformation de notre tableau de mots en un tableau de mots bidimensionnel.

L'organisation du fichier `redir_in_pipe.c` est similaire à l'organisation du fichier `redirection.c`. C'est-à-dire qu'une fonction principale `redirection_dans_pipe` gère la création des redirections. La fonction est simplifiée car elle ne doit s'occuper que de la création des redirections : elle n'a pas besoin de gérer les réinitialisations des valeurs initiales d'entrées-sorties. Il s'agit d'une différence avec le comportement de la fonction dans le fichier `redirection.c`. Ici, comme les redirections sont effectuées au sein d'un processus enfant, il n'est pas nécessaire de réinitialiser les valeurs initiales d'entrées-sorties.

Dans la fonction `main` (fichier `projet_co-main.c`), nous initialisons le booléen `ilya_pipe` à faux. Ensuite, dans la boucle du prompt, on met à jour le booléen `ilya_pipe` grâce à la fonction `cb_pipe`. La fonction `redirection` ne sera exécutée que si le booléen `ilya_pipe` est faux. Si le booléen `ilya_pipe` est vrai, les redirections seront traitées à l'intérieur des fonctions dédiées aux pipes. Ainsi, les redirections se passeront alors à l'intérieur du processus enfant.

```
27 int
28 main(int argc, char * argv[]){
29
30     bool ilya_pipe = false ;
31
32     /* BOUCLE DU PROMPT */
33     while((ligne = readline(PROMPT)) != NULL){
34
35         /* BOUCLE DU PROMPT */
36         while((ligne = readline(PROMPT)) != NULL){
37
37             int nb_pipe_total = cb_pipe(mot, nb_total_mots, &ilya_pipe);
38             /* renvoie nb total de pipes | + permet passage ilya_pipe à vrai */
39
40             /* LES REDIRECTIONS */
41             if (!ilya_pipe){
42                 redir = true ;
43                 redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &redir_s_entree, &
44                 old_entree, &old_sortie, &old_erreur,&redir, &reset);
45             }
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74 }
```

Dans le fichier `pipe.c`, au sein des fonctions dédiées à la gestion des pipes (`un_pipe` et `deux_pipes`), la fonction `redirection_dans_pipe` est appelée après le `fork()`, à l'intérieur du processus enfant et avant `execv`.

```
79 int un_pipe(int mes_pipes[][2],char pathname[], int pathname_size, char* dirs[], char*
80     mots_pipe[] [MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe){
81
82     int tmp, tmp2;
83
84     if (pipe(mes_pipes[0]) < 0){ /* vérification erreur pipe */
85         perror("pipe");
86         return -1; }
87
88     /* PROCESSUS PARENT: FORK1 */
89     if ((tmp= fork())<0){
90         perror("fork");
91         return -1; }
92
93     /* PROCESSUS ENFANT 1 */
94     if (tmp ==0) {
95         dup2(mes_pipes[0][1], 1); /* redirection sortie vers pipe */
96         close(mes_pipes[0][0]); /* fermeture pipe */
97         close(mes_pipes[0][1]);
```

```

98     int nb_elt_tab1 = nb_mots(mots_pipe, 0); /* compte les mots du pipe */
99
100    redirection_dans_pipe(mots_pipe, 0);
101
102    if (strcmp(mots_pipe[0][0], "monman") == 0){ /*si monman*/
103        monman(mots_pipe[0]);
104        exit(0); /* sinon on reste bloqué on attend */
105    }
106
107    if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
108        appel_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
109        exit(0); /* sinon on reste bloqué on attend */
110    }
111
112    if (strcmp(mots_pipe[0][0], "monexit") == 0){ /* si monexit*/
113        mon_exit_pipe(mots_pipe[0]);
114        exit(0); /* sinon on reste bloqué on attend */
115    }
116
117    else{
118        for(int i = 0; dirs[i] != 0; i++){
119            sprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
120            execv(pathname, mots_pipe[0]);
121        }

```

La fonction `redirection_dans_pipe` se trouve dans le fichier `pipe.c`. Elle prend pour argument le tableau bidimensionnel `char* mots_pipe[][] [MaxMots]` et l'int `ligne`, qui permet de préciser le sous-tableau dans lequel on se trouve.

C'est la fonction centralisatrice des redirections. Elle compare le contenu du sous-tableau bidimensionnel (qui correspond à la commande qui va ensuite être exécutée par `execv`) pour voir s'il contient des redirections.

Nous parcourons les mots du sous-tableau tant que les mots sont différents de NULL. Pour chacune des fonctions de redirection, nous utilisons également la `ligne` (correspondant au sous-tableau) en argument.

La principale différence entre ces fonctions et les fonctions de la partie précédente est que ces fonctions sont programmées pour fonctionner avec un tableau bidimensionnel. En outre, comme nous n'avons pas besoin, en l'espèce, de réinitialiser les entrées-sorties, nous avons pu considérablement simplifier la fonction.

La fonction `redirection_dans_pipe` parcourt tous les mots d'une ligne d'un tableau et teste, pour chaque mot, l'ensemble des fonctions de redirection. Si l'une des fonctions est exécutée, alors elle renvoie 2 et cela a pour conséquence de ne pas exécuter la suite du code (`continue`).

Les modes sont définis de manière à ce que tout le monde puisse lire, écrire et exécuter.

```

1  /* ****
2 # Nom ..... : redir_in_pipe.c
3 # Rôle ..... : Fonction de gestion des redirections dans les pipes:
4 #           >, >>, 2>, 2>>, >&, <
5 # Auteur ..... : Avrile Floro
6 # Version ..... : V0.1 du 28/10/2023
7 # Licence ..... : réalisé dans le cadre du cours de SE
8 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
9 # ****
10
11 #include "sys.h"
12 #include "f_head.h"
13
14 int redirection_dans_pipe(char* mots_pipe[][] [MaxMot], int ligne){
15
16 /* GESTION DES REDIRECTIONS: CRÉATION */
17
18     mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR | /* tout le monde */
19                 S_IRGRP | S_IWGRP | S_IXGRP | /* peut lire, écrire */
20                 S_IROTH | S_IWOTH | S_IXOTH ; /* et exécuter */
21
22     for (int i = 0; mots_pipe[ligne][i] != NULL ; i++) { /* on parcourt les mots */
23         /* redirection simple > */
24
25         if (simple_sortie_pipe (mots_pipe, ligne, mode, i) == 2)
26             continue;

```

```

27     /* redirection double >> */
28     if (double_sortie_pipe (mots_pipe, ligne, mode, i) == 2)
29         continue;
30
31     /* redirection stderr simple sortie 2> */
32     if (stderr_simple_sortie_pipe(mots_pipe, ligne, mode, i) == 2)
33         continue;
34
35     /* redirection stderr double sortie 2>> */
36     if (stderr_double_sortie_pipe(mots_pipe, ligne, mode, i) == 2)
37         continue;
38
39     /* redirection stderr et stdout en sortie simple  >& */
40     if (stderr_stdout_sortie_pipe(mots_pipe, ligne, mode, i) == 2)
41         continue;
42
43     /* redirection simple < */
44     if (simple_entree_pipe (mots_pipe, ligne, mode, i) == 2)
45         continue;
46
47 }
48
49 return 0;
}

```

1.11.1 La redirection simple en sortie > avec les pipes

La création de la redirection

Cette commande permet de rediriger la sortie vers un fichier. Cette redirection va créer le fichier s'il n'existe pas et va écraser son contenu s'il existe déjà.

La création de la redirection est gérée par la fonction `simple_sortie_pipe`, appelée par `redirection_dans_pipe`. Au sein de la fonction `simple_sortie_pipe`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot de la ligne s'il est égal à `>`. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_redir_s_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection `>`, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_TRUNC` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écrasement du contenu du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie (correspondant au fichier 1) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`. Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[ligne][i]=0` et `mot[ligne][i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`. Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Nous retournons 2.

```

54 int simple_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){
55
56     /* redirection simple en sortie > */
57     if (strcmp(mots_pipe[ligne][i], ">") == 0){
58         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le > */
59
60         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
61             /* write only, créé le fichier ou le remet à 0 si existe */
62         if (fd_redir_sortie < 0)
63             perror("fd_redir_sortie");
64
65         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */
66         if (t < 0)
67             perror("dup2 fd_redir_sortie");
68
69         mots_pipe[ligne][i] = 0; /* suppression du > */

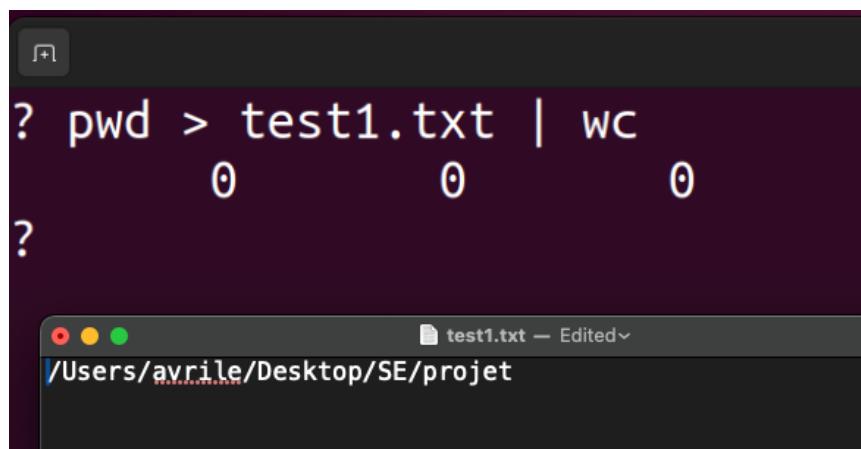
```

```

70     mots_pipe[ligne][i + 1] = 0; /* suppression du nom de fichier */
71
72     close(fd_redir_sortie); /* fermeture */
73     return 2; /* signe continue dans gestion_redirection */
74 }
75 return 0 ;
76 }
```

Les tests

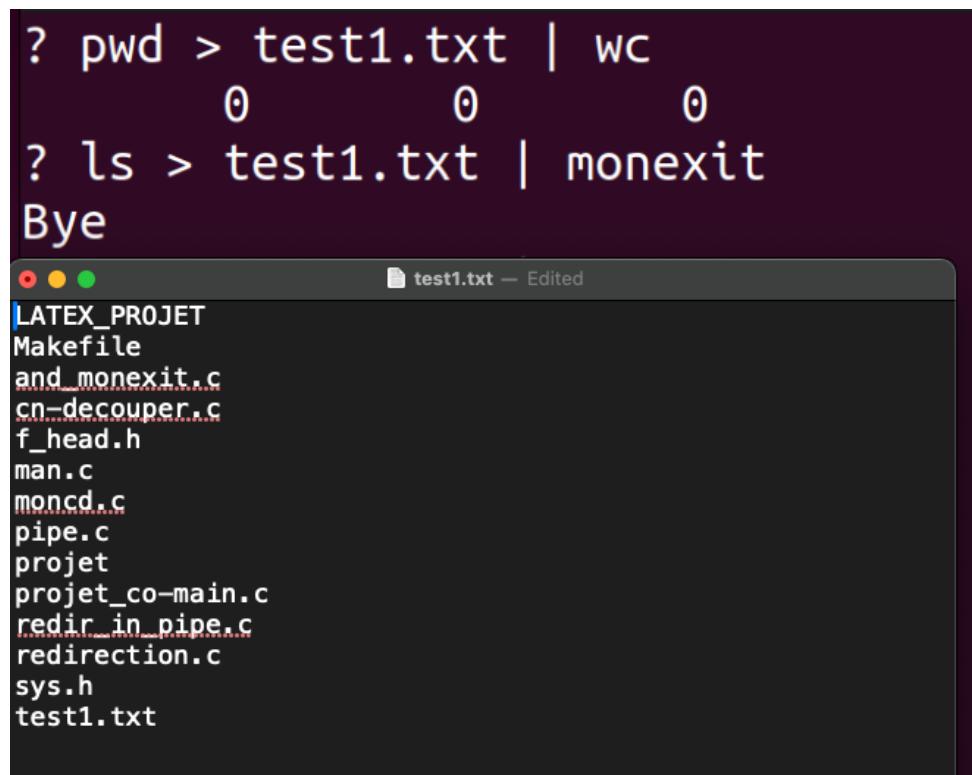
Nous avons effectué des tests pour illustrer le fonctionnement de notre redirection avec les pipes. Nous avons utilisé les commandes `pwd`, `ls` et `wc`, que nous avons redirigées avec succès. Enfin, nous avons illustré une erreur avec la commande `echo`.



```
? pwd > test1.txt | wc
      0      0      0
?

? 
```

FIGURE 34 – Au sein d'un pipe, nous redirigeons la sortie de la commande "pwd" au sein du fichier test1.txt.



```
? pwd > test1.txt | wc
      0      0      0
? ls > test1.txt | monexit
Bye

? 
```

LATEX_PROJECT
Makefile
and_monexit.c
cn-decouper.c
f_head.h
man.c
moncd.c
pipe.c
projet
projet_co-main.c
redir_in_pipe.c
redirection.c
sys.h
test1.txt

FIGURE 35 – Au sein d'un pipe, nous redirigeons la sortie de la commande "ls" au sein du fichier test1.txt.

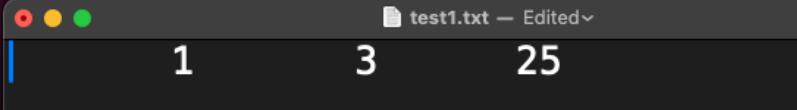
```
? ls | wc | wc > test1.txt  
?   
? 1 3 25
```

FIGURE 36 – Au sein d'un pipe, nous redirigeons la sortie de la commande "wc" au sein du fichier test1.txt.

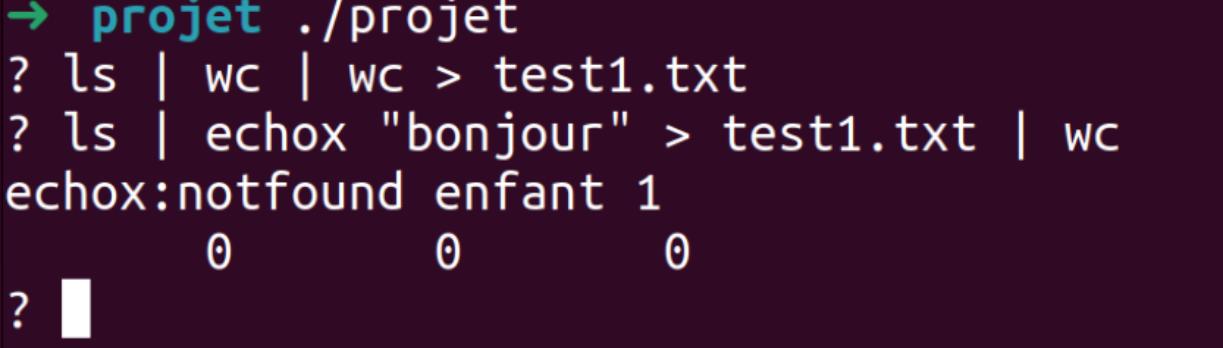
```
→ projet ./projet  
? ls | wc | wc > test1.txt  
? ls | echox "bonjour" > test1.txt | wc  
echox: not found enfant 1  
0 0 0  
? 
```

FIGURE 37 – La redirection de la sortie de "echox" vers le fichier test1.txt n'a pas fonctionné. On obtient en sortie une erreur qui s'affiche sur le terminal (stderr).

1.11.2 La redirection double en sortie » avec les pipes

La création de la redirection

Cette commande permet de rediriger la sortie vers un fichier. Cette redirection va créer le fichier s'il n'existe pas et va ajouter le contenu à la fin du fichier s'il existe déjà. C'est la distinction principale avec la redirection simple en sortie. Ainsi, avec la redirection double en sortie, on ne supprime pas le contenu du fichier s'il existe déjà mais on écrit la suite à la fin du fichier.

La création de la redirection est gérée par la fonction `double_sortie_pipe`, appelée par `redirection_dans_pipe`. Au sein de la fonction `double_sortie_pipe`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot de la ligne s'il est égal à `>>`. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_redir_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection `>>`, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_APPEND` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'ajout du contenu à la fin du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie (correspondant au fichier 1) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`. Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[ligne][i]=0` et `mot[ligne][i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

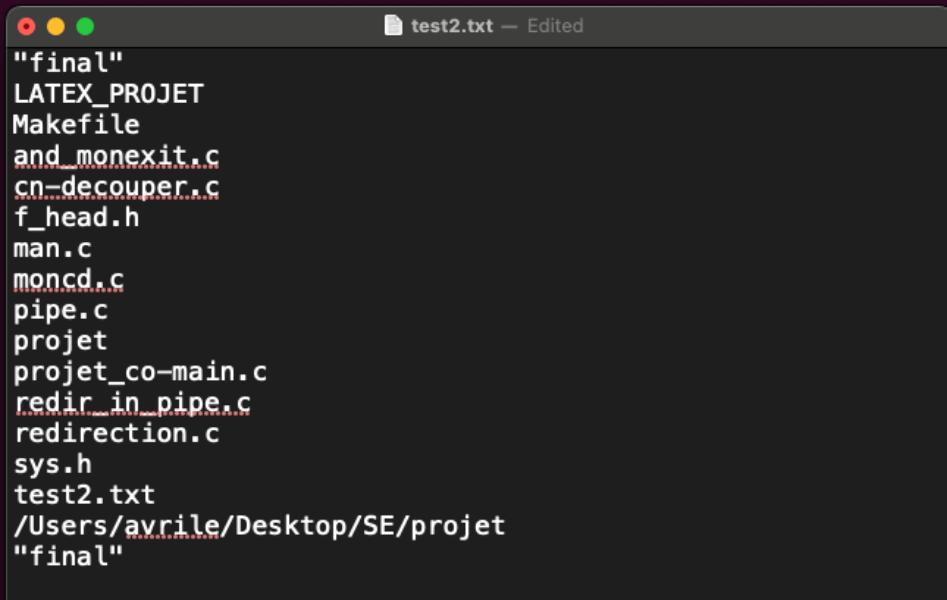
Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis, nous retournons 2.

```
79 int double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){  
80  
81     /* redirection simple en sortie > */  
82     if (strcmp(mots_pipe[ligne][i], ">>") == 0){  
83  
84         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le > */  
85  
86         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode);  
87             /* write only, créé le fichier ou ajoute à la suite si existe */  
88         if (fd_redir_sortie < 0)  
89             perror("fd_redir_sortie");  
90  
91         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */  
92         if (t < 0)  
93             perror("dup2 fd_redir_sortie");  
94  
95         mots_pipe[ligne][i] = 0; /* suppression du >> */  
96         mots_pipe[ligne][i + 1] = 0; /* suppression du nom de fichier */  
97  
98         close(fd_redir_sortie); /* fermeture */  
99         return 2; /* signe continue dans gestion_redirection */  
100    }  
101    return 0 ;  
102}
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection double en sortie. Nous avons utilisé trois commandes différentes avec des pipes : `ls`, `pwd` et `echo`. Nous avons aussi illustré une erreur, lorsque la commande n'est pas reconnue. Dans ce cas-là, l'erreur apparaît sur le terminal car il ne s'agit pas du `stdout` mais du `stderr`. Nous constatons que les redirections successives sont bien ajoutées à la fin du fichier.

```
? ls >> test2.txt | echo "bonjour"
"bonjour"
? ls | pwd >> test2.txt | echo "au revoir"
"au revoir"
? ls | wc | echo "final" >> test2.txt
? ls | wc | echox "merci" >> test2.txt
echox:notfound enfant 1
```



A screenshot of a terminal window titled "test2.txt — Edited". The window displays the following text:

```
"final"
LATEX_PROJET
Makefile
and_monexit.c
cn-decouper.c
f_head.h
man.c
moncd.c
pipe.c
projet
projet_co-main.c
redir_in_pipe.c
redirection.c
sys.h
test2.txt
/Users/avrile/Desktop/SE/projet
"final"
```

FIGURE 38 – Nous redirigeons les commandes en sortie vers le fichier "test2.txt". Les sorties des commandes successives sont ajoutées à la fin du fichier. Lorsqu'il y a une erreur, elle apparaît sur stderr et non sur stdout.

1.11.3 La redirection simple de stderr en sortie 2> avec les pipes

La création de la redirection

2> est une redirection de la sortie d'erreur `stderr` vers un fichier. Elle crée le fichier s'il n'existe pas et elle l'écrase s'il existe déjà.

La redirection simple de `stderr` en sortie est gérée par la fonction `stderr_simple_sortie_pipe`, appelée par `redirection_dans_pipe`.

Au sein de la fonction `stderr_simple_sortie_pipe`, grâce à la fonction `strcmp`, nous vérifions, pour chaque mot de la ligne s'il est égal à 2>. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant. Nous définissons également la variable `fd_stderr_s_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection 2>, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_TRUNC` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écrasement du contenu du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie d'erreur standard `stderr` (correspondant au fichier 2) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`. Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[ligne][i]=0` et `mot[ligne][i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous retournons 2.

```
105 int stderr_simple_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){  
106  
107     /* redirection simple erreur en sortie 2> */  
108     if (strcmp(mots_pipe[ligne][i], "2>") == 0){  
109         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le 2> */  
110  
111         int fd_stderr_s_redir = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;  
112         /* write only, créé le fichier ou le remet à 0 si existe */  
113         if (fd_stderr_s_redir < 0)  
114             perror("fd_stderr_s_redir");  
115  
116         int t = dup2(fd_stderr_s_redir, 2); /* stderr redirigée vers l'arg du 2> */  
117         if (t < 0)  
118             perror("dup2 fd_stderr_sortie");  
119  
120         mots_pipe[ligne][i] = 0;  
121         mots_pipe[ligne][i+1]=0;  
122  
123         close(fd_stderr_s_redir);  
124         return 2; /* signe continue dans gestion_redirection */  
125     }  
126     return 0;  
127 }  
128 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection simple de `stderr` en sortie avec des pipes. Nous avons utilisé trois commandes différentes : `lss`, `echox` et `commande_inconnue`. Nous avons aussi illustré une commande qui n'inscrit rien dans le fichier puisqu'elle ne cause pas d'erreur, c'est la commande `ls`. Ensuite, nous avons retesté la commande `echox` pour illustrer que le contenu précédent du fichier était écrasé. Par ailleurs, lors de l'utilisation de la redirection avec les pipes, nous avons volontairement utilisé d'autres commandes entraînant des erreurs pour illustrer que la réinitialisation des valeurs initiales pour les entrées-sorties avait bien eu lieu.

The terminal window shows the following session:

```

./projet
? lss 2> test3_1.txt | ls | wcc
wcc:notfound enfant 3
? ls | echoc "bonjour" 2> test3_2.txt | echox "ok"
echox:notfound enfant 3
? ls | ls | commande_inconnue 2> test3_3.txt
? ls | ls | ls 2> test3_4.txt
LATEX_PROJET      man.c
Makefile          moncd.c
and_monexit.c    pipe.c
cn-decouper.c     projet
f_head.h         projet_co-main.c
? █

```

On the right, four file browser windows are shown:

- test3_1.txt — Edited: lss:notfound enfant 1
- test3_2.txt — Edited: echoc:notfound enfant 2
- test3_3.txt — Edited: commande_inconnue:notfound enfant 3
- test3_4.txt: sys.h, test2.txt, test3.txt, test3_3.txt, test3_4.txt, test4.txt

FIGURE 39 – Nous redirigeons stderr vers des fichiers successifs. Nous illustrons les commandes : "lss", "echox", "commande_inconnue" (qui sont redirigées vers les fichiers). Les autres commandes causant des erreurs ne sont pas redirigées. En revanche, la commande "ls" apparaît sur le terminal (stdout) car elle ne cause pas d'erreur.

The terminal window shows the following session:

```

? ls | echox "adieu" 2> test3_1.txt
? █

```

A file browser window titled "test3_1.txt — Edited" is shown with the message "echox:notfound enfant 2".

FIGURE 40 – Nous relançons une commande causant une erreur en la redirigeant vers le fichier "test3_1". Le contenu précédent du fichier est écrasé.

1.11.4 La redirection double de stderr en sortie 2» avec les pipes

La création de la redirection

2» est une redirection de la sortie d'erreur `stderr` vers un fichier. Elle crée le fichier s'il n'existe pas et elle écrit à la fin s'il existe déjà. C'est la distinction principale avec la redirection simple de `stderr` en sortie.

La redirection double de `stderr` en sortie est gérée par la fonction `stderr_double_sortie_pipe`, appelée par `redirection_dans_pipe`.

Au sein de la fonction `stderr_double_sortie_pipe`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot de la ligne s'il est égal à 2». Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_stderr_d_sortie`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection 2», nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT`, `O_APPEND` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écriture à la fin du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie d'erreur standard `stderr` (correspondant au fichier 2) vers notre fichier. Nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`.

Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[ligne][i]=0` et `mot[ligne][i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous retournons 2.

```
130 int stderr_double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){  
131  
132     /* redirection double erreur en sortie 2» */  
133     if (strcmp(mots_pipe[ligne][i], "2»") == 0){  
134         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le 2» */  
135  
136         int fd_stderr_d_redir = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode) ;  
137             /* write only, créé le fichier ou ajoute à la fin si existe */  
138         if (fd_stderr_d_redir < 0)  
139             perror("fd_stderr_d_redir");  
140  
141         int t = dup2(fd_stderr_d_redir, 2); /* stderr redirigée vers l'arg du 2» */  
142         if (t < 0)  
143             perror("dup2 fd_stderr_sortie");  
144  
145         mots_pipe[ligne][i] = 0;  
146         mots_pipe[ligne][i+1]=0;  
147  
148         close(fd_stderr_d_redir);  
149         return 2; /* signe continue dans gestion_redirection */  
150     }  
151     return 0;  
152 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection double de `stderr` en sortie. Nous avons utilisé trois commandes différentes : `lss`, `echox` et `commande_inconnue`. Nous avons aussi illustré une commande qui n'inscrit rien dans le fichier puisqu'elle ne cause pas d'erreur, c'est la commande `ls`.

Nous constatons que lors de chaque redirection le nouveau contenu est ajouté à la fin du fichier.

The screenshot shows two windows. The top window is a terminal session titled `./projet`. It displays the following commands and their outputs:

```
? lss 2>> test4.txt | echox "bonjour"
echox:notfound enfant 2
? echo "bonjour" | wc | echox "bonjour" 2>> test4.txt
? ls | wc | commande_inconnue 2>> test4.txt
? ls | ls | ls 2>> test4.txt
```

Below these commands, the terminal lists files in the current directory:

LATEX_PROJET	man.c	redir_i
Makefile	moncd.c	redirec
sys.h		test2.t
test3.t		.c

The bottom window is a file editor titled `test4.txt — Edited`. It contains the following text:

```
lss:notfound enfant 1
echox:notfound enfant 3
commande_inconnue:notfound enfant 3
```

FIGURE 41 – Nous redirigeons stderr en sortie vers le fichier "test4.txt". Les sorties successives de stderr sont ajoutées à la fin du fichier.

1.11.5 La redirection de la sortie et de la sortie d'erreur vers un fichier >& avec les pipes

La création de la redirection

La redirection >&, qui est l'une des graphies permettant d'arriver à ce résultat (source : https://www.gnu.org/software/bash/manual/html_node/Redirections.html) redirige à la fois la sortie standard et la sortie d'erreur vers un fichier. Les deux flux sont fusionnés et traités de la même manière. Cette redirection va créer le fichier s'il n'existe pas et va écraser son contenu s'il existe déjà.

Nous définissons la redirection de la sortie et de la sortie d'erreur vers un fichier grâce à la fonction `stderr_stdout_sortie_pipe`, appelée par `redirection_dans_pipe`.

Au sein de `stderr_stdout_sortie_pipe`, grâce à la fonction `strcmp`, nous vérifions pour chaque mot de la ligne s'il est égal à >&. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant. Nous définissons également la variable `fd_s_sortie_and`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie.

Afin de respecter les spécifications de la redirection >&, nous avons ajouté les drapeaux `O_WRONLY`, `O_CREAT_`, `O_TRUNC` (qui correspondent à l'écriture, à la création du fichier s'il n'existe pas et à l'écrasement du contenu du fichier s'il existe déjà). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger la sortie d'erreur (correspondant au fichier 2) vers notre fichier. Nous effectuons la même opération avec la sortie `stdout` qui correspond au fichier 1. En effet, dans le cadre de cette redirection, à la fois la sortie standard et la sortie d'erreur sont redirigées vers le même fichier. Ensuite, nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`.

Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[ligne][i]=0` et `mot[ligne][i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous retournons 2.

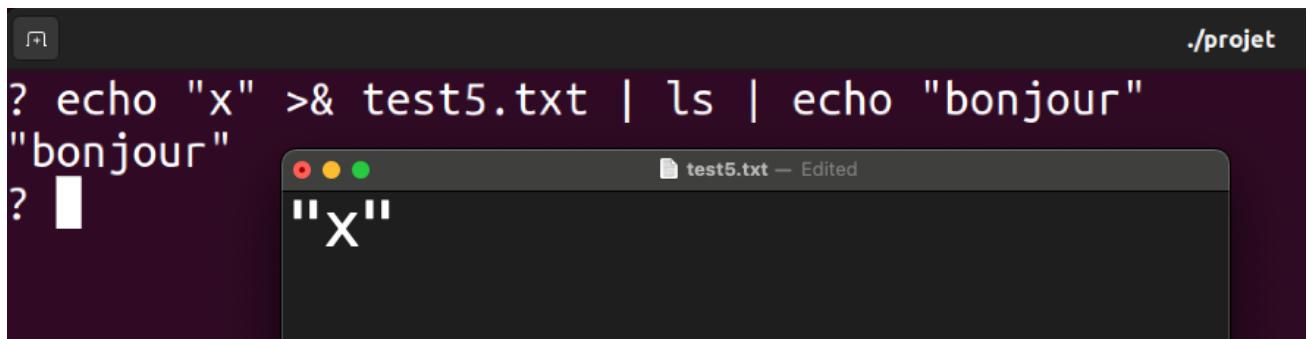
```
154 int stderr_stdout_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){  
155  
156     if (strcmp(mots_pipe[ligne][i], ">&") == 0){  
157         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le > */  
158  
159         int fd_s_sortie_and = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;  
160         /* write only, créé le fichier ou le remet à 0 si existe */  
161         if (fd_s_sortie_and < 0)  
162             perror("fd_s_sortie_and");  
163  
164         int t = dup2(fd_s_sortie_and, 1); /* redir stout vers le fichier */  
165         if (t < 0)  
166             perror("dup2 stout fd_s_sortie_and");  
167  
168         int tt = dup2(fd_s_sortie_and, 2); /* redir stderr vers le fichier */  
169         if (tt < 0)  
170             perror("dup2 stderr fd_s_sortie_and");  
171  
172         mots_pipe[ligne][i] = 0; /* suppression du > */  
173         mots_pipe[ligne][i+1] = 0; /* suppression du nom de fichier */  
174  
175         close(fd_s_sortie_and); /* fermeture */  
176         return 2; /* signe continue dans gestion_redirection */  
177     }  
178  
179     return 0;  
180 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection des sorties standard et d'erreur vers un fichier (`>&`). Les exemples illustrent l'utilisation de la redirection avec les pipes. On a veillé à montrer que les réinitialisations des valeurs initiales pour `stdin` et `stderr` avaient bien eu lieu.

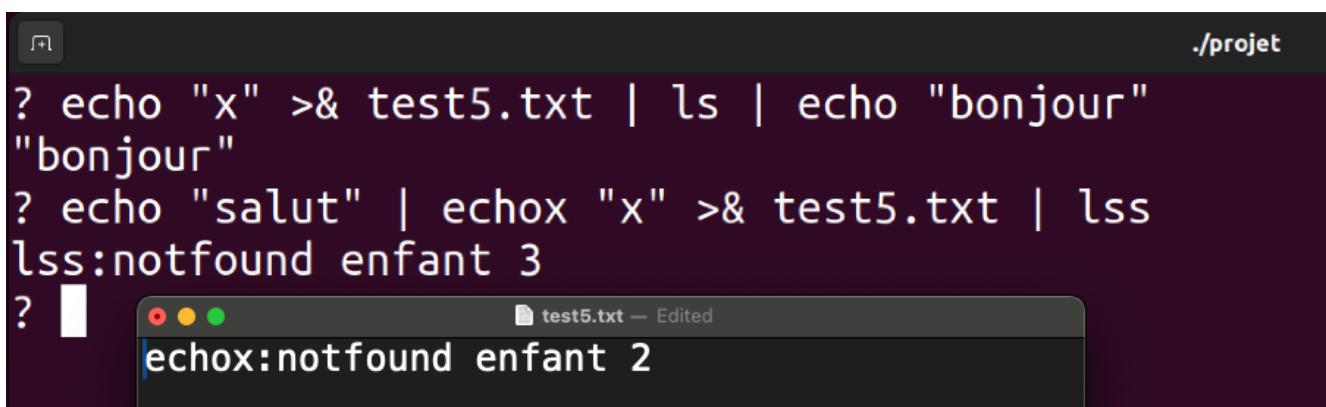
Nous avons utilisé trois commandes différentes : `echo`, `echox` et `ls`. Nous avons aussi illustré une commande qui ne fonctionne pas, lorsque l'on écrit la redirection `> &`, c'est-à-dire avec un espace. Il convient d'écrire la commande de redirection sans espace.

Nous constatons que, lors de chaque redirection, le contenu du fichier est écrasé (dans le cas où le fichier existait déjà).



The screenshot shows a terminal window titled `./projet`. The command entered is `? echo "x" >& test5.txt | ls | echo "bonjour"`. The output shows the word "bonjour" followed by a question mark and a blank line. Below the terminal is a file editor window titled `test5.txt — Edited` containing the text "`"X"`".

FIGURE 42 – Nous redirigeons les sorties standard et d'erreur vers un fichier. Nous illustrons avec le passage d'une commande "echo".



The screenshot shows a terminal window titled `./projet`. The command entered is `? echo "x" >& test5.txt | ls | echo "bonjour"`. The output shows the word "bonjour" followed by a question mark and a blank line. Below the terminal is a file editor window titled `test5.txt — Edited` containing the text "`echox:notfound enfant 2`".

FIGURE 43 – Nous redirigeons les sorties standard et d'erreur vers un fichier. Nous illustrons avec le passage d'une commande "echox" (qui cause une erreur).

The screenshot shows a terminal window with a dark background and white text. At the top right, it says `./projet`. The terminal history shows:

```
? echo "x" >& test5.txt | ls | echo "bonjour"  
"bonjour"  
? echo "salut" | echox "x" >& test5.txt | lss  
lss:notfound enfant 3  
? ls | wc | ls >& test5.txt  
?
```

Below the terminal is a file viewer window titled `test5.txt — Edited`. It contains the following list of files:

- LATEX_PROJET
- Makefile
- and_monexit.c
- cn-decouper.c
- f_head.h
- man.c
- moncd.c
- pipe.c
- projet
- projet_co-main.c
- redir_in_pipe.c
- redirection.c
- sys.h
- test5.txt

FIGURE 44 – Nous redirigeons les sorties standard et d’erreur vers un fichier. Nous illustrons avec le passage d’une commande “ls”.

```
? ls | wc | lss > & test5.txt  
lss:notfound enfant 3  
?
```

FIGURE 45 – Nous redirigeons les sorties standard et d’erreur vers un fichier. Nous illustrons avec le passage d’une commande “echox” qui cause une erreur. En revanche, nous n’avons pas bien orthographié la commande de redirection car nous y avons ajouté un espace. Ça ne marche pas.

1.11.6 La redirection de l'entrée simple < avec les pipes

La création de la redirection

La redirection < redirige l'entrée standard (`stdin`) d'une commande depuis un fichier, plutôt que depuis le clavier.

Nous définissons la redirection de l'entrée vers un fichier grâce à la fonction `simple_entree_pipe`, appelée par `redirection_dans_pipe`. Au sein de `simple_entree_pipe`, grâce à la fonction `strcmp`, nous vérifions pour chacun des mots de la ligne s'il est égal à <. Si c'est le cas, nous définissons la variable `sortie`, qui correspond au mot suivant.

Nous définissons également la variable `fd_redir_s_entree`, qui correspond au numéro de fichier attribué lors de l'ouverture de notre fichier de sortie. Afin de respecter les spécifications de la redirection <, nous avons ajouté le drapeau `O_RDONLY` (qui correspond à la lecture du fichier). Bien entendu, nous vérifions qu'il n'y a pas d'erreur à l'ouverture du fichier.

Après l'attribution d'un numéro de fichier à notre fichier de sortie, nous pouvons appeler la fonction `dup2` afin de rediriger l'entrée standard vers notre fichier. Ensuite, nous vérifions qu'il n'y a pas eu d'erreur lors de l'exécution de la fonction `dup2`.

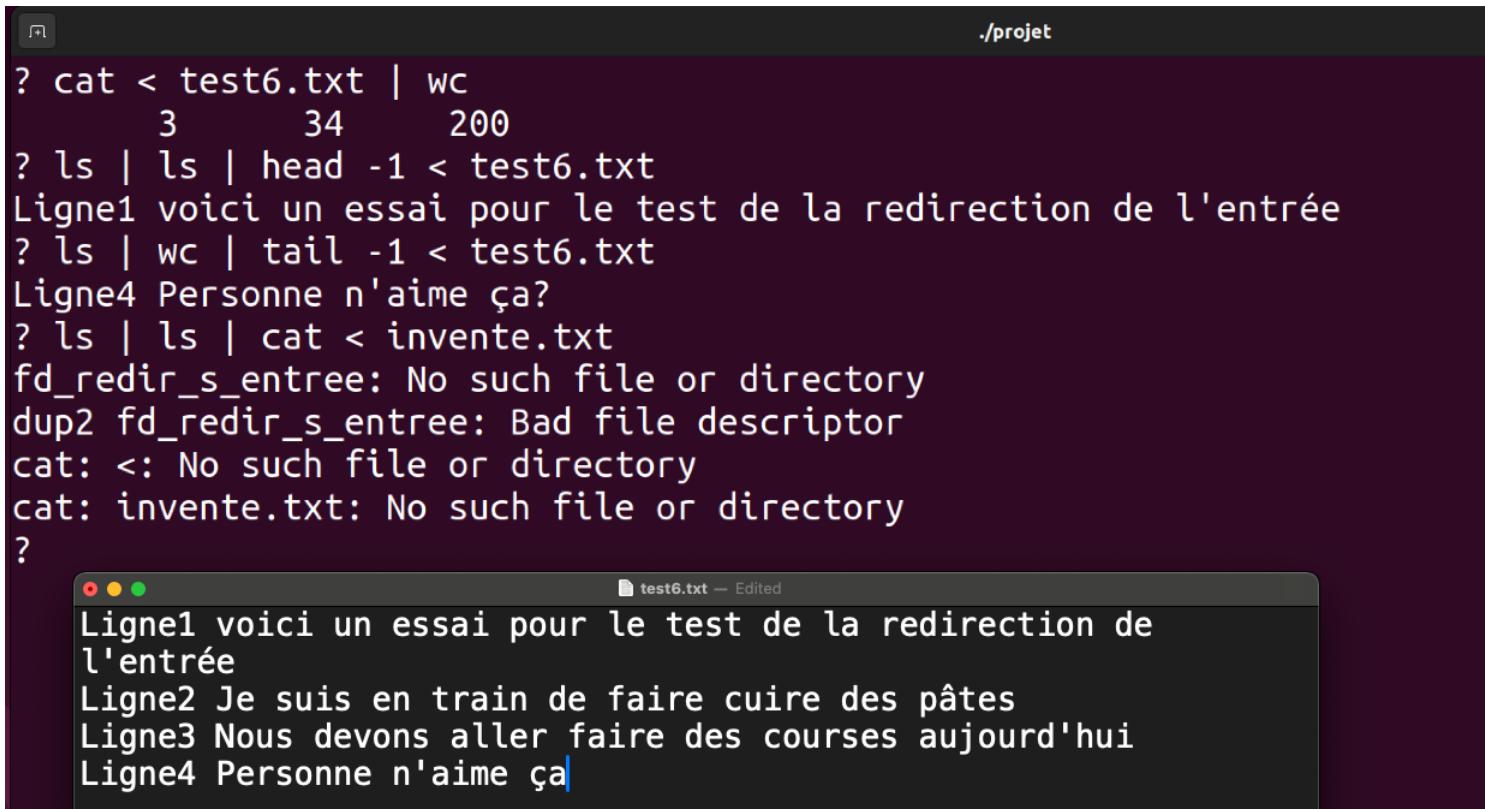
Par ailleurs, nous veillons à supprimer de la liste des mots la redirection ainsi que son fichier de sortie avec `mot[ligne][i]=0` et `mot[ligne][i+1]=0`. En conséquence, nous diminuons le `nb_total_mots`.

Finalement, nous fermons le fichier. Une fermeture de fichier n'échoue jamais, nous n'avons donc pas besoin de vérifier sa réussite. Puis nous retournons 2.

```
182 int simple_entree_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){  
183  
184     /* redirection simple en entrée < */  
185     if (strcmp(mots_pipe[ligne][i], "<") == 0){  
186         char* entree = mots_pipe[ligne][i+1]; /* le fichier entrée après le < */  
187  
188         int fd_redir_s_entree = open(entree, O_RDONLY, mode) ;  
189         /* read only */  
190         if (fd_redir_s_entree < 0)  
191             perror("fd_redir_s_entree");  
192  
193         int t = dup2(fd_redir_s_entree, 0); /* entrée redirigée vers l'arg du < */  
194         if (t < 0)  
195             perror("dup2 fd_redir_s_entree");  
196  
197         mots_pipe[ligne][i] = 0; /* suppression du < */  
198         mots_pipe[ligne][i+1] = 0; /* suppression du nom de fichier */  
199  
200         close(fd_redir_s_entree);  
201         return 2; /* signe continue dans gestion_redirection */  
202     }  
203     return 0;  
204 }
```

Les tests

Ci-après, voici des exemples d'utilisation de la redirection de l'entrée standard vers un fichier. Nous avons utilisé des commandes différentes : `cat`, `head` et `tail`. Nous avons aussi illustré une commande qui ne fonctionne pas, lorsque l'on tente de rediriger l'entrée vers un fichier qui n'existe pas.



```

./projet
? cat < test6.txt | wc
      3      34     200
? ls | ls | head -1 < test6.txt
Ligne1 voici un essai pour le test de la redirection de l'entrée
? ls | wc | tail -1 < test6.txt
Ligne4 Personne n'aime ça?
? ls | ls | cat < invente.txt
fd_redir_s_entree: No such file or directory
dup2 fd_redir_s_entree: Bad file descriptor
cat: <: No such file or directory
cat: invente.txt: No such file or directory
?

```

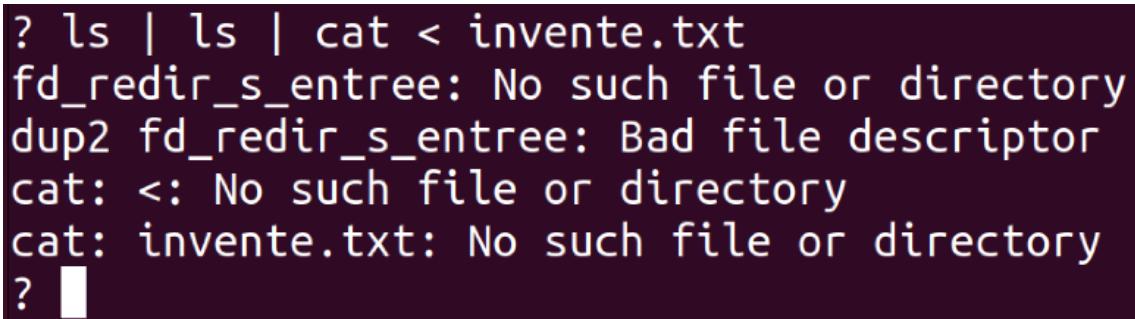
The terminal window title is "test6.txt — Edited". The content of the file is:

```

Ligne1 voici un essai pour le test de la redirection de
l'entrée
Ligne2 Je suis en train de faire cuire des pâtes
Ligne3 Nous devons aller faire des courses aujourd'hui
Ligne4 Personne n'aime ça

```

FIGURE 46 – Nous redirigeons l'entrée standard vers le fichier "test6.txt" et nous appelons les commandes "cat", "head" et "tail". Nous illustrons notre redirection avec l'usage des pipes.



```

? ls | ls | cat < invente.txt
fd_redir_s_entree: No such file or directory
dup2 fd_redir_s_entree: Bad file descriptor
cat: <: No such file or directory
cat: invente.txt: No such file or directory
?

```

FIGURE 47 – Nous redirigeons l'entrée standard vers le fichier "moninvention.txt" et nous appelons la commande "cat". Nous obtenons une erreur car le fichier n'existe pas.

Conclusion pour les redirections d'entrées-sorties avec les pipes : La fonction `redirection_dans_pipe`, qui se trouve au sein du fichier `redir_in_pipe.c`, est la fonction en charge de la gestion des redirections lorsque des pipes sont utilisés. En son sein, cette fonction est divisée en sous-fonctions qui correspondent à la création des redirections.

Nous avons présenté six redirections : `>`, `>>`, `2>`, `2>>`, `>&` et `<`. Pour chaque redirection, nous avons présenté la fonction particulière (celle dédiée à la création de la redirection) et nous avons illustré par trois exemples et une erreur.

1.12 Les pipes

– Les pipes

L'organisation générale de la gestion des pipes

Dans la fonction `main` de notre programme, nous déclarons un booléen `ilya_pipe`, initialisé à faux. Nous déclarons un tableau bidimensionnel de int `mes_pipes [MaxPipes] [2]` avec `MaxPipes` défini à 3 (au sein du fichier `f_head.h`). Nous déclarons également un tableau bidimensionnel `char* mots_pipes [MaxPipes] [MaxMot]`. Ce tableau nous permettra de stocker les différentes commandes se trouvant avant, entre et après les pipes.

Dans la boucle du prompt, nous initialisons la variable `int nb_pipe_total` avec le retour de la fonction `cb_pipe`. L'appel à cette fonction dans le prompt, au début, est important car c'est cette fonction qui va permettre le passage du bool `ilya_pipe` à vrai, en cas de pipe(s) dans la ligne de commande.

Plus loin, mais toujours avant le `fork()`, nous vérifions si le booléen `ilya_pipe` est vrai. Dans ce cas, nous appelons la fonction `gestion_pipes` qui s'occupe de traiter les pipes, selon qu'il y en a un ou deux sur la ligne de commande.

```
27 int
28 main(int argc, char * argv[]) {
46     int mes_pipes[MaxPipes][2]; /* tableaux pour les FD des pipes */
47
48     bool ilya_pipe = false ;
49
50     char * mots_pipe[MaxPipes][MaxMot]; /* sous-tab de mots pour c/ côté de pipe*/
51
52     int nb_pipe_total = cb_pipe(mot, nb_total_mots, &ilya_pipe);
53     /* renvoie nb total de pipes | + permet passage ilya_pipe à vrai */
54
55     /* LES REDIRECTIONS */
56     if (!ilya_pipe){
57         redir = true ;
58         redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &redir_s_entree, &
59         old_entree, &old_sortie, &old_erreur,&redir, &reset);
60     }
61
62     /* LES PIPES */
63     else if (ilya_pipe){
64         gestion_pipes(nb_pipe_total, nb_total_mots, mot, mots_pipe, mes_pipes, dirs,
65         pathname, sizeof(pathname), attend_enfant, &ilya_pipe) ;
66     }
67
68     /* FORK INITIAL SI PAS DE PIPE */
69     else {
70
71         tmp = fork(); /* lancement du processus enfant */
72     }
73 }
```

Le fichier `pipe.c`

Afin de favoriser la modularité du code, l'ensemble des fonctions en lien avec les pipes est contenu dans le fichier `pipe.c`.

La fonction `cb_pipe`

La première fonction contenue dans ce fichier est la fonction `cb_pipe`. Cette fonction permet de détecter la présence ou non de pipe(s) parmi les commandes.

La fonction `cb_pipe` est appelée dans la fonction `main` afin d'attribuer une valeur à la variable `nb_pipe_total`.

La fonction `cb_pipe` renvoie le nombre total de pipes présents sur la ligne de commande et permet le passage du booléen `ilya_pipe` à vrai.

La fonction est construite afin de boucler sur chaque mot de la ligne de commande. Pour chaque mot correspondant au signe pipe ("|"), elle incrémente le total des pipes. Avant de modifier la valeur du booléen `ilya_pipe` à vrai, on le déréférence car la fonction utilise un pointeur.

```
1 /* **** */
2 # Nom ..... : pipe.c
3 # Rôle ..... : Gestion des pipes : | et |
4 # Auteur ..... : Avrile Floro
5 # Version ..... : V0.1 du 26/10/2023
6 # Licence ..... : réalisé dans le cadre du cours de SE
7 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
8 ****
15 int cb_pipe(char* mot[], int nb_total_mots, bool* ilya_pipe)
16 { /* compte le nb de signe "pipe" | dans la ldc */
17     int c = 0;
18     for (int i = 0; i < nb_total_mots - 1 ; i++) {
19         if (strcmp(mot[i], "|") == 0) /* à chaque pipe re ontré */
20         {
21             c++;
22             *ilya_pipe= true;
23         }
24     }
25 }
26 return c; /* renvoie le nb de pipes */
27 }
```

La fonction gestion_pipes

La fonction `gestion_pipes` est la fonction centrale permettant la gestion des pipes. Cette fonction est appelée en présence d'un ou de plusieurs pipes. Elle appelle ensuite la fonction appropriée (selon qu'il y ait un ou deux pipes).

Parmi ses nombreux arguments, elle reçoit le `nb_pipe_total`, obtenu dans la fonction `main` grâce à la fonction `cb_pipe`, discutée précédemment.

Lorsque la fonction `gestion_pipes` est appelée, cela signifie que le booléen `ilya_pipe` est vrai, dès lors, la variable `nb_pipe_total` doit être supérieure à 0.

Si la variable `nb_pipe_total` est supérieure à 0 (c'est-à-dire s'il y a un ou deux pipes) alors la fonction `div_tab` est appelée. Cette fonction permet de construire des sous-tableaux contenant les mots de chaque commande, qui devront être réutilisés avec les pipes.

Si la variable `nb_pipe_total` est égale à 1, c'est-à-dire s'il y a un pipe sur la ligne de commande, alors la fonction `un_pipe` est appelée. Sa valeur est assignée à une variable `v`. Peu importe le retour de la fonction `un_pipe`, on retourne 0.

Si la variable `nb_pipe_total` est égale à 2, c'est-à-dire s'il y a deux pipes sur la ligne de commande, alors la fonction `deux_pipes` est appelée. Sa valeur est assignée à une variable `v`. Peu importe le retour de la fonction `deux_pipes`, on retourne 0.

```

39     int v = un_pipe(mes_pipes, pathname, pathname_size, dirs, mots_pipe, attend_enfant,
40     ilya_pipe, nb_mots_lastpipe);
41     if (v == 0 || v == -1)
42         return 0;
43 }
44 if (nb_pipe_total==2){ /* s'il y a deux | */
45     int v = deux_pipes(mes_pipes, pathname, pathname_size, dirs, mots_pipe, attend_enfant,
46     ilya_pipe, nb_mots_lastpipe);
47     if (v == 0 || v == -1)
48         return 0;
49 }
50 return 0;
}

```

La fonction div_tab

La fonction `div_tab` permet de diviser la ligne de commande comprenant des pipes en plusieurs sous-tableaux, chacun correspondant à une commande distincte. Elle est appelée par la fonction `gestion_pipes` afin d'effectuer un traitement préalable de la ligne de commande. Cette division préalable est essentielle pour isoler les commandes à exécuter séquentiellement avec les pipes. La fonction initialise les variables `ligne` (pour le sous-tableau actuel) et `mot_actuel` (pour l'indice du mot courant dans ce sous-tableau) à 0.

Grâce à une boucle, on parcourt les mots de la ligne de commande. Lorsque le mot est un pipe ("|"), on va inscrire dans le sous-tableau un caractère NULL, qui correspond au dernier élément d'un tableau. On va également réinitialiser le `mot_actuel` à 0 car on recommencera à compter à partir de 0 dans le prochain sous-tableau. À ce moment, `nb_mots_lastpipe` est aussi réinitialisé à 0 pour préparer le comptage des mots de la prochaine commande. Enfin, on va incrémenter `ligne`. Cela signale qu'un nouveau sous-tableau (nouvelle commande) va commencer.

En revanche, lorsque le mot lu est différent de pipe ("|"), alors on le copie dans le tableau et on incrémente le mot actuel pour pouvoir continuer de parcourir le tableau. Simultanément, `nb_mots_lastpipe` est incrémenté pour chaque mot ajouté, ce qui permet de suivre le nombre de mots dans le dernier sous-tableau traité.

En outre, il est important de rajouter après la boucle, un dernier caractère NULL pour le dernier élément du tableau. Nous avons été confrontée à ce bug causé par l'absence de caractère NULL à la fin du tableau.

Afin de faciliter la gestion de la commande `moncd` avec les pipes, nous avons rajouté un pointeur vers un `int nb_mots_lastpipe`. Cette variable permet de garder une trace du nombre de mots dans le dernier pipe. On la passera comme argument à la fonction `appel_mon_cd`. Son fonctionnement a déjà été étudié en détail précédemment.

```

53 int div_tab(int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot], int mes_pipes[][2], int
54 * nb_mots_lastpipe){
55 /* permet de remplir les sous-tableaux [lignes] correspondant aux commandes des pipes */
56
57 int ligne = 0; /* on commence au 1er sous-tableau */
58 int mot_actuel=0; /* on commence au premier mot */
59 for (int i = 0 ; i < nb_total_mots ; i++){
60
61     if (strcmp(mot[i], "|") == 0){ /* si le caractère est un | */
62         mots_pipe[ligne][mot_actuel] = NULL; /* on le remplace par NULL */
63         mot_actuel = 0; /* on recommencera à compter à 0 dans le prochain sous-tableau */
64         (*nb_mots_lastpipe)=0; /* init nb laspipe */
65         ligne++; /* changement de sous-tableau car rencontré un pipe */
66     }
67
68     else { /* si le caractère est pas un | */
69         mots_pipe[ligne][mot_actuel] = mot[i]; /* j'écris le mot dans le tableau */
70         mot_actuel++; /* je parcours les mots */
71         (*nb_mots_lastpipe)++; /* on incrémente */
72     }
73 }
74 mots_pipe[ligne][mot_actuel] = NULL; /* on termine le tableau avec NULL */
75

```

```
76     return 0;
77 }
```

La fonction un_pipe

La fonction `un_pipe` est appelée par la fonction `gestion_pipe` lorsque le nombre total de pipes sur la ligne de commande est égal à 1. En particulier, la fonction `un_pipe` gère les forks et l'exécution des commandes.

Initialement, rajouter les forks a considérablement compliqué notre code. Nous avons trouvé une explication concise sur un forum Unix (source : <https://www.unix.com/unix-for-beginners-questions-and-answers/278569-one-parent-multiple-children-pipe-fork.html>). Nous avons adapté notre code sur ce modèle et cela a grandement simplifié son organisation.

Nous avons aussi passé `pathname size` (correspondant à `sizeof(pathname)`) parmi les arguments. En effet, puisque nous passons le tableau `char pathname[]` (qui est en réalité un pointeur sur le premier élément) en argument de la fonction, nous ne pouvons pas utiliser `sizeof` à l'intérieur de la fonction.

In short, how fork() really works:

Code:

```
pid_t pid=fork();

if(pid < 0)
{
    perror("couldn't fork");
    exit(1);
}

if(pid == 0) {
    // Child code
    do_stuff();
    exit(0);
}

// Parent code
```

FIGURE 48 – Un exemple simple du fonctionnement des `fork()`.

Nous créons le pipe avec la commande `pipe(mes_pipes[0])` et nous vérifions que cela ne cause pas d'erreur. Ensuite nous créons une variable `tmp` que nous associons au résultat du `fork()` et nous vérifions que cela ne cause pas d'erreur. Si le résultat du fork est égal à 0, alors nous nous trouvons dans le processus enfant numéro 1. Nous redirigeons la sortie grâce à `dup2` (car nous souhaitons que la sortie soit dirigée vers le pipe) et nous fermons l'entrée et la sortie du pipe.

On gère avec la fonction `redirection_dans_pipe` les possibles redirections. Grâce à la fonction `strcmp` et à des `if` successifs (permettant d'éviter le lancement de `execv` en cas de réalisation), on vérifie si le premier mot de la première ligne est `monman`, `moncd` ou `monexit`, et selon, on appelle la fonction appropriée parmi `monman`, `appel_mon_cd` et `mon_exit_pipe`. Finalement, si aucun `if` ne s'est réalisé, on lance l'exécution de la commande contenue dans notre premier sous-tableau avec `execv`.

Si la variable `tmp` n'est pas négative et n'est pas égale à 0, alors nous sommes dans le processus parent.

Nous n'avons pas placé ce processus parent au sein d'un `if` ou d'un `else` afin de gagner en lisibilité. Au sein du processus parent, nous pouvons lancer le second fork en associant `tmp2` à `fork`. En cas de pipe, il est nécessaire de lancer plusieurs processus enfants. En effet, un processus distinct exécute la commande qui se trouvait respectivement avant ou après le pipe.

Sur le même modèle que pour le premier fork, nous vérifions pour le second fork que le fork ne cause pas d'erreur. Si la variable `tmp2` est égale à 0, alors on est dans le processus enfant. On redirige l'entrée (car on souhaite que l'entrée provienne du pipe) avec `dup2` et on ferme l'entrée et la sortie du pipe. On gère également, avec la fonction `redirection_dans_pipe`, les possibles redirections.

Grâce à la fonction `strcmp` et à des `if` successifs (permettant d'éviter le lancement de `execv` en cas de réalisation), on vérifie si le premier mot de la deuxième ligne est `monman`, `moncd` ou `monexit`, et selon, on appelle la fonction appropriée parmi `monman`, `appel_mon_cd` et `mon_exit_pipe`.

Finalement, si aucun `if` ne s'est réalisé, on lance l'exécution de la commande contenue dans notre deuxième sous-tableau avec `execv`.

Enfin, si `tmp2` n'est pas égal à 0, alors cela signifie que l'on se trouve une nouvelle fois dans le processus parent. On ferme alors, au sein du processus parent, l'entrée et la sortie du pipe. On attend que tous les processus enfant soient terminés avec `while (wait(NULL) > 0)` ; lorsque `attend_enfant` est vrai. L'utilisation de cette construction pour le `wait` nous a permis de résoudre des bugs persistents. Enfin, nous passons le booléen `ilya_pipe` à faux.

```

79 int un_pipe(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
80             mots_pipe[] [MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe){
81
82     int tmp, tmp2;
83
84     if (pipe(mes_pipes[0]) < 0){ /* vérification erreur pipe */
85         perror("pipe");
86         return -1; }
87
88     /* PROCESSUS PARENT: FORK1 */
89     if ((tmp= fork())<0){
90         perror("fork");
91         return -1; }
92
93     /* PROCESSUS ENFANT 1 */
94     if (tmp ==0) {
95         dup2(mes_pipes[0][1], 1); /* redirection sortie vers pipe */
96         close(mes_pipes[0][0]); /* fermeture pipe */
97         close(mes_pipes[0][1]);
98
99         int nb_elt_tab1 = nb_mots(mots_pipe, 0); /* compte les mots du pipe */
100
101        redirection_dans_pipe(mots_pipe, 0);
102
103        if (strcmp(mots_pipe[0][0], "monman") == 0){ /*si monman*/
104            monman(mots_pipe[0]);
105            exit(0); /* sinon on reste bloqué on attend */
106        }
107
108        if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
109            appel_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
110            exit(0); /* sinon on reste bloqué on attend */
111        }
112
113        if (strcmp(mots_pipe[0][0], "monexit") == 0){ /* si monexit*/
114            mon_exit_pipe(mots_pipe[0]);
115            exit(0); /* sinon on reste bloqué on attend */
116        }
117
118        else{
119            for(int i = 0; dirs[i] != 0; i++){
120                snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
121                execv(pathname, mots_pipe[0]);
122            }
123
124            fprintf(stderr, "%s: notfound enfant 1\n", mots_pipe[0][0]);
125            exit(1);
126        }
127    }
}
```

```

128 /* PROCESSUS PARENT > FORK 2 */
129 if ((tmp2=fork()) <0){
130     perror("fork") ;
131     return -1;
132 }
133
134 /* PROCESSUS ENFANT 2 */
135 if (tmp2 ==0) {
136     dup2(mes_pipes[0][0], 0); /* redirection entrée depuis pipe */
137     close(mes_pipes[0][0]); /* fermeture pipe */
138     close(mes_pipes[0][1]);
139
140     redirection_dans_pipe(mots_pipe, 1);
141
142     if (strcmp(mots_pipe[1][0], "monman") == 0){ /* si monman*/
143         monman(mots_pipe[1]);
144         exit(0); /* sinon on reste bloqué on attend */
145     }
146
147     if (strcmp(mots_pipe[1][0], "moncd") == 0){ /* si moncd*/
148         appel_mon_cd(dirs, mots_pipe[1], nb_mots_lastpipe);
149         exit(0); /* sinon on reste bloqué on attend */
150     }
151
152     if (strcmp(mots_pipe[1][0], "monexit") == 0){ /* si monexit*/
153         mon_exit_pipe(mots_pipe[1]);
154         exit(0); /* sinon on reste bloqué on attend */
155     }
156
157     else{
158         for(int i = 0; dirs[i] != 0; i++){
159             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[1][0]);
160             execv(pathname, mots_pipe[1]);
161         }
162
163         fprintf(stderr, "%s: notfound enfant 2\n", mots_pipe[1][0]);
164         exit(1) ;
165     }
166 }
167
168 /* PROCESSUS PARENT FIN */
169
170 close(mes_pipes[0][0]); /* fermeture pipe */
171 close(mes_pipes[0][1]);
172
173 if (attend_enfant == true){
174     /* on attend tous les enfants */
175     while (wait(NULL) > 0)
176         ;
177 }
178
179 *ilya_pipe = false;
180 return 0;
181 }
```

La fonction deux_pipes

La fonction `deux_pipes` a un fonctionnement similaire à la fonction `un_pipe`, si ce n'est qu'elle traite deux pipes. Elle est appelée par la fonction `gestion_pipe` lorsque le nombre total de pipe sur la ligne de commande est égal à 2. En particulier, la fonction `deux_pipes` gère les forks et l'exécution des commandes.

Nous créons les deux pipes avec la commande `pipe(mes_pipes[0])` et `pipe(mes_pipes[1])`. Nous vérifions que cela ne cause pas d'erreur. Ensuite nous associons la variable `tmp` au résultat du premier `fork()` et nous vérifions que cela ne cause pas d'erreur. Si le résultat du fork est égal à 0, alors nous nous trouvons dans le processus enfant numéro 1. Nous redirigeons la sortie grâce à `dup2` (car nous souhaitons que la sortie soit dirigée vers le pipe) et nous fermions l'entrée et la sortie des deux pipes (même si nous n'en avons utilisé qu'un).

On gère avec la fonction `redirection_dans_pipe` les possibles redirections. Grâce à la fonction `strcmp` et à des `if` successifs (permettant d'éviter le lancement de `execv` en cas de réalisation), on vérifie si le premier mot de la première ligne est `monman`, `moncd` ou `monexit`, et selon, on appelle la

fonction appropriée parmi `monman`, `appel_mon_cd` et `mon_exit_pipe`.

Puis, si aucun des `if` ne s'est réalisé, on lance l'exécution de la commande contenue dans notre premier sous-tableau avec `execv`.

Si la variable `tmp` n'est pas négative et n'est pas égale à 0, alors nous sommes dans le processus parent. Nous n'avons pas placé ce processus parent au sein d'un `if` ou d'un `else` afin de gagner en lisibilité. Au sein du processus parent nous pouvons lancer le second fork en associant `tmp2` à `fork`. En cas de pipe, il est nécessaire de lancer plusieurs processus enfants car un processus distinct exécute la commande qui se trouvait respectivement avant ou après le pipe.

Sur le même modèle que pour le premier fork, nous vérifions pour le second fork que le fork ne cause pas d'erreur. Si la variable `tmp2` est égale à 0, alors on est dans le processus enfant. On redirige l'entrée (car on souhaite que l'entrée provienne du pipe) avec `dup2`. On redirige également la sortie vers le second pipe, toujours avec `dup2`.

On gère avec la fonction `redirection_dans_pipe` les possibles redirections.

Grâce à la fonction `strcmp` et à des `if` successifs (permettant d'éviter le lancement de `execv` en cas de réalisation), on vérifie si le premier mot de la deuxième ligne est `monman`, `moncd` ou `monexit`, et selon, on appelle la fonction appropriée parmi `monman`, `appel_mon_cd` et `mon_exit_pipe`.

Puis, si aucun des `if` ne s'est réalisé, on lance l'exécution de la commande contenue dans notre deuxième sous-tableau avec `execv`.

Si `tmp2` n'est pas égal à 0, alors cela signifie que l'on se trouve une nouvelle fois dans le processus parent. On ferme alors également au sein du processus parent l'entrée et la sortie du premier pipe, car nous n'en avons plus besoin. Ensuite, on peut procéder au troisième fork en lui associant la variable `tmp3`.

Suivant toujours la même logique, on vérifie que la variable ne soit pas négative. Si la variable `tmp3` est égale à 0, alors on se trouve dans le troisième processus enfant. On redirige alors l'entrée depuis le pipe2 avec `dup2` puis on ferme l'entrée et la sortie du pipe2.

On gère avec la fonction `redirection_dans_pipe` les possibles redirections. Grâce à la fonction `strcmp` et à des `if` successifs (permettant d'éviter le lancement de `execv` en cas de réalisation), on vérifie si le premier mot de la troisième ligne est `monman`, `moncd` ou `monexit`, et selon, on appelle la fonction appropriée parmi `monman`, `appel_mon_cd` et `mon_exit_pipe`.

Puis, si aucun des `if` ne s'est réalisé, on lance l'exécution de la commande contenue dans notre troisième sous-tableau avec `execv`.

Enfin, si la variable `tmp3` n'est ni négative, ni égale à 0, alors on se trouve dans le processus parent. On peut alors fermer le second pipe. On attend également que tous les processus enfant soient terminés avec `while (wait(NULL) >0) ;`. Enfin, nous passons le booléen `ilya_pipe` à faux.

```
186 int deux_pipes(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
187 mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe){
188
189     int tmp, tmp2, tmp3;
190
191     /* vérification réussite pipe */
192     if (pipe(mes_pipes[0]) < 0 || pipe(mes_pipes[1]) < 0){
193         perror("pipe");
194         return -1;
195     }
196
197     /* PROCESSUS PARENT > FORK 1 */
198     if ((tmp = fork()) < 0){
199         perror("fork");
200         /* vérification réussite fork1 */
201         return -1;
202     }
203
204     /* PROCESSUS ENFANT 1 */
205     if (tmp == 0) {
206         dup2(mes_pipes[0][1], 1); /* redirection sortie */
207         close(mes_pipes[0][0]); /* fermeture de tous les pipes */
208         close(mes_pipes[0][1]);
209         close(mes_pipes[1][0]);
210         close(mes_pipes[1][1]);
211
212         int nb_elt_tab1 = nb_mots(mots_pipe, 0);
213
214         redirection_dans_pipe(mots_pipe, 0);
215     }
216 }
```

```

211
212     if (strcmp(mots_pipe[0][0], "monman") == 0){ /* si monman*/
213         monman(mots_pipe[0]);
214         exit(0); /* sinon on reste bloqué on attend */
215     }
216
217     if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
218         appel_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
219         exit(0); /* sinon on reste bloqué on attend */
220     }
221
222     if (strcmp(mots_pipe[0][0], "monexit") == 0){ /* si monexit*/
223         mon_exit_pipe(mots_pipe[0]);
224         exit(0); /* sinon on reste bloqué on attend */
225     }
226
227
228     else{
229         for(int i = 0; dirs[i] != 0; i++){
230             sprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
231             execv(pathname, mots_pipe[0]);
232         }
233
234         fprintf(stderr, "%s: notfound enfant 1\n", mots_pipe[0][0]);
235         exit(1) ;
236     }
237 }
238
/* PROCESSUS PARENT > FORK 2 */
239 if ((tmp2 = fork()) <0){
240     perror("fork");
241     return -1;
242 }
243
/* PROCESSUS ENFANT 2 */
244 if (tmp2 ==0) {
245     dup2(mes_pipes[0][0], 0); /* redirection entrée depuis pipe1 */
246     dup2(mes_pipes[1][1], 1); /* redirection sortie vers pipe2 */
247     close(mes_pipes[0][0]);
248     close(mes_pipes[0][1]);
249     close(mes_pipes[1][0]);
250     close(mes_pipes[1][1]);
251
252     int nb_elt_tab2 = nb_mots(mots_pipe, 1); /* nb mots pipe 2 */
253     redirection_dans_pipe(mots_pipe, 1);
254
255     if (strcmp(mots_pipe[1][0], "monman") == 0){ /* si monman*/
256         monman(mots_pipe[1]);
257         exit(0); /* sinon on reste bloqué on attend */
258     }
259
260     if (strcmp(mots_pipe[1][0], "moncd") == 0){ /* si moncd*/
261         appel_mon_cd(dirs, mots_pipe[1], &nb_elt_tab2);
262         exit(0); /* sinon on reste bloqué on attend */
263     }
264
265     if (strcmp(mots_pipe[1][0], "monexit") == 0){ /* si monexit*/
266         mon_exit_pipe(mots_pipe[1]);
267         exit(0); /* sinon on reste bloqué on attend */
268     }
269
270     else{
271         for(int i = 0; dirs[i] != 0; i++){
272             sprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[1][0]);
273             execv(pathname, mots_pipe[1]);
274         }
275
276         fprintf(stderr, "%s: notfound enfant 2\n", mots_pipe[1][0]);
277         exit(1) ;
278     }
279 }
280
/* PROCESSUS PARENT > FORK3*/
281 close(mes_pipes[0][0]); /* fermeture complète pipe1 */
282 close(mes_pipes[0][1]);
283
284 if ((tmp3= fork()) <0){

```

```

287     perror("fork3") ;
288     return -1;
289 }
290
291 /* PROCESSUS ENFANT 3 */
292 if (tmp3 == 0) {
293     dup2(mes_pipes[1][0], 0); /* redirection entrée depuis pipe2 */
294     close(mes_pipes[1][0]); /* fermeture pipe2 */
295     close(mes_pipes[1][1]);
296
297     redirection_dans_pipe(mots_pipe, 2);
298
299     if (strcmp(mots_pipe[2][0], "monman") == 0){ /* si monman*/
300         monman(mots_pipe[2]);
301         exit(0); /* sinon on reste bloqué on attend */
302     }
303
304     if (strcmp(mots_pipe[2][0], "moncd") == 0){ /* si moncd */
305         appel_mon_cd(dirs, mots_pipe[2], nb_mots_lastpipe);
306         exit(0); /* sinon on reste bloqué on attend */
307     }
308
309     if (strcmp(mots_pipe[2][0], "monexit") == 0){ /* si monexit */
310         mon_exit_pipe(mots_pipe[2]);
311         exit(0); /* sinon on reste bloqué on attend */
312     }
313
314     else{
315         for(int i = 0; dirs[i] != 0; i++){
316             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[2][0]);
317             execv(pathname, mots_pipe[2]);
318         }
319
320         fprintf(stderr, "%s: notfound enfant 3\n", mots_pipe[2][0]);
321         exit(1);
322     }
323 }
324
325 /* PROCESSUS PARENT FIN*/
326 close(mes_pipes[1][0]); /* fermeture pipe2 */
327 close(mes_pipes[1][1]);
328
329 if (attend_enfant == true){
330     while (wait(NULL) > 0) /* on attend tous les processus enfant */
331         ; /* se terminent */
332 }
333
334 *ilya_pipe = false; /* on marque bool "pipe" à faux */
335 return 0;
336
337 }

```

Les tests

Nous avons déjà illustré à plusieurs reprises le fonctionnement des pipes dans les autres sections de ce projet (notamment pour démontrer le fonctionnement de certaines commandes avec les pipes). Nous allons néanmoins proposer, de nouveau, certains exemples d'utilisation des pipes.

Dans un premier temps, nous utilisons deux pipes, avec les commandes `ls` et `wc`. Nous voyons que la commande a marché, avec le `wc` final. Ensuite, nous illustrons l'utilisation de deux pipes et des procédures en arrière-plan avec le `&`. Ça fonctionne également. Par ailleurs, on voit que lorsque `monexit` est le premier élément de la ligne de commande lorsqu'il y a des pipes, le processus parent n'est pas terminé. Cela correspond au comportement attendu.

On a illustré deux erreurs : lorsqu'il y a trois pipes, on n'obtient pas de sortie. C'est normal, notre programme ne fonctionne qu'avec deux pipes. Par ailleurs, on obtient une erreur lorsque la commande utilisée n'existe pas.

```

avrile@vmubuntu:~/Desktop/projet$ ./projet
? ls | ls | wc
      18      18     190
? echo "bonjour" | wc &
?           1      1     10
ls | moncd
? monexit | moncd
? ls | ls | monexit
Bye
? ls | ls > test4.txt
? ls | ls | ls | ls
? ls | ls | ls
and_monexit.c  LATEX_PROJET  moncd.c  proje
cn-decouper.c  Makefile      pipe.c   redir
f_head.h       man.c        projet   redir
? ls | inconnu
inconnu:notfound enfant 2
? pwx
pwx:notfound
? pwd
/home/avrile/Desktop/projet
? ls | wc | moncd /home/avrile
? ls | wc | moncd
? ls | wc | wc
      1      3     24
?

```

FIGURE 49 – Nous illustrons le fonctionnement des pipes au sein de notre shell avec des commandes qui fonctionnent mais également avec une commande qui cause une erreur.

Conclusion pour les pipes : Nous avons présenté l'utilisation des pipes. Nous avons prévu la gestion de un et de deux pipes sur la ligne de commande. Nous avons permis l'utilisation des pipes avec l'ensemble des autres fonctionnalités du projet.

Afin de gérer les pipes, nous devons diviser la ligne de commande en plusieurs commandes distinctes qui seront exécutées au sein de processus enfants lors de forks. Nous avons utilisé un tableau à deux dimensions pour isoler les différentes commandes. Nous avons abordé les différents points importants de notre code. Nous avons illustré grâce à des exemples.

1.13 L'auto-complétion des noms de fichiers

– Ajouter l'**auto-complétion** des noms de fichiers(*)

Pour ajouter l'autocomplétion des noms de fichiers et des chemins d'accès, nous avons utilisé la bibliothèque **Readline** GNU. Nous avons installé la bibliothèque grâce à la commande `sudo apt-get install libreadline-dev` (source : <https://askubuntu.com/questions/194523/how-do-i-install-gnu-readline>).

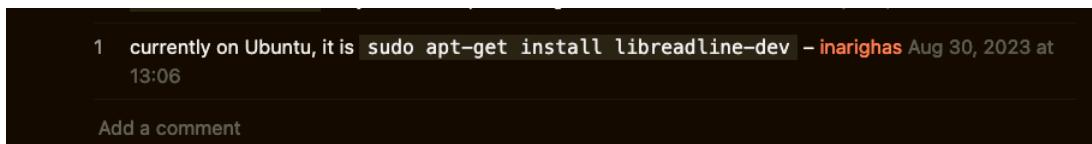


FIGURE 50 – Nous installons Readline avec la commande "sudo apt-get install libreadline-dev" (source : <https://askubuntu.com/questions/194523/how-do-i-install-gnu-readline>).

```
avrile@vmubuntu:~$ sudo apt-get install libreadline-dev
[sudo] password for avrile:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libncurses-dev
Suggested packages:
  ncurses-doc readline-doc
The following NEW packages will be installed:
  libncurses-dev libreadline-dev
0 upgraded, 2 newly installed, 0 to remove and 5 not upgraded.
Need to get 559 kB of archives.
After this operation, 3 253 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://mx.ports.ubuntu.com/ubuntu-ports jammy-updates/main arm64 libncurses-dev arm64 6.3-2ubuntu0.1 [381 kB]
Get:2 http://mx.ports.ubuntu.com/ubuntu-ports jammy/main arm64 libreadline-dev arm64 8.1.2-1 [178 kB]
Fetched 559 kB in 2s (346 kB/s)
Selecting previously unselected package libncurses-dev:arm64.
(Reading database ... 218389 files and directories currently installed.)
Preparing to unpack .../libncurses-dev_6.3-2ubuntu0.1_arm64.deb ...
Unpacking libncurses-dev:arm64 (6.3-2ubuntu0.1) ...
```

FIGURE 51 – L'installation de Readline sous Linux.

Afin de pouvoir utiliser Readline avec notre programme, nous avons dû modifier notre Makefile. Nous avons dû lier notre code à la bibliothèque Readline en passant un drapeau `-lreadline` au compilateur (source : https://en.wikipedia.org/wiki/GNU_Readline).

```
1 # Nom ..... : Makefile
2 # Rôle ..... : Compile projet
3 # Auteur ..... : Avrile Floro
4 # Version ..... : V0.1 du 25/10/23
5 # Licence ..... : réalisé dans le cadre du cours de SE
6 # Usage : Pour exécuter : make (depuis le répertoire)
7
8 all:
```

```

9  gcc -Wall projet_co-main.c and_monexit.c redirection.c redir_in_pipe.c cn-decouper.c moncd.c
    pipe.c man.c -lreadline -o projet
10
11 clean:
12   rm -f projet

```

Sample code [edit]

The following code is in **C** and must be linked against the readline library by passing a **-lreadline** flag to the compiler:

```

#include <stdlib.h>
#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>

```

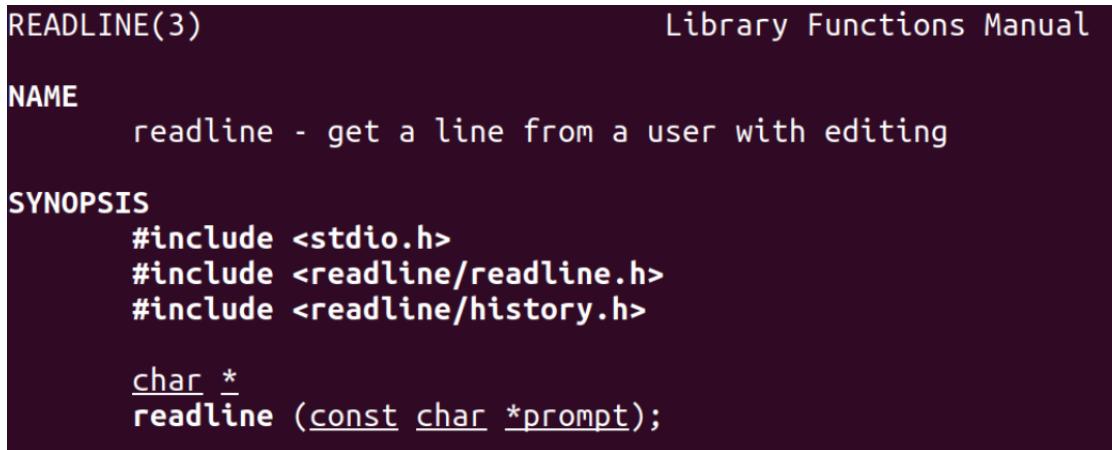
FIGURE 52 – On doit passer un drapeau "**-lreadline**" au compilateur pour pouvoir lier notre code à la bibliothèque Readline.

Pour pouvoir utiliser Readline, nous devons ajouter certaines bibliothèques à notre liste des bibliothèques usuelles. Cette liste se trouve dans le fichier **sys.h**. Les bibliothèques à ajouter sont mentionnées dans le manuel **man readline**.

```

19 #include <readline/readline.h>
20 #include <readline/history.h>

```



READLINE(3) Library Functions Manual

NAME
readline - get a line from a user with editing

SYNOPSIS

```

#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>

char *
readline (const char *prompt);

```

FIGURE 53 – On obtient le manuel de Readline avec la commande "man readline"

L'auto-complétion que nous souhaitons utiliser avec Readline est la plus simple, c'est-à-dire la compléition des noms de fichiers et des chemins d'accès.

Nous avons dû modifier légèrement notre code afin de pouvoir utiliser Readline. Au sein du fichier **projet_co-main.c** et de la fonction **main**, nous avons modifié **ligne**. Nous en avons fait un pointeur vers le premier caractère de la ligne (source : <https://thoughtbot.com/blog/tab-completion-in-gnu-readline>).

Par ailleurs, puisque la fonction **readline** (qui est celle que nous utilisons) prend en argument le **prompt**, nous avons pu simplifier la boucle **for** du prompt pour en faire un **while**. Nous avons associé à **ligne** le résultat de **readline(PROMPT)**, tant que ce résultat est différent de **NULL**. L'impression du prompt est dorénavant prise en charge par la fonction **readline**.

La compléition des noms de fichiers ou des chemins d'accès est très simple, il suffit d'appuyer sur la touche **Tab**.

```
27 int
28 main(int argc, char * argv[]) {
29     char* ligne; /* pointeur vers ligne pour Readline */
30
31     /* BOUCLE DU PROMPT */
32     while((ligne = readline(PROMPT)) != NULL) {
33         /* GESTION DES COMMANDES */
34         if(strstr(ligne, "exit") != NULL) {
35             /* Ferme la boucle */
36             break;
37         }
38
39         /* Affiche la ligne entrée */
40         printf("Ligne : %s\n", ligne);
41
42         /* GESTION DES AUTOCOMPTEURS */
43         if(strstr(ligne, "ls") != NULL) {
44             /* Affiche la liste des fichiers dans le répertoire courant */
45             /* Utilise la fonction systeme */
46             system("ls");
47         }
48
49         /* GESTION DES AUTOCOMPTEURS */
50         if(strstr(ligne, "cd") != NULL) {
51             /* Change de répertoire */
52             /* Utilise la fonction systeme */
53             system("cd");
54         }
55
56         /* GESTION DES AUTOCOMPTEURS */
57         if(strstr(ligne, "rm") != NULL) {
58             /* Supprime un fichier */
59             /* Utilise la fonction systeme */
60             system("rm");
61         }
62
63         /* GESTION DES AUTOCOMPTEURS */
64         if(strstr(ligne, "cp") != NULL) {
65             /* Copie un fichier */
66             /* Utilise la fonction systeme */
67             system("cp");
68         }
69
70         /* GESTION DES AUTOCOMPTEURS */
71         if(strstr(ligne, "mv") != NULL) {
72             /* Déplace un fichier */
73             /* Utilise la fonction systeme */
74             system("mv");
75         }
76
77         /* GESTION DES AUTOCOMPTEURS */
78         if(strstr(ligne, "cat") != NULL) {
79             /* Affiche le contenu d'un fichier */
80             /* Utilise la fonction systeme */
81             system("cat");
82         }
83
84         /* GESTION DES AUTOCOMPTEURS */
85         if(strstr(ligne, "grep") != NULL) {
86             /* Recherche une chaîne dans un fichier */
87             /* Utilise la fonction systeme */
88             system("grep");
89         }
90
91         /* GESTION DES AUTOCOMPTEURS */
92         if(strstr(ligne, "wc") != NULL) {
93             /* Affiche les statistiques d'un fichier */
94             /* Utilise la fonction systeme */
95             system("wc");
96         }
97
98         /* GESTION DES AUTOCOMPTEURS */
99         if(strstr(ligne, "head") != NULL) {
100            /* Affiche les premières lignes d'un fichier */
101            /* Utilise la fonction systeme */
102            system("head");
103        }
104
105        /* GESTION DES AUTOCOMPTEURS */
106        if(strstr(ligne, "tail") != NULL) {
107            /* Affiche les dernières lignes d'un fichier */
108            /* Utilise la fonction systeme */
109            system("tail");
110        }
111
112        /* GESTION DES AUTOCOMPTEURS */
113        if(strstr(ligne, "sort") != NULL) {
114            /* Trie les lignes d'un fichier */
115            /* Utilise la fonction systeme */
116            system("sort");
117        }
118
119        /* GESTION DES AUTOCOMPTEURS */
120        if(strstr(ligne, "uniq") != NULL) {
121            /* Supprime les lignes identiques d'un fichier */
122            /* Utilise la fonction systeme */
123            system("uniq");
124        }
125
126        /* GESTION DES AUTOCOMPTEURS */
127        if(strstr(ligne, "tr") != NULL) {
128            /* Remplace certaines chaînes dans un fichier */
129            /* Utilise la fonction systeme */
130            system("tr");
131        }
132
133        /* GESTION DES AUTOCOMPTEURS */
134        if(strstr(ligne, "cut") != NULL) {
135            /* Sépare les colonnes d'un fichier */
136            /* Utilise la fonction systeme */
137            system("cut");
138        }
139
140        /* GESTION DES AUTOCOMPTEURS */
141        if(strstr(ligne, "grep -v") != NULL) {
142            /* Recherche une chaîne dans un fichier et inverse le sens de la recherche */
143            /* Utilise la fonction systeme */
144            system("grep -v");
145        }
146
147        /* GESTION DES AUTOCOMPTEURS */
148        if(strstr(ligne, "grep -i") != NULL) {
149            /* Recherche une chaîne dans un fichier sans tenir compte de la casse */
150            /* Utilise la fonction systeme */
151            system("grep -i");
152        }
153
154        /* GESTION DES AUTOCOMPTEURS */
155        if(strstr(ligne, "grep -c") != NULL) {
156            /* Affiche le nombre de lignes correspondantes à la recherche */
157            /* Utilise la fonction systeme */
158            system("grep -c");
159        }
160
161        /* GESTION DES AUTOCOMPTEURS */
162        if(strstr(ligne, "grep -n") != NULL) {
163            /* Affiche les numéros des lignes correspondantes à la recherche */
164            /* Utilise la fonction systeme */
165            system("grep -n");
166        }
167
168        /* GESTION DES AUTOCOMPTEURS */
169        if(strstr(ligne, "grep -A") != NULL) {
170            /* Affiche les lignes suivantes de la recherche */
171            /* Utilise la fonction systeme */
172            system("grep -A");
173        }
174
175        /* GESTION DES AUTOCOMPTEURS */
176        if(strstr(ligne, "grep -B") != NULL) {
177            /* Affiche les lignes précédentes de la recherche */
178            /* Utilise la fonction systeme */
179            system("grep -B");
180        }
181
182        /* GESTION DES AUTOCOMPTEURS */
183        if(strstr(ligne, "grep -E") != NULL) {
184            /* Utilise des expressions régulières pour la recherche */
185            /* Utilise la fonction systeme */
186            system("grep -E");
187        }
188
189        /* GESTION DES AUTOCOMPTEURS */
190        if(strstr(ligne, "grep -F") != NULL) {
191            /* Utilise une chaîne fixe pour la recherche */
192            /* Utilise la fonction systeme */
193            system("grep -F");
194        }
195
196        /* GESTION DES AUTOCOMPTEURS */
197        if(strstr(ligne, "grep -l") != NULL) {
198            /* Affiche les noms des fichiers contenant la recherche */
199            /* Utilise la fonction systeme */
200            system("grep -l");
201        }
202
203        /* GESTION DES AUTOCOMPTEURS */
204        if(strstr(ligne, "grep -r") != NULL) {
205            /* Recherche une chaîne dans tous les fichiers d'un répertoire */
206            /* Utilise la fonction systeme */
207            system("grep -r");
208        }
209
210        /* GESTION DES AUTOCOMPTEURS */
211        if(strstr(ligne, "grep -R") != NULL) {
212            /* Recherche une chaîne dans tous les fichiers d'un répertoire et récursivement */
213            /* Utilise la fonction systeme */
214            system("grep -R");
215        }
216
217        /* GESTION DES AUTOCOMPTEURS */
218        if(strstr(ligne, "grep -q") != NULL) {
219            /* Ne pas afficher les résultats de la recherche */
220            /* Utilise la fonction systeme */
221            system("grep -q");
222        }
223
224        /* GESTION DES AUTOCOMPTEURS */
225        if(strstr(ligne, "grep -w") != NULL) {
226            /* Recherche une chaîne dans un fichier et ignore les espaces et les caractères spéciaux */
227            /* Utilise la fonction systeme */
228            system("grep -w");
229        }
230
231        /* GESTION DES AUTOCOMPTEURS */
232        if(strstr(ligne, "grep -z") != NULL) {
233            /* Recherche une chaîne dans un fichier et utilise un caractère de fin de chaîne nul */
234            /* Utilise la fonction systeme */
235            system("grep -z");
236        }
237
238        /* GESTION DES AUTOCOMPTEURS */
239        if(strstr(ligne, "grep -o") != NULL) {
240            /* Affiche uniquement les parties correspondantes à la recherche */
241            /* Utilise la fonction systeme */
242            system("grep -o");
243        }
244
245        /* GESTION DES AUTOCOMPTEURS */
246        if(strstr(ligne, "grep -P") != NULL) {
247            /* Utilise des expressions régulières avec des options spéciales */
248            /* Utilise la fonction systeme */
249            system("grep -P");
250        }
251
252        /* GESTION DES AUTOCOMPTEURS */
253        if(strstr(ligne, "grep -q") != NULL) {
254            /* Ne pas afficher les résultats de la recherche */
255            /* Utilise la fonction systeme */
256            system("grep -q");
257        }
258
259        /* GESTION DES AUTOCOMPTEURS */
260        if(strstr(ligne, "grep -w") != NULL) {
261            /* Recherche une chaîne dans un fichier et ignore les espaces et les caractères spéciaux */
262            /* Utilise la fonction systeme */
263            system("grep -w");
264        }
265
266        /* GESTION DES AUTOCOMPTEURS */
267        if(strstr(ligne, "grep -z") != NULL) {
268            /* Recherche une chaîne dans un fichier et utilise un caractère de fin de chaîne nul */
269            /* Utilise la fonction systeme */
270            system("grep -z");
271        }
272
273        /* GESTION DES AUTOCOMPTEURS */
274        if(strstr(ligne, "grep -o") != NULL) {
275            /* Affiche uniquement les parties correspondantes à la recherche */
276            /* Utilise la fonction systeme */
277            system("grep -o");
278        }
279
280        /* GESTION DES AUTOCOMPTEURS */
281        if(strstr(ligne, "grep -P") != NULL) {
282            /* Utilise des expressions régulières avec des options spéciales */
283            /* Utilise la fonction systeme */
284            system("grep -P");
285        }
286
287        /* GESTION DES AUTOCOMPTEURS */
288        if(strstr(ligne, "grep -q") != NULL) {
289            /* Ne pas afficher les résultats de la recherche */
290            /* Utilise la fonction systeme */
291            system("grep -q");
292        }
293
294        /* GESTION DES AUTOCOMPTEURS */
295        if(strstr(ligne, "grep -w") != NULL) {
296            /* Recherche une chaîne dans un fichier et ignore les espaces et les caractères spéciaux */
297            /* Utilise la fonction systeme */
298            system("grep -w");
299        }
299
300        /* GESTION DES AUTOCOMPTEURS */
301        if(strstr(ligne, "grep -z") != NULL) {
302            /* Recherche une chaîne dans un fichier et utilise un caractère de fin de chaîne nul */
303            /* Utilise la fonction systeme */
304            system("grep -z");
305        }
306
307        /* GESTION DES AUTOCOMPTEURS */
308        if(strstr(ligne, "grep -o") != NULL) {
309            /* Affiche uniquement les parties correspondantes à la recherche */
310            /* Utilise la fonction systeme */
311            system("grep -o");
312        }
313
314        /* GESTION DES AUTOCOMPTEURS */
315        if(strstr(ligne, "grep -P") != NULL) {
316            /* Utilise des expressions régulières avec des options spéciales */
317            /* Utilise la fonction systeme */
318            system("grep -P");
319        }
320
321        /* GESTION DES AUTOCOMPTEURS */
322        if(strstr(ligne, "grep -q") != NULL) {
323            /* Ne pas afficher les résultats de la recherche */
324            /* Utilise la fonction systeme */
325            system("grep -q");
326        }
327
328        /* GESTION DES AUTOCOMPTEURS */
329        if(strstr(ligne, "grep -w") != NULL) {
330            /* Recherche une chaîne dans un fichier et ignore les espaces et les caractères spéciaux */
331            /* Utilise la fonction systeme */
332            system("grep -w");
333        }
334
335        /* GESTION DES AUTOCOMPTEURS */
336        if(strstr(ligne, "grep -z") != NULL) {
337            /* Recherche une chaîne dans un fichier et utilise un caractère de fin de chaîne nul */
338            /* Utilise la fonction systeme */
339            system("grep -z");
340        }
341
342        /* GESTION DES AUTOCOMPTEURS */
343        if(strstr(ligne, "grep -o") != NULL) {
344            /* Affiche uniquement les parties correspondantes à la recherche */
345            /* Utilise la fonction systeme */
346            system("grep -o");
347        }
348
349        /* GESTION DES AUTOCOMPTEURS */
350        if(strstr(ligne, "grep -P") != NULL) {
351            /* Utilise des expressions régulières avec des options spéciales */
352            /* Utilise la fonction systeme */
353            system("grep -P");
354        }
355
356        /* GESTION DES AUTOCOMPTEURS */
357        if(strstr(ligne, "grep -q") != NULL) {
358            /* Ne pas afficher les résultats de la recherche */
359            /* Utilise la fonction systeme */
360            system("grep -q");
361        }
362    }
363 }
```

Les tests

Nous avons illustré l'utilisation de Readline qui permet la complétion des noms de fichiers et des chemins d'accès. Pour lancer notre shell, nous exécutons la commande `./projet`, puis employons `pwd` pour afficher le chemin d'accès courant. Ensuite, à l'aide de la commande `cat`, nous ouvrons un fichier. La complétion des noms de fichiers et des chemins par Readline s'active en appuyant sur la touche `Tab` après avoir entamé la saisie du nom. Si plusieurs répertoires correspondent, Readline présente les différentes options disponibles. Dès la saisie d'une première lettre, Readline s'active et suggère les possibles correspondances parmi les répertoires. L'auto-complétion fonctionne de manière similaire pour les noms de fichiers : dans un répertoire, si on appuie sur `Tab`, Readline affiche tous les fichiers présents. Ce mécanisme est également valable lorsqu'on entame le nom du fichier par une lettre. Si aucun fichier ou chemin correspondant n'est trouvé, Readline ne propose aucune suggestion.

```
avrile@vmubuntu:~/Desktop/projet$ ./projet
? pwd
/home/avrile/Desktop/projet
? cat /home/avrile/Desktop/
Desktop/  Documents/ Downloads/
? cat /home/avrile/Desktop/
old/  projet/
? cat /home/avrile/Desktop/projet/
cat: /home/avrile/Desktop/projet/: Is a directory
? cat /home/avrile/Desktop/projet/and_monexit.c
/* ****
# Nom ..... : and_monexit.c
# Rôle ..... : Gestion des processus en arrière-plan
#           : Gestion de mon_exit
# Auteur ..... : Avrile Floro
# Version ..... : V0.1 du 26/10/2023
# Licence ..... : réalisé dans le cadre du cours de SE
# Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
****/
```

FIGURE 54 – Nous avons illustré l'utilisation de Readline.

```
? cat /home/avrile/Desktop/projet/  
.DS_Store      cn-decouper.c    pipe.c        redirection.c   test3.txt  
LATEX_PROJET/  f_head.h       projet        sys.h         test4.txt  
Makefile       man.c          projet_co-main.c test1.txt     test5.txt  
and_monexit.c moncd.c       redir_in_pipe.c  test2.txt  
? cat /home/avrile/Desktop/projet/p  
pipe.c          projet        projet_co-main.c  
? cat /home/avrile/Desktop/projet/pipe.c
```

FIGURE 55 – Readline fonctionne également pour la complétiōn des noms de fichier.

Conclusion pour Readline : Nous avons installé Readline sur Ubuntu. Nous avons ajouté les bibliothèques de Readline à notre fichier `sys.h`. Nous avons passé un drapeau à notre compilateur afin de pouvoir utiliser les fonctionnalités de Readline. En outre, nous avons légèrement modifié notre code pour implémenter la complétion des chemins d'accès et des noms de fichiers grâce à Readline.

1.14 Le manuel monman

- La documentation des commandes et accessible par MAN.

Afin d'inclure la documentation des commandes, nous avons décidé de créer un manuel que nous avons appelé `monman`. Pour conserver une bonne lisibilité du projet, nous avons dédié un fichier séparé au manuel, que nous avons appelé `man.c`.

Le fichier `man.c` contient la fonction `monman` qui est en charge du manuel pour les commandes. On vérifie avant l'appel de la fonction `monman` que la première commande sur la ligne de commande correspond à `monman`. Néanmoins, par sécurité, nous avons décidé d'intégrer cette vérification également à l'intérieur de la fonction. Cela fait un peu double usage mais pourrait permettre au code de demeurer plus facilement maintenable. Ainsi, la fonction `monman` commence par vérifier si le premier élément de la commande est `monman`.

Si c'est le cas, il y a plusieurs possibilités, définies grâce à des `if`. Si la commande `monman` n'a pas d'argument, alors la fonction renvoie un message demandant à quelle page du manuel l'utilisateur souhaite accéder, à l'image du retour du vrai `man` sous Bash. Par ailleurs, si la commande `monman` a deux arguments ou plus, alors la fonction renvoie un `usage`.

Finalement, si la commande `monman` a reçu un seul argument, la fonction va renvoyer le manuel. Nous avons préparé la commande `monman monman` : il s'agit du manuel du manuel. Nous avons également préparé le manuel pour la commande `monexit`, qui peut être appelé avec `monman monexit` (source : <https://man7.org/linux/man-pages/man1/exit.1p.html>), ainsi que le manuel pour la commande `moncd`, qui peut être appelé avec `monman moncd` (source : <https://man7.org/linux/man-pages/man1/cd.1p.html>). Enfin, nous avons préparé le cas dans lequel l'argument passé à la commande `monman` n'est pas reconnu.

```
1 /* # Nom ..... : man.c
2 # Rôle ..... : Manuel du shell
3 # Auteur ..... : Avrile Floro
4 # Version ..... : V0.1 du 30/01/2024
5 # Licence ..... : réalisé dans le cadre du cours de SE
6 # Usage : Compilé via make */
7
8
9 #include "sys.h"
10 #include "f_head.h"
11
12 int monman(char* mot[]){
13     if (strcmp(mot[0], "monman") == 0){ /* si commence par mon man */
14         if (mot[1] == 0){ /* si pas d'argument */
15             fprintf(stderr, "Quelle page du manuel voulez-vous?\nPar exemple, essayer 'monman
16 monman'.\n");
17         }
18
19         else if (mot[2] != NULL){ /* si deux arguments */
20             fprintf(stderr, "Usage: monman %s\n", mot[1]);
21         }
22
23         else if (strcmp(mot[1], "monman") == 0){ // man man
24             printf("\nNOM\nmonman - une interface pour le manuel de référence du système\n
25 DESCRIPTION\nmonman est le pagineur de manuel du système. Chaque argument donné à monman
26 est le nom d'une commande créée pour ce shell élémentaire. La page de manuel associée à
27 son argument est alors trouvée et affichée.\n\nEXEMPLES D'UTILISATION\nmonman monman\
28 monman moncd\nmonman monexit\n\n");
29         }
30
31         else if (strcmp(mot[1], "monexit") == 0){
32             printf("\nNOM\nmonexit - provoque la sortie du shell\n
33 DESCRIPTION\nLa commande
34 monexit permet au shell de sortir de son environnement d'exécution actuel. La commande
35 monexit ne prend pas d'option.\n\nEXEMPLE D'UTILISATION\nmonexit\n\n");
36         }
37
38         else if (strcmp(mot[1], "moncd") == 0){
39             printf("\nNOM\nmoncd - changer le répertoire de travail\n
40 DESCRIPTION\nL'utilitaire
41 moncd change le répertoire de travail de l'environnement d'exécution du shell actuel en
42 exécutant les étapes suivantes dans l'ordre.\n      1. Si un répertoire est donné en
43 argument à la commande 'moncd', alors le répertoire de travail du shell sera modifié pour
```

```

correspondre au chemin spécifié en argument.\n      2. Si aucun répertoire n'est spécifié
en argument, 'moncd' se comportera comme si le chemin spécifié dans la variable d'
environnement HOME était l'argument. Cela signifie que le répertoire de travail sera changé pour celui défini dans HOME, à condition que cette variable ne soit ni vide ni indéfinie
. \n\nEXEMPLES D'UTILISATION\nmoncd /home/avrile/Desktop\n\n");
32     }
33
34     else{ /* si l'argument passé n'existe pas */
35         fprintf(stderr, "Cette commande n'existe pas.\n");
36     }
37 }
38
39 return 0;
40 }
```

La fonction `monman` est appelée au sein de la fonction `main` (fichier `projet_co-main.c`). Au sein de la boucle du prompt, avant le `fork`, si le booléen `ilya_pipe` est faux et que le premier mot de la commande est `monman`, alors on appelle la fonction `monman`.

```

27 int
28 main(int argc, char * argv[]){
48     bool ilya_pipe = false ;
55     /* BOUCLE DU PROMPT */
56     while((ligne = readline(PROMPT)) != NULL){
67         int nb_pipe_total = cb_pipe(mot, nb_total_mots, &ilya_pipe);
68         /* renvoie nb total de pipes | + permet passage ilya_pipe à vrai */
88         /* MONMAN */
89         if ((!ilya_pipe)&&(strcmp(mot[0], "monman") == 0)){
90             monman(mot);
91         }
99         /* FORK INITIAL SI PAS DE PIPE */
100        else {
102            tmp = fork(); /* lancement du processus enfant */
103        }
104    }
105 }
```

Comme pour les autres commandes, la gestion de la commande `monman` est différente selon qu'on utilise des pipes ou non. Lorsqu'il y a des pipes sur la ligne de commande, la gestion du manuel se fait directement au sein des fonctions en charge des pipes, c'est-à-dire `un_pipe` et `deux_pipes`. Ces fonctions se trouvent dans le fichier `pipe.c`.

Pour chaque processus enfant (il y en a deux lorsqu'il y a un pipe et il y en a trois lorsqu'il y a deux pipes), on dirige avec des `if` le comportement du programme. C'est-à-dire que si le premier élément de la commande est `monman`, alors la fonction `monman` est appelée et l'`execv` ne sera ensuite pas exécuté.

```

186 int deux_pipes(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe){
196     if ((tmp = fork()) <0){
197         perror("fork") ; /* vérification réussite fork1 */
198         return -1; }
199
200     /* PROCESSUS ENFANT 1 */
201
202     dup2(mes_pipes[0][1], 1); /* redirection sortie */
203     close(mes_pipes[0][0]); /* fermeture de tous les pipes */
204     close(mes_pipes[0][1]);
205     close(mes_pipes[1][0]);
206     close(mes_pipes[1][1]);
207
208     int nb_elt_tab1 = nb_mots(mots_pipe, 0);
209
210     redirection_dans_pipe(mots_pipe, 0);
211 }
```

```

212     if (strcmp(mots_pipe[0][0], "monman") == 0){ /* si monman*/
213         monman(mots_pipe[0]);
214         exit(0); /* sinon on reste bloqué on attend */
215     }
216
217
218     else{
219         for(int i = 0; dirs[i] != 0; i++){
220             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
221             execv(pathname, mots_pipe[0]);
222         }
223
224         fprintf(stderr, "%s: not found enfant 1\n", mots_pipe[0][0]);
225         exit(1) ;
226     }
227 }

```

Les tests

Nous allons illustrer le fonctionnement de notre commande `monman`, notamment avec des pipes.

Nous lançons notre shell avec la commande `./projet`, puis nous lançons la commande `monman`, sans argument. Nous obtenons un message nous invitant à préciser la page recherchée, aussi bien lorsque la commande est utilisée avec un pipe que sans. En outre, si nous passons plus d'un argument à la commande, nous obtenons un `Usage`.

Nous appelons la commande `monman moncd` et nous obtenons le manuel de la commande `moncd`. Cependant, lorsque nous essayons d'utiliser la commande `monman inconnu`, nous obtenons une erreur nous précisant que cette commande n'existe pas. Grâce à la commande `monman monexit`, nous obtenons le manuel de la commande `monexit`. Grâce à la commande `monman monman`, nous obtenons le manuel du manuel.

```

avrile@vmubuntu:~/Desktop/projet$ ./projet
? monman
Quelle page du manuel voulez-vous?
Par exemple, essayer 'monman monman'.
? echo "bonjour" | monman
Quelle page du manuel voulez-vous?
Par exemple, essayer 'monman monman'.
? echo "bonjour" | monman arg1 arg2
Usage: monman arg1
? ls | monman moncd

NOM
moncd - changer le répertoire de travail

DESCRIPTION
L'utilitaire moncd change le répertoire de travail de l'environnement d'exécution du shell actuel en exécutant les étapes suivantes dans l'ordre.
    1. Si un répertoire est donné en argument à la commande `moncd`, alors le répertoire de travail du shell sera modifié pour correspondre au chemin spécifié en argument.
    2. Si aucun répertoire n'est spécifié en argument, `moncd` se comportera comme si le chemin spécifié dans la variable d'environnement HOME était l'argument. Cela signifie que le répertoire de travail sera changé pour celui défini dans HOME, à condition que cette variable ne soit ni vide ni indéfinie.

EXEMPLES D'UTILISATION
moncd
moncd /home/avrile/Desktop

? monexit | monman inconnu
Cette commande n'existe pas.
?
```

FIGURE 56 – Nous illustrons le fonctionnement de la commande "monman", notamment avec des pipes.

```
? monman monexit
NOM
monexit - provoque la sortie du shell

DESCRIPTION
La commande monexit permet au shell de sortir de son environnement d'exécution actuel. La commande monexit ne prend pas d'option.

EXEMPLE D'UTILISATION
monexit

? monman
Quelle page du manuel voulez-vous?
Par exemple, essayer 'monman monman'.
? monman monman

NOM
monman - une interface pour le manuel de référence du système

DESCRIPTION
monman est le pagineur de manuel du système. Chaque argument donné à monman est le nom d'une commande créée pour ce shell élémentaire. La page de manuel associée à son argument est alors trouvée et affichée.

EXEMPLES D'UTILISATION
monman monman
monman moncd
monman monexit

? monman monman1 monman2
Usage: monman monman1
? █
```

FIGURE 57 – Nous illustrons le fonctionnement de la commande "monman".

Conclusion de la commande monman : Nous avons présenté la commande `monman` qui sert de manuel pour les commandes `moncd`, `monexit` et `monman`. Nous avons présenté le fonctionnement de la commande `monman`, notamment avec les pipes. Nous avons illustré le fonctionnement de la commande avec des exemples réussis et avec des erreurs.

1.15 Difficultés rencontrées

Avant de commencer ce cours, nous avions un peu d’appréhension, principalement à cause de nos lacunes en C. Cependant, grâce à un projet progressif clairement articulé, nous avons non seulement réussi à développer un shell en C, mais nous estimons également avoir nettement amélioré nos compétences en C (bien qu’il nous reste énormément de progrès à faire).

Les défis majeurs se sont manifestés à des moments clés du projet : lors de l’implémentation du fork dans le chapitre 2, de la mise en place des redirections dans le chapitre 5, et enfin de la création des pipes dans le chapitre 6. Notre principale difficulté résidait dans notre incapacité à conceptualiser comment ces fonctionnalités pourraient être implémentées. Nous manquions de vision.

Tout au long du projet, nous avons également dû faire face à divers bugs. Bien que souvent mineurs, ils se révélaient frustrants, provoquant par exemple des erreurs de segmentation. Il semble que ce soit une composante habituelle du développement, mais cela nous a également enseigné l’importance de la persévérance et de l’attention aux détails dans la résolution de problèmes.

1.16 Conclusions et perspectives

L’ajout des pipes vers la fin du cours a représenté l’intégration la plus difficile, nécessitant une révision complète des fonctionnalités existantes pour les adapter à ce cas de figure. Il est probable qu’une approche plus optimisée puisse être envisagée à cet égard. Une évolution intéressante serait de permettre l’utilisation d’un nombre illimité de pipes, alors que notre gestion actuelle se limite à deux.

La construction progressive de notre shell a été une expérience enrichissante, nous démontrant l’importance de segmenter les projets informatiques en phases gérables.

Contrairement à nos appréhensions initiales, le projet s’est révélé être moins intimidant que prévu, grâce à la structure bien pensée du cours. Ce travail nous a permis de renforcer nos compétences en C. Nous nous sentons désormais plus confiante dans l’usage de ce langage.

Notre projet final présente un shell codé en C, intégrant les commandes `moncd` et `monexit`, la gestion des processus en arrière-plan, les redirections d’entrées-sorties, les pipes, l’auto-complétion des noms de fichiers, ainsi qu’un manuel utilisateur `monman`. Chaque fonctionnalité a été détaillée avec des exemples et des explications sur les erreurs potentielles. Un Makefile, exécutable via la commande `make`, a également été préparé pour faciliter l’utilisation et la compilation du projet.

Annexes

.1 Le Makefile en intégralité

Voici l'intégralité du Makefile.

```
1 # Nom ..... : Makefile
2 # Rôle ..... : Compile projet
3 # Auteur ..... : Avrile Floro
4 # Version ..... : V0.1 du 25/10/23
5 # Licence ..... : réalisé dans le cadre du cours de SE
6 # Usage : Pour exécuter : make (depuis le répertoire)
7
8 all:
9     gcc -Wall projet_co-main.c and_monexit.c redirection.c redir_in_pipe.c cn-decouper.c moncd.c
10    pipe.c man.c -lreadline -o projet
11
12 clean:
13     rm -f projet
```

.2 Le fichier "sys.h"

Voici l'intégralité du fichier `sys.h`, utilisé dans l'ensemble des programmes C.

```
1  /**# Nom ..... : sys.h
2  # Rôle ..... : les bibliothèques couramment utilisées
3  # Auteur ..... : Avrile Floro
4  # Version ..... : V0.1 du 30/01/2024
5  # Licence ..... : réalisé dans le cadre du cours de SE */
6
7
8 #include <assert.h>
9 #include <string.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <unistd.h>
13 #include <sys/stat.h>
14 #include <sys/types.h>
15 #include <fcntl.h>
16 #include <sys/wait.h>
17 #include <stdbool.h>
18 #include <errno.h>
19 #include <readline/readline.h>
20 #include <readline/history.h>
```

.3 Le fichier "f_head.h"

Voici l'intégralité du fichier `f_head.h`, utilisé dans l'ensemble des programmes C.

```
1  /* ****
2 # Nom ..... : f_head.h
3 # Rôle ..... : Les déclarations de fonctions
4 # Auteur ..... : Avrile Floro
5 # Version ..... : V0.1 du 02/02/2024
6 # Licence ..... : réalisé dans le cadre du cours de SE
7 *****/
8
9 enum {
10 MaxLigne = 1024, // longueur max d'une ligne de commandes
11 MaxMot = MaxLigne / 2, // nbre max de mot dans la ligne
12 MaxDirs = 100, // nbre max de repertoire dans PATH MaxPathLength = 512,
13 MaxPathLength = 512, // longueur max d'un nom de fichier
14 MaxPipes = 3, // nb de pipes maximum
15 };
16
17
18 int decouper(char *, char *, char **, int);
19
20 /* & */
21 int arriere_plan(int* nb_total_mots, char* mot[], bool* attend_enfant);
22
23 /* MONCD */
24 int appelle_mon_cd(char *dir[], char *mot[], int* nb_total_mots);
25 int mon_cd(char *dir[]);
26
27 /* MONEXIT */
28 int mon_exit(char *mot[]);
29
30 int redirection(char** mot, int* nb_total_mots, bool* redir_sortie, bool* stderr_sortie, bool*
   redir_s_entree, int* old_entree, int* old_sortie, int* old_erreur, bool* redir, bool*
   reset);
31
32 int gestion_pipes (int nb_pipe_total, int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot],
   int mes_pipes[][2], char* dirs[], char pathname[], int pathname_size, bool
   attend_enfant, bool* ilya_pipe);
33
34 int cb_pipe(char* mot[], int nb_total_mots, bool* ilya_pipe);
35
36 int monman(char* mot[]);
37
38 int div_tab(int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot], int mes_pipes[][2], int
   * nb_mots_lastpipe);
39
40 int un_pipe(int mes_pipes[][2],char pathname[], int pathname_size, char* dirs[], char*
   mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe);
41
42 int deux_pipes(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
   mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe);
43
44 int redirection_dans_pipe(char* mots_pipe[][MaxMot], int ligne);
45
46 int mon_exit_pipe(char *mot[]);
47
48
49
50 /* CRÉATION REDIRECTIONS*/
51 /* > */
52 int simple_sortie(char** mot, int* nb_total_mots, mode_t mode, bool* redir_sortie, int i) ;
53
54 /* >> */
55 int double_sortie (char** mot, int* nb_total_mots, bool* redir_sortie, mode_t mode, int i) ;
56
57 /* 2> */
58 int stderr_simple_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
   int i);
59
60 /* 2>> */
61 int stderr_double_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
```

```

63     int i);
64
65     /* >& */
66     int stderr_stdout_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, bool*
67     redir_sortie, mode_t mode, int i);
68
69     /* < */
70     int simple_entree(char** mot, int* nb_total_mots, bool* redir_s_entree, mode_t mode, int i) ;
71
72     /* RÉINITIALISATION*/
73
74     /* réinitialisation après > et >> */
75     int reset_sortie(bool* redir_sortie, int* old_sortie);
76
77     /* réinitialisation après 2> et 2>> */
78     int reset_stderr(bool* stderr_sortie, int* old_stderr);
79
80     /* réinitialisation après < */
81     int reset_stdin(bool* redir_s_entree, int* old_entree);
82
83     /* CRÉATION REDIRECTIONS POUR LES PIPES */
84     /* > */
85     int simple_sortie_pipe(char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i) ;
86
87     /* >> */
88     int double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i) ;
89
90     /* 2> */
91     int stderr_simple_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i);
92
93     /* 2>> */
94     int stderr_double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i);
95
96     /* >& */
97     int stderr_stdout_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i);
98
99     /* < */
100    int simple_entree_pipe(char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i) ;
101
102    /* MANUEL */
103    int monman(char* mot[]);
104
105    /* calcul nb mots dans pipes */
106    int nb_mots(char* mots_pipe[][512], int ligne);

```

.4 Le fichier "projet_co-main.c" en intégralité

Voici l'intégralité du fichier projet_co-main.c.

```
1  /* ****
2 # Nom ..... : projet_co-main.c
3 # Rôle ..... : Main du shell simple du cours
4 #           + gestion du "&" pour lancer un processus en arrière plan
5 #           + gestion de la commande "moncd"
6 #           + gestion de la commande "monexit"
7 #           + gestion des redirections: >, >>, 2>, 2>>, >&, <
8 #           + gestion des pipes |, ||
9 #           + manuel "monman" [monman monman, monman moncd, monman monexit]
10 #           + auto-complétion des noms de fichiers (appui sur tab)
11 # Auteur ..... : Avrile Floro
12 # Version ..... : V0.1 du 30/01/2024
13 # Licence ..... : réalisé dans le cadre du cours de SE
14 # Compilation : gcc -Wall -g projet_co-main.c
15 #                 cn-decouper.c moncd.c redirection.c man.c -lreadline -o projet
16 #                 un makefile a été préaré > make
17 # Usage : Pour exécuter : ./projet
18 #           puis "monexit" pour quitter
19 #*****
20
21 #include "sys.h"
22 #include "f_head.h"
23
24
25 #define PROMPT "? "
26
27 int
28 main(int argc, char * argv[]){
29     char* ligne; /* pointeur vers ligne pour Readline */
30     char pathname[MaxPathLength]; char * mot[MaxMot];
31     char * dirs [MaxDirs];
32     int i , tmp;
33     int mon_cdrom, out; // pour moncd et monexit
34
35     bool redir = false ; /* si on veut créer une redirection */
36     bool reset = false ; /* si on veut réinitialiser les entrées-sorties */
37
38     bool redir_sortie = false ; /* > et >> et >& */
39     bool stderr_sortie = false ; /* 2> et 2 >> et >& */
40     bool redir_s_entree = false ; /* < */
41
42     int old_entree = dup(0); /* copie de stdin initiale */
43     int old_sortie = dup(1); /* copie de stdout initiale */
44     int old_erreur = dup(2); /* copie la sortie stderr initiale */
45
46     int mes_pipes[MaxPipes][2]; /* tableaux pour les FD des pipes */
47
48     bool ilya_pipe = false ;
49
50     char * mots_pipe[MaxPipes][MaxMot]; /* sous-tab de mots pour c/ côté de pipe*/
51
52     /* Decouper UNE COPIE de PATH en repertoires */
53     decouper(strdup(getenv("PATH")), ":", dirs, MaxDirs);
54
55     /* BOUCLE DU PROMPT */
56     while((ligne = readline(PROMPT)) != NULL){
57
58         bool attend_enfant = true;
59         /* réiniti pour c/ boucle: par défaut, on attend le processus enfant */
60
61         int nb_total_mots = decouper(ligne, "\t\n", mot, MaxMot);
62         /* on récupère le nb total de mot */
63
64         /* ARRIÉRE-PLAN & */
65         arriere_plan(&nb_total_mots, mot, &attend_enfant);
66
67         int nb_pipe_total = cb_pipe(mot, nb_total_mots, &ilya_pipe);
68         /* renvoie nb total de pipes | + permet passage ilya_pipe à vrai */
69
70         /* LES REDIRECTIONS */
```

```

71     if (!ilya_pipe){
72         redir = true ;
73         redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &redir_s_entree, &
74         old_entree, &old_sortie, &old_erreur,&redir, &reset);
75     }
76
77     /* LIGNE VIDE */
78     if (mot[0] == 0)
79         continue;
80
81     /* MONCD */
82     if ((!ilya_pipe)&&((mon_cdrom = appel_mon_cd(dirs, mot, &nb_total_mots)) == 0 ||
83     mon_cdrom == 1))
84         continue;
85
86     /* MON_EXIT */
87     if ((!ilya_pipe)&&(out = mon_exit(mot)) == -1)
88         perror("mon_exit");
89
90     /* MONMAN */
91     if ((!ilya_pipe)&&(strcmp(mot[0], "monman") == 0)){
92         monman(mot);
93     }
94
95     /* LES PIPES */
96     else if (ilya_pipe){
97         gestion_pipes(nb_pipe_total, nb_total_mots, mot, mots_pipe, mes_pipes, dirs,
98         pathname, sizeof(pathname), attend_enfant, &ilya_pipe) ;
99     }
100
101
102     /* FORK INITIAL SI PAS DE PIPE */
103     else {
104
105         tmp = fork(); /* lancement du processus enfant */
106
107         if (tmp<0){ /* erreur tmp */
108             perror("fork") ;
109             continue;
110         }
111
112         /* PROCESSUS PARENT */
113         else if (tmp > 0){
114             /* si on attend le processus enfant */
115             if (attend_enfant == true){
116                 while (wait(NULL) > 0)
117                     ;
118             }
119             /* fin du processus enfant */
120
121             /* RÉINITIALISATION STDIN, STDOUT, STDERR */
122             reset = true;
123             redirection(mot, &nb_total_mots, &redir_sortie, &stderr_sortie, &
124             redir_s_entree, &old_entree, &old_sortie, &old_erreur,&redir, &reset);
125
126             /* si on n'attend pas le processus enfant,
127             ou que le processus enfant est déjà terminé on continue */
128             continue;
129         }
130
131         /* PROCESSUS ENFANT */
132         else { /* tmp = 0 */
133
134             for(i = 0; dirs[i] != 0; i++){
135                 snprintf(pathname, sizeof pathname, "%s/%s", dirs[i], mot[0]);
136                 execv(pathname, mot); /* exécution du programme */
137             }
138
139             /* aucun exec n'a fonctionné */
140             fprintf(stderr, "%s: notfound\n",mot[0]);
141             exit(1) ;
142         }
143     }
144
145     printf ("Bye\n");

```

```
143     return 0;
144
145 }
```

.5 Le fichier "cn-decouper" en intégralité

Voici l'intégralité du fichier cn-decouper.c.

```
1  /* ****
2 # Nom ..... : cn-decouper.c
3 # Rôle ..... : découper ligne en token
4 # Auteur ..... : tiré du cours
5 # Licence ..... : réalisé dans le cadre du cours de SE
6 # Compilation : compilé avec les autres programmes
7 #*****
8
9 /* cn=decouper.c
10 Un wrapper autour de strtok
11 */
12
13 # include <stdio.h>
14 # include <string.h>
15
16 /* decouper == decouper une chaine en mots */
17
18 int decouper(char * ligne, char * separ, char * mot[], int maxmot) {
19     int i;
20
21     mot[0] = strtok(ligne, separ);
22     for(i = 1; mot[i - 1] != 0; i++) {
23         if (i == maxmot) {
24             fprintf(stderr, "Erreur dans la fonction decouper: trop de mots\n");
25             mot[i - 1] = 0;
26             break;
27         }
28         mot[i] = strtok(NULL, separ);
29     }
30     return i-1; /* = nb total mots */
31 }
32
33 # ifdef TEST
34
35 int
36 main(int ac, char * av[]) {
37     char *elem[10];
38     int i;
39
40     if (ac<3){
41         fprintf(stderr, "usage:%sphrasesepar\n",av[0]);
42         return 1;
43     }
44
45     printf("On decoupe '%s' suivantes '%s' :\n", av[1], av[2]);
46     decouper(av[1],av[2], elem,10);
47
48     for(i =0; elem[i] != 0; i++)
49         printf("%d:%s\n",i, elem[i]);
50
51     return 0;
52 }
53
54 # endif
```

.6 Le fichier "pipe.c" en intégralité

Voici l'intégralité du fichier `pipe.c`:

```

1  /* ****
2 # Nom ..... : pipe.c
3 # Rôle ..... : Gestion des pipes : | et |
4 # Auteur ..... : Avrile Floro
5 # Version ..... : V0.1 du 26/10/2023
6 # Licence ..... : réalisé dans le cadre du cours de SE
7 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
8 *****/
9
10 #include "sys.h"
11 #include "f_head.h"
12
13
14 int cb_pipe(char* mot[], int nb_total_mots, bool* ilya_pipe)
15 { /* compte le nb de signe "pipe" | dans la ldc */
16     int c = 0;
17     for (int i = 0; i < nb_total_mots - 1 ; i++) {
18         if (strcmp(mot[i], "|") == 0) /* à chaque pipe re ontré */
19         {
20             c++;
21             *ilya_pipe= true;
22         }
23     }
24
25 }
26 return c; /* renvoie le nb de pipes */
27 }
28
29 int gestion_pipes (int nb_pipe_total, int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot],
30                     int mes_pipes[][2], char* dirs[], char pathname[], int pathname_size, bool
31                     attend_enfant, bool* ilya_pipe){
32
33     int* nb_mots_lastpipe = malloc(sizeof(int));
34     *nb_mots_lastpipe = 0; /* pour calculer longueur pipes */
35
36     if (nb_pipe_total > 0) /* qu'il y ait | ou || */
37         div_tab(nb_total_mots, mot, mots_pipe, mes_pipes, nb_mots_lastpipe);
38         /* on remplit les sous tableau de chaque commande pour les pipes */
39
40     if (nb_pipe_total ==1){ /* s'il y a un | */
41         int v = un_pipe(mes_pipes, pathname, pathname_size, dirs, mots_pipe, attend_enfant,
42                         ilya_pipe, nb_mots_lastpipe);
43         if (v == 0 || v == -1)
44             return 0;
45     }
46
47     if (nb_pipe_total==2){ /* s'il y a deux || */
48         int v = deux_pipes(mes_pipes, pathname, pathname_size, dirs, mots_pipe, attend_enfant,
49                         ilya_pipe, nb_mots_lastpipe);
50         if (v == 0 || v == -1)
51             return 0;
52     }
53     return 0;
54 }
55
56
57 int div_tab(int nb_total_mots, char* mot[], char* mots_pipe[][MaxMot], int mes_pipes[][2],
58             int * nb_mots_lastpipe){
59
60     /* permet de remplir les sous-tableaux [lignes] correspondant aux commandes des pipes */
61
62     int ligne = 0; /* on commence au 1er sous-tableau */
63     int mot_actuel=0; /* on commence au premier mot */
64     for (int i = 0 ; i < nb_total_mots ; i++){
65
66         if (strcmp(mot[i], "|") == 0){ /* si le caractère est un | */
67             mots_pipe[ligne][mot_actuel] = NULL ; /* on le remplace par NULL */
68             mot_actuel = 0; /* on recommencera à compter à 0 dans le prochain sous-tableau */
69             (*nb_mots_lastpipe)=0; /* init nb laspipe */
70             ligne++; /* changent de sous-tableau car rencontré un pipe */
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
99
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114
115
116
117
118
119
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
188
189
189
190
191
192
193
194
195
196
197
198
199
199
200
201
202
203
204
205
206
207
208
209
209
210
211
212
213
214
215
216
217
218
219
219
220
221
222
223
224
225
226
227
228
229
229
230
231
232
233
234
235
236
237
238
239
239
240
241
242
243
244
245
246
247
247
248
249
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
995
996
997
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1395
1396
1397
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1495
1496
1497
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1677
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1689
1689
1690
1691
1692
1693
1694
1695
1695
1696
1697
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322

```

```

66
67
68     }
69
70     else { /* si le caractère est pas un | */
71         mots_pipe[ligne][mot_actuel] = mot[i]; /* j'écris le mot dans le tableau */
72         mot_actuel++; /* je parcours les mots */
73         (*nb_mots_lastpipe)++; /* on incrémente */
74     }
75
76     mots_pipe[ligne][mot_actuel] = NULL; /* on termine le tableau avec NULL */
77
78     return 0;
79 }
80
81 int un_pipe(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
82             mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe){
83
84     int tmp, tmp2;
85
86     if (pipe(mes_pipes[0]) < 0){ /* vérification erreur pipe */
87         perror("pipe");
88         return -1; }
89
90
91     /* PROCESSUS PARENT: FORK1 */
92     if ((tmp= fork())<0){
93         perror("fork");
94         return -1; }
95
96
97     /* PROCESSUS ENFANT 1 */
98     if (tmp ==0) {
99         dup2(mes_pipes[0][1], 1); /* redirection sortie vers pipe */
100        close(mes_pipes[0][0]); /* fermeture pipe */
101        close(mes_pipes[0][1]);
102
103        int nb_elt_tab1 = nb_mots(mots_pipe, 0); /* compte les mots du pipe */
104
105        redirection_dans_pipe(mots_pipe, 0);
106
107        if (strcmp(mots_pipe[0][0], "monman") == 0){ /*si monman*/
108            monman(mots_pipe[0]);
109            exit(0); /* sinon on reste bloqué on attend */
110        }
111
112        if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
113            appel_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
114            exit(0); /* sinon on reste bloqué on attend */
115        }
116
117        if (strcmp(mots_pipe[0][0], "monexit") == 0){ /* si monexit*/
118            mon_exit_pipe(mots_pipe[0]);
119            exit(0); /* sinon on reste bloqué on attend */
120        }
121
122        else{
123            for(int i = 0; dirs[i] != 0; i++){
124                snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
125                execv(pathname, mots_pipe[0]);
126            }
127
128            fprintf(stderr, "%s: notfound enfant 1\n", mots_pipe[0][0]);
129            exit(1);
130        }
131    }
132
133
134    /* PROCESSUS PARENT > FORK 2 */
135    if ((tmp2=fork()) <0){
136        perror("fork");
137        return -1;
138    }
139
140    /* PROCESSUS ENFANT 2 */
141    if (tmp2 ==0) {
142        dup2(mes_pipes[0][0], 0); /* redirection entrée depuis pipe */
143        close(mes_pipes[0][0]); /* fermeture pipe */
144        close(mes_pipes[0][1]);
145
146        redirection_dans_pipe(mots_pipe, 1);

```

```

141
142     if (strcmp(mots_pipe[1][0], "monman") == 0){ /* si monman*/
143         monman(mots_pipe[1]);
144         exit(0); /* sinon on reste bloqué on attend */
145     }
146
147     if (strcmp(mots_pipe[1][0], "moncd") == 0){ /* si moncd*/
148         appell_mon_cd(dirs, mots_pipe[1], nb_mots_lastpipe);
149         exit(0); /* sinon on reste bloqué on attend */
150     }
151
152     if (strcmp(mots_pipe[1][0], "monexit") == 0){ /* si monexit*/
153         mon_exit_pipe(mots_pipe[1]);
154         exit(0); /* sinon on reste bloqué on attend */
155     }
156
157     else{
158         for(int i = 0; dirs[i] != 0; i++){
159             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[1][0]);
160             execv(pathname, mots_pipe[1]);
161         }
162
163         fprintf(stderr, "%s: notfound enfant 2\n", mots_pipe[1][0]);
164         exit(1) ;
165     }
166 }
167
/* PROCESSUS PARENT FIN */
168
169 close(mes_pipes[0][0]); /* fermeture pipe */
170 close(mes_pipes[0][1]);
171
172 if (attend_enfant == true){
173     /* on attend tous les enfants */
174     while (wait(NULL) > 0)
175         ;
176 }
177
178 *ilya_pipe = false;
179 return 0;
180 }
181
182
183
184
185
186 int deux_pipes(int mes_pipes[][2], char pathname[], int pathname_size, char* dirs[], char*
187 mots_pipe[][MaxMot], bool attend_enfant, bool* ilya_pipe, int* nb_mots_lastpipe){
188
189     int tmp, tmp2, tmp3;
190
191     /* vérification réussite pipe */
192     if (pipe(mes_pipes[0]) < 0 || pipe(mes_pipes[1]) < 0){
193         perror("pipe");
194         return -1; }
195
196     /* PROCESSUS PARENT > FORK 1 */
197     if ((tmp = fork()) <0){
198         perror("fork") ; /* vérification réussite fork1 */
199         return -1; }
200
201     /* PROCESSUS ENFANT 1 */
202     if (tmp ==0) {
203         dup2(mes_pipes[0][1], 1); /* redirection sortie */
204         close(mes_pipes[0][0]); /* fermeture de tous les pipes */
205         close(mes_pipes[0][1]);
206         close(mes_pipes[1][0]);
207         close(mes_pipes[1][1]);
208
209         int nb_elt_tab1 = nb_mots(mots_pipe, 0);
210
211         redirection_dans_pipe(mots_pipe, 0);
212
213         if (strcmp(mots_pipe[0][0], "monman") == 0){ /* si monman*/
214             monman(mots_pipe[0]);
215             exit(0); /* sinon on reste bloqué on attend */
216         }

```

```

216
217     if (strcmp(mots_pipe[0][0], "moncd") == 0){ /* si moncd*/
218         appel_mon_cd(dirs, mots_pipe[0], &nb_elt_tab1);
219         exit(0); /* sinon on reste bloqué on attend */
220     }
221
222     if (strcmp(mots_pipe[0][0], "monexit") == 0){ /* si monexit*/
223         mon_exit_pipe(mots_pipe[0]);
224         exit(0); /* sinon on reste bloqué on attend */
225     }
226
227
228     else{
229         for(int i = 0; dirs[i] != 0; i++){
230             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[0][0]);
231             execv(pathname, mots_pipe[0]);
232         }
233
234         fprintf(stderr, "%s: notfound enfant 1\n", mots_pipe[0][0]);
235         exit(1) ;
236     }
237 }
238
/* PROCESSUS PARENT > FORK 2 */
239 if ((tmp2 = fork()) <0){
240     perror("fork");
241     return -1;
242 }
243
/* PROCESSUS ENFANT 2 */
244 if (tmp2 ==0) {
245     dup2(mes_pipes[0][0], 0); /* redirection entrée depuis pipe1 */
246     dup2(mes_pipes[1][1], 1); /* redirection sortie vers pipe2 */
247     close(mes_pipes[0][0]);
248     close(mes_pipes[0][1]);
249     close(mes_pipes[1][0]);
250     close(mes_pipes[1][1]);
251
252     int nb_elt_tab2 = nb_mots(mots_pipe, 1); /* nb mots pipe 2 */
253     redirection_dans_pipe(mots_pipe, 1);
254
255     if (strcmp(mots_pipe[1][0], "monman") == 0){ /* si monman*/
256         monman(mots_pipe[1]);
257         exit(0); /* sinon on reste bloqué on attend */
258     }
259
260     if (strcmp(mots_pipe[1][0], "moncd") == 0){ /* si moncd*/
261         appel_mon_cd(dirs, mots_pipe[1], &nb_elt_tab2);
262         exit(0); /* sinon on reste bloqué on attend */
263     }
264
265     if (strcmp(mots_pipe[1][0], "monexit") == 0){ /* si monexit*/
266         mon_exit_pipe(mots_pipe[1]);
267         exit(0); /* sinon on reste bloqué on attend */
268     }
269
270     else{
271         for(int i = 0; dirs[i] != 0; i++){
272             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[1][0]);
273             execv(pathname, mots_pipe[1]);
274         }
275
276         fprintf(stderr, "%s: notfound enfant 2\n", mots_pipe[1][0]);
277         exit(1) ;
278     }
279 }
280
/* PROCESSUS PARENT > FORK3*/
281 close(mes_pipes[0][0]); /* fermeture complète pipe1 */
282 close(mes_pipes[0][1]);
283
284 if ((tmp3= fork()) <0){
285     perror("fork3");
286     return -1;
287 }
288
/* PROCESSUS ENFANT 3 */

```

```

292 if (tmp3 == 0) {
293     dup2(mes_pipes[1][0], 0); /* redirection entrée depuis pipe2 */
294     close(mes_pipes[1][0]); /* fermeture pipe2 */
295     close(mes_pipes[1][1]);
296
297     redirection_dans_pipe(mots_pipe, 2);
298
299     if (strcmp(mots_pipe[2][0], "monman") == 0){ /* si monman*/
300         monman(mots_pipe[2]);
301         exit(0); /* sinon on reste bloqué on attend */
302     }
303
304     if (strcmp(mots_pipe[2][0], "moncd") == 0){ /* si moncd */
305         appel_mon_cd(dirs, mots_pipe[2], nb_mots_lastpipe);
306         exit(0); /* sinon on reste bloqué on attend */
307     }
308
309     if (strcmp(mots_pipe[2][0], "monexit") == 0){ /* si monexit */
310         mon_exit_pipe(mots_pipe[2]);
311         exit(0); /* sinon on reste bloqué on attend */
312     }
313
314     else{
315         for(int i = 0; dirs[i] != 0; i++){
316             snprintf(pathname, pathname_size, "%s/%s", dirs[i], mots_pipe[2][0]);
317             execv(pathname, mots_pipe[2]);
318         }
319
320         fprintf(stderr, "%s: notfound enfant %d\n", mots_pipe[2][0]);
321         exit(1);
322     }
323 }
324
325 /* PROCESSUS PARENT FIN*/
326 close(mes_pipes[1][0]); /* fermeture pipe2 */
327 close(mes_pipes[1][1]);
328
329 if (attend_enfant == true){
330     while (wait(NULL) > 0) /* on attend tous les processus enfant */
331         ; /* se terminent */
332 }
333
334 *ilya_pipe = false; /* on marque bool "pipe" à faux */
335 return 0;
336
337 }
338
339
340
341 int nb_mots(char* mots_pipe[][MaxMot], int ligne){
342     int nb_elmt = 0;
343     for (int i = 0; i < MaxMot; i++){ /* parcourt la ligne */
344         if (mots_pipe[ligne][i] != NULL) /* si non nul */
345             nb_elmt++; /* incrémente nb_elemt */
346     }
347     return nb_elmt; /* renvoie nb élémt */
348 }
```

.7 Le fichier "redir_in_pipe.c" en intégralité

Voici l'intégralité du fichier `redir_in_pipe.c`.

```
1  /* ****
2 # Nom ..... : redir_in_pipe.c
3 # Rôle ..... : Fonction de gestion des redirections dans les pipes:
4 #           >, >>, 2>, 2>>, >&, <
5 # Auteur ..... : Avrile Floro
6 # Version ..... : V0.1 du 28/10/2023
7 # Licence ..... : réalisé dans le cadre du cours de SE
8 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
9 #*****
10
11 #include "sys.h"
12 #include "f_head.h"
13
14 int redirection_dans_pipe(char* mots_pipe[][MaxMot], int ligne){
15
16 /* GESTION DES REDIRECTIONS: CRÉATION */
17
18     mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR | /* tout le monde */
19                 S_IRGRP | S_IWGRP | S_IXGRP | /* peut lire, écrire */
20                 S_IROTH | S_IWOTH | S_IXOTH; /* et exécuter */
21
22     for (int i = 0; mots_pipe[ligne][i] != NULL ; i++) { /* on parcourt les mots */
23         /* redirection simple > */
24
25         if (simple_sortie_pipe (mots_pipe, ligne, mode, i) == 2)
26             continue;
27
28         /* redirection double >> */
29         if (double_sortie_pipe (mots_pipe, ligne, mode, i) == 2)
30             continue;
31
32         /* redirection stderr simple sortie 2> */
33         if (stderr_simple_sortie_pipe(mots_pipe, ligne, mode, i) == 2)
34             continue;
35
36         /* redirection stderr double sortie 2>> */
37         if (stderr_double_sortie_pipe(mots_pipe, ligne, mode, i) == 2)
38             continue;
39
40         /* redirection stderr et stdout en sortie simple >& */
41         if (stderr_stdout_sortie_pipe(mots_pipe, ligne, mode, i) == 2)
42             continue;
43
44         /* redirection simple < */
45         if (simple_entree_pipe (mots_pipe, ligne, mode, i) == 2)
46             continue;
47     }
48
49     return 0;
50 }
51
52 /* CRÉATION DES REDIRECTIONS */
53
54 int simple_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){
55
56     /* redirection simple en sortie > */
57     if (strcmp(mots_pipe[ligne][i], ">") == 0){
58         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le > */
59
60         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
61         /* write only, créé le fichier ou le remet à 0 si existe */
62         if (fd_redir_sortie < 0)
63             perror("fd_redir_sortie");
64
65         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */
66         if (t < 0)
67             perror("dup2 fd_redir_sortie");
68
69         mots_pipe[ligne][i] = 0; /* suppression du > */
70         mots_pipe[ligne][i + 1] = 0; /* suppression du nom de fichier */
```

```

71     close(fd_redir_sortie); /* fermeture */
72     return 2; /* signe continue dans gestion_redirection */
73 }
74
75 return 0 ;
76
77
78
79 int double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){
80
81     /* redirection simple en sortie > */
82     if (strcmp(mots_pipe[ligne][i], ">>") == 0){
83
84         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le > */
85
86         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode) ;
87             /* write only, créé le fichier ou ajoute à la suite si existe */
88         if (fd_redir_sortie < 0)
89             perror("fd_redir_sortie");
90
91         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */
92         if (t < 0)
93             perror("dup2 fd_redir_sortie");
94
95         mots_pipe[ligne][i] = 0; /* suppression du >> */
96         mots_pipe[ligne][i + 1] = 0; /* suppression du nom de fichier */
97
98         close(fd_redir_sortie); /* fermeture */
99         return 2; /* signe continue dans gestion_redirection */
100    }
101
102 return 0 ;
103
104
105 int stderr_simple_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){
106
107     /* redirection simple erreur en sortie 2> */
108     if (strcmp(mots_pipe[ligne][i], "2>") == 0){
109         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le 2> */
110
111         int fd_stderr_s_redir = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
112             /* write only, créé le fichier ou le remet à 0 si existe */
113         if (fd_stderr_s_redir < 0)
114             perror("fd_stderr_s_redir");
115
116         int t = dup2(fd_stderr_s_redir, 2); /* stderr redirigée vers l'arg du 2> */
117         if (t < 0)
118             perror("dup2 fd_stderr_sortie");
119
120         mots_pipe[ligne][i] = 0;
121         mots_pipe[ligne][i+1]=0;
122
123         close(fd_stderr_s_redir);
124         return 2; /* signe continue dans gestion_redirection */
125    }
126
127 return 0 ;
128}
129
130 int stderr_double_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){
131
132     /* redirection double erreur en sortie 2>> */
133     if (strcmp(mots_pipe[ligne][i], "2>>") == 0){
134         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le 2>> */
135
136         int fd_stderr_d_redir = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode) ;
137             /* write only, créé le fichier ou ajoute à la fin si existe */
138         if (fd_stderr_d_redir < 0)
139             perror("fd_stderr_d_redir");
140
141         int t = dup2(fd_stderr_d_redir, 2); /* stderr redirigée vers l'arg du 2>> */
142         if (t < 0)
143             perror("dup2 fd_stderr_sortie");
144
145         mots_pipe[ligne][i] = 0;
146         mots_pipe[ligne][i+1]=0;

```

```

147     close(fd_stderr_d_redir);
148     return 2; /* signe continue dans gestion_redirection */
149 }
150 return 0;
151 }

154 int stderr_stdout_sortie_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){
155
156     if (strcmp(mots_pipe[ligne][i], ">&") == 0){
157         char* sortie = mots_pipe[ligne][i+1]; /* le fichier de sortie après le > */
158
159         int fd_s_sortie_and = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
160         /* write only, crée le fichier ou le remet à 0 si existe */
161         if (fd_s_sortie_and < 0)
162             perror("fd_s_sortie_and");
163
164         int t = dup2(fd_s_sortie_and, 1); /* redir stout vers le fichier */
165         if (t < 0)
166             perror("dup2 stout fd_s_sortie_and");
167
168         int tt = dup2(fd_s_sortie_and, 2); /* redir stderr vers le fichier */
169         if (tt < 0)
170             perror("dup2 stderr fd_s_sortie_and");
171
172         mots_pipe[ligne][i] = 0; /* suppression du > */
173         mots_pipe[ligne][i+1] = 0; /* suppression du nom de fichier */
174
175         close(fd_s_sortie_and); /* fermeture */
176         return 2; /* signe continue dans gestion_redirection */
177     }
178
179     return 0;
180 }

182 int simple_entree_pipe (char* mots_pipe[][MaxMot], int ligne, mode_t mode, int i){
183
184     /* redirection simple en entrée < */
185     if (strcmp(mots_pipe[ligne][i], "<") == 0){
186         char* entree = mots_pipe[ligne][i+1]; /* le fichier entrée après le < */
187
188         int fd_redir_s_entree = open(entree, O_RDONLY, mode) ;
189         /* read only */
190         if (fd_redir_s_entree < 0)
191             perror("fd_redir_s_entree");
192
193         int t = dup2(fd_redir_s_entree, 0); /* entrée redirigée vers l'arg du < */
194         if (t < 0)
195             perror("dup2 fd_redir_s_entree");
196
197         mots_pipe[ligne][i] = 0; /* suppression du < */
198         mots_pipe[ligne][i+1] = 0; /* suppression du nom de fichier */
199
200         close(fd_redir_s_entree);
201         return 2; /* signe continue dans gestion_redirection */
202     }
203
204     return 0;
205 }

```

.8 Le fichier "moncd.c" en intégralité

Voici l'intégralité du fichier moncd.c.

```
1  /* ****
2 # Nom ..... : moncd.c
3 # Rôle ..... : la fonction mon_cd permet de changer de répertoire
4 # Auteur ..... : Avrile Floro
5 # Version ..... : V0.1 du 01/09/2023
6 # Licence ..... : réalisé dans le cadre du cours de SE
7 # Compilation : compiler avec les autres programmes (selon l'appelant)
8 ****
9
10 #include "sys.h"
11 #include "f_head.h"
12
13 int appelle_mon_cd(char *dir[], char *mot[], int* nb_total_mots){
14     if (strcmp(mot[0], "moncd") == 0){ // si le mot[0] est moncd
15
16         if ((*nb_total_mots > 2)&&(mot[2]!=NULL)){ // s'il y a plus de 2 mots
17             fprintf(stderr, "Usage: moncd [dir]\n");
18             return 1;
19         }
20         if (*nb_total_mots == 2){
21             mon_cd(&mot[1]); // on lance la fonction mon_cd
22             return 0;
23         }
24
25         else { // si mot[1] n'existe pas
26             mon_cd(NULL); // on lance la fonction mon_cd
27             return 0;
28         }
29     }
30     return 2; /* pour ne pas trigger le "continue" dans main */
31 }
32
33
34 int mon_cd(char *dir[]){
35     int t;
36
37     if (dir != NULL && *dir != NULL){ // s'il y a un path en argument
38         t = (chdir(*dir)); // on change de répertoire
39         if (t < 0){ // si erreur
40             perror("chdir"); // affiche msg erreur
41             return 1;
42         }
43     }
44
45     else { // s'il n'y a pas de path en argument
46         char * h = getenv("HOME"); // on récupère le path du répertoire HOME
47
48         if (h == NULL){ // si erreur
49             perror("Pas de répertoire HOME"); // affiche msg erreur
50             return 1;
51         }
52
53         else{
54             t = chdir(h) ; // on change de répertoire vers HOME
55
56             if (t < 0){ // si erreur
57                 perror("Pas de répertoire HOME"); // affiche msg erreur
58                 return 1;
59             }
60         }
61     }
62     return 0;
63 }
```

.9 Le fichier "and_monexit.c" en intégralité

Voici l'intégralité du fichier `and_monexit.c`.

```
1  /* ****
2 # Nom ..... : and_monexit.c
3 # Rôle ..... : Gestion des processus en arrière-plan
4 #           Gestion de mon_exit
5 # Auteur ..... : Avrile Floro
6 # Version ..... : V0.1 du 26/10/2023
7 # Licence ..... : réalisé dans le cadre du cours de SE
8 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
9 # ****
10
11
12 #include "sys.h"
13 #include "f_head.h"
14
15
16 int arriere_plan(int* nb_total_mots, char* mot[], bool* attend_enfant){
17
18     if ((*nb_total_mots > 1) && mot[*nb_total_mots-1] && (strcmp (mot[*nb_total_mots-1], "&") == 0)){
19         // s'il y a plus d'un mot et que le dernier mot est "&"
20         *attend_enfant = false;
21         /* pas besoin d'attendre l'enfant, passage à faux */
22
23         mot[*nb_total_mots-1] = 0; /* remplacement & par 0 */
24         /* pour ne pas être pris en compte par execv */
25
26         (*nb_total_mots)--; /* mise à jour du nb de mots */
27         return 0;
28     }
29
30     return 0;
31 }
32
33
34 int mon_exit(char *mot[]){
35     if (strcmp(mot[0], "monexit") == 0){ // si le mot[0] est monexit
36         printf("Bye\n");
37         exit(0); // on quitte le shell sans erreur
38     }
39     return 0;
40 }
41
42
43 int mon_exit_pipe(char *mot[]){
44     if (strcmp(mot[0], "monexit") == 0){ // si le mot[0] est monexit
45         printf("Bye\n");
46     }
47     return 0;
48 }
```

.10 Le fichier "redirection.c" en intégralité

Voici l'intégralité du fichier redirection.c.

```
1  /* ****
2 # Nom ..... : redirection.c
3 # Rôle ..... : Fonction de gestion des indirections:
4 #           >, >>, 2>, 2>>, >&, <
5 # Auteur ..... : Avrile Floro
6 # Version ..... : V0.1 du 18/10/2023
7 # Licence ..... : réalisé dans le cadre du cours de SE
8 # Compilation : Compilé grâce au Makefile (make) avec les autres fichiers
9 # ****
10
11 #include "sys.h"
12 #include "f_head.h"
13
14 int redirection(char** mot, int* nb_total_mots, bool* redir_sortie, bool* stderr_sortie, bool*
15   redir_s_entree, int* old_entree, int* old_sortie, int* old_erreur, bool* redir, bool*
16   reset){
17
18   /* CRÉATION DES REDIRECTIONS */
19
20   if (*redir == true){
21
22     mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR | /* tout le monde */
23                   S_IRGRP | S_IWGRP | S_IXGRP | /* peut lire, écrire */
24                   S_IROTH | S_IWOTH | S_IXOTH ; /* et exécuter */
25
26     for (int i = 0; i < *nb_total_mots - 1 ; i++) { /* on parcourt les mots */
27       /* redirection simple > */
28       simple_sortie (mot, nb_total_mots, mode, redir_sortie, i);
29       if (*redir_sortie)
30         continue;
31
32       /* redirection double >> */
33       double_sortie (mot, nb_total_mots, redir_sortie, mode, i);
34       if (*redir_sortie)
35         continue;
36
37       /* redirection stderr simple sortie 2> */
38       stderr_simple_sortie(mot, nb_total_mots, stderr_sortie, mode, i);
39       if (*stderr_sortie)
40         continue;
41
42       /* redirection stderr double sortie 2>> */
43       stderr_double_sortie(mot, nb_total_mots, stderr_sortie, mode, i);
44       if (*stderr_sortie)
45         continue;
46
47       /* redirection stderr et stdout en sortie simple >& */
48       stderr_stdout_sortie(mot, nb_total_mots, stderr_sortie, redir_sortie, mode, i);
49       if ((*stderr_sortie) && (*redir_sortie))
50         continue;
51
52       /* redirection simple < */
53       simple_entree (mot, nb_total_mots, redir_s_entree, mode, i);
54       if (*redir_s_entree)
55         continue;
56     }
57     *redir = false;
58   }
59
60   /* QUAND ON VEUT RÉINITIALISER LES ENTRÉES ET SORTIES */
61   else if (*reset == true) {
62
63     /* reset > et >> et >& stdout */
64     reset_sortie(redir_sortie, old_sortie);
65
66     /* reset 2> et 2>> et >&, stderr */
67     reset_stderr(stderr_sortie, old_erreur);
68
69     /* reset < , stdin */
70     reset_stdin(redir_s_entree, old_entree);
71
72 }
```

```

69     *reset = false;}
70
71     return 0;
72 }
73
74
75
76
77 /* CRÉATION DES REDIRECTIONS */
78
79 int simple_sortie (char** mot, int* nb_total_mots, mode_t mode, bool* redir_sortie, int i){
80
81     /* redirection simple en sortie > */
82     if (strcmp(mot[i], ">") == 0){
83         char* sortie = mot[i+1]; /* le fichier de sortie après le > */
84
85         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
86             /* write only, crée le fichier ou le remet à 0 si existe */
87         if (fd_redir_sortie < 0)
88             perror("fd_redir_sortie");
89
90         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */
91         if (t < 0)
92             perror("dup2 fd_redir_sortie");
93
94         mot[i] = 0; /* suppression du > */
95         (*nb_total_mots)-- ; /* mise à jour du nb de mots */
96         mot[i + 1] = 0; /* suppression du nom de fichier */
97         (*nb_total_mots)-- ; /* mise à jour du nb de mots */
98
99
100        close(fd_redir_sortie); /* fermeture */
101        *redir_sortie = true ;
102        /* passage à vrai du bool */
103    }
104
105    return 0 ;
106}
107
108 int double_sortie (char** mot, int* nb_total_mots, bool* redir_sortie, mode_t mode, int i){
109
110     /* redirection simple en sortie >> */
111     if (strcmp(mot[i], ">>") == 0){
112
113         char* sortie = mot[i+1]; /* le fichier de sortie après le > */
114
115         int fd_redir_sortie = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode) ;
116             /* write only, crée le fichier ou ajoute à la suite si existe */
117         if (fd_redir_sortie < 0)
118             perror("fd_redir_sortie");
119
120         int t = dup2(fd_redir_sortie, 1); /* sortie redirigée vers l'arg du > */
121         if (t < 0)
122             perror("dup2 fd_redir_sortie");
123
124         mot[i] = 0; /* suppression du >> */
125         (*nb_total_mots)-- ; /* mise à jour du nb de mots */
126         mot[i + 1] = 0; /* suppression du nom de fichier */
127         (*nb_total_mots)-- ; /* mise à jour du nb de mots */
128
129         close(fd_redir_sortie); /* fermeture */
130         *redir_sortie = true ; /* passage à vrai du bool */
131    }
132
133    return 0 ;
134}
135
136 int stderr_simple_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
137     int i){
138
139     /* redirection simple erreur en sortie 2> */
140     if (strcmp(mot[i], "2>") == 0){
141         char* sortie = mot[i+1]; /* le fichier de sortie après le 2> */
142
143         int fd_stderr_s_redir = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
144             /* write only, crée le fichier ou le remet à 0 si existe */

```

```

144     if (fd_stderr_s_redir < 0)
145         perror("fd_stderr_s_redir");
146
147     int t = dup2(fd_stderr_s_redir, 2); /* stderr redirigée vers l'arg du 2> */
148     if (t < 0)
149         perror("dup2 fd_stderr_sortie");
150
151     mot[i] = 0;
152     (*nb_total_mots)--;
153     mot[i+1]=0;
154     (*nb_total_mots)--;
155
156     close(fd_stderr_s_redir);
157     *stderr_sortie = true ;
158
159 }
160
161 return 0;
}
162
163 int stderr_double_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, mode_t mode,
164 int i){
165
166     /* redirection double erreur en sortie 2>> */
167     if (strcmp(mot[i], "2>>") == 0){
168         char* sortie = mot[i+1]; /* le fichier de sortie après le 2>> */
169
170         int fd_stderr_d_redir = open(sortie, O_WRONLY | O_CREAT | O_APPEND, mode) ;
171             /* write only, créé le fichier ou ajoute à la fin si existe */
172         if (fd_stderr_d_redir < 0)
173             perror("fd_stderr_d_redir");
174
175         int t = dup2(fd_stderr_d_redir, 2); /* stderr redirigée vers l'arg du 2>> */
176         if (t < 0)
177             perror("dup2 fd_stderr_sortie");
178
179         mot[i] = 0;
180         (*nb_total_mots)--;
181         mot[i+1]=0;
182         (*nb_total_mots)--;
183
184         close(fd_stderr_d_redir);
185         *stderr_sortie = true ;
186     }
187
188 return 0;
}
189
190 int stderr_stdout_sortie (char** mot, int* nb_total_mots, bool* stderr_sortie, bool*
191 redirect_sortie, mode_t mode, int i){
192
193     if (strcmp(mot[i], ">&") == 0){
194         char* sortie = mot[i+1]; /* le fichier de sortie après le > */
195
196         int fd_s_sortie_and = open(sortie, O_WRONLY | O_CREAT | O_TRUNC, mode) ;
197             /* write only, créé le fichier ou le remet à 0 si existe */
198         if (fd_s_sortie_and < 0)
199             perror("fd_s_sortie_and");
200
201         int t = dup2(fd_s_sortie_and, 1); /* redir stout vers le fichier */
202         if (t < 0)
203             perror("dup2 stout fd_s_sortie_and");
204
205         int tt = dup2(fd_s_sortie_and, 2); /* redir stderr vers le fichier */
206         if (tt < 0)
207             perror("dup2 stderr fd_s_sortie_and");
208
209         mot[i] = 0; /* suppression du > */
210         (*nb_total_mots)--; /* mise à jour du nb de mots */
211         mot[i + 1] = 0; /* suppression du nom de fichier */
212         (*nb_total_mots)--; /* mise à jour du nb de mots */
213
214         close(fd_s_sortie_and); /* fermeture */
215         *stderr_sortie = true ; /* passage à vrai du bool */
216         *redirect_sortie = true ; /* passage à vrai du bool */
217     }
218
219 return 0;
}

```

```

218 }
219
220 int simple_entree (char** mot, int* nb_total_mots, bool* redir_s_entree, mode_t mode, int i){
221
222     /* redirection simple en entrée < */
223     if (strcmp(mot[i], "<") == 0){
224         char* entree = mot[i+1]; /* le fichier entrée après le < */
225
226         int fd_redir_s_entree = open(entree, O_RDONLY, mode) ;
227             /* read only */
228         if (fd_redir_s_entree < 0)
229             perror("fd_redir_s_entree");
230
231         int t = dup2(fd_redir_s_entree, 0); /* entrée redirigée vers l'arg du < */
232         if (t < 0)
233             perror("dup2 fd_redir_s_entree");
234         else {
235             mot[i] = 0; /* suppression du < */
236             (*nb_total_mots]--;
237             mot[i + 1] = 0; /* suppression du nom de fichier */
238             (*nb_total_mots]--;
239             close(fd_redir_s_entree);
240             *redir_s_entree = true ; /* passage à vrai du bool */
241         }
242     }
243     return 0;
244 }
245
246 /* RÉINITIALISATION DES VALEURS INITIALES */
247
248 int reset_sortie(bool* redir_sortie, int* old_sortie){
249     /* si > ou >> ou >& (la sortie)*/
250     if (*redir_sortie == true) {
251         int t = dup2(*old_sortie, 1); /* réinitialisation de stdout */
252         if (t < 0)
253             perror("dup old_sortie");
254         else if (*redir_sortie)
255             *redir_sortie = false ;
256     }
257     return 0;
258 }
259
260 int reset_stderr(bool* stderr_sortie, int* old_erreur){
261     /* si 2> ou 2>> ou >& (stderr)*/
262     if (*stderr_sortie == true) {
263         int t = dup2(*old_erreur, 2); /* réinitialisation de stderr */
264         if (t < 0)
265             perror("dup old_sortie");
266         else if (*stderr_sortie)
267             *stderr_sortie = false ;
268     }
269     return 0;
270 }
271
272
273 int reset_stdin(bool* redir_s_entree, int* old_entree){
274     /* si < */
275     if (*redir_s_entree == true){
276         int t = dup2(*old_entree,0); /* réinit de strdin */
277         if (t < 0)
278             perror("dup old_entree");
279         *redir_s_entree = false ;
280     }
281     return 0;
282 }
```

.11 Le fichier "monman.c" en intégralité

Voici l'intégralité du fichier monman.c.

```
1 /* # Nom ..... : man.c
2 # Rôle ..... : Manuel du shell
3 # Auteur ..... : Avrile Floro
4 # Version ..... : V0.1 du 30/01/2024
5 # Licence ..... : réalisé dans le cadre du cours de SE
6 # Usage : Compilé via make */
7
8
9 #include "sys.h"
10 #include "f_head.h"
11
12 int monman(char* mot[]){
13     if (strcmp(mot[0], "monman") == 0){ /* si commence par mon man */
14         if (mot[1] == 0){ /* si pas d'argument */
15             fprintf(stderr, "Quelle page du manuel voulez-vous?\nPar exemple, essayer 'monman
16 monman'.\n");
17         }
18
19         else if (mot[2] != NULL){ /* si deux arguments */
20             fprintf(stderr, "Usage: monman %s\n", mot[1]);
21         }
22
23         else if (strcmp(mot[1], "monman") == 0){ // man man
24             printf("\nNOM\nmonman - une interface pour le manuel de référence du système\n
25 DESCRIPTION\nmonman est le pagineur de manuel du système. Chaque argument donné à monman
26 est le nom d'une commande créée pour ce shell élémentaire. La page de manuel associée à
27 son argument est alors trouvée et affichée.\n\nEXEMPLES D'UTILISATION\nmonman monman\
28 moncd\nmonman monexit\n\n");
29         }
30
31         else if (strcmp(mot[1], "monexit") == 0){
32             printf("\nNOM\nmonexit - provoque la sortie du shell\n\nDESCRIPTION\nLa commande
33 monexit permet au shell de sortir de son environnement d'exécution actuel. La commande
34 monexit ne prend pas d'option.\n\nEXEMPLE D'UTILISATION\nmonexit\n\n");
35         }
36
37         else if (strcmp(mot[1], "moncd") == 0){
38             printf("\nNOM\nmoncd - changer le répertoire de travail\n\nDESCRIPTION\nL'utilitaire
39 moncd change le répertoire de travail de l'environnement d'exécution du shell actuel en
40 exécutant les étapes suivantes dans l'ordre.\n    1. Si un répertoire est donné en
argument à la commande 'moncd', alors le répertoire de travail du shell sera modifié pour
correspondre au chemin spécifié en argument.\n    2. Si aucun répertoire n'est spécifié
en argument, 'moncd' se comportera comme si le chemin spécifié dans la variable d'
environnement HOME était l'argument. Cela signifie que le répertoire de travail sera changé
pour celui défini dans HOME, à condition que cette variable ne soit ni vide ni indéfinie
.\n\nEXEMPLES D'UTILISATION\nmoncd /home/avrile/Desktop\n\n");
41         }
42
43         else{ /* si l'argument passé n'existe pas */
44             fprintf(stderr, "Cette commande n'existe pas.\n");
45         }
46     }
47
48     return 0;
49 }
```

.12 L'évaluation du cours

10.2 Évaluation du cours

Un grand merci à vous de prendre le temps de répondre aux questions suivantes pour m'aider à évaluer la façon dont le cours s'est passé de votre point de vue.

– **Le contenu du cours a-t-il correspondu à ce que vous attendiez ? (sinon, de quelle manière)**
Oui, je n'avais aucune connaissance dans le domaine avant le début du cours. J'ai découvert beaucoup de nouveaux concepts.

– **Qu'est-ce qui vous a le plus surpris (en bien) ?**
L'organisation très progressive du cours qui nous permet de réaliser un projet ambitieux par étapes.

– **Qu'est-ce qui vous a le plus déçu ?**
Rien.

– **Quels étaient les chapitres les plus difficiles ?**
Les chapitres 2, 5 et 6 étaient les plus difficiles. Ils correspondent aux développements de notre shell.

– **Quels étaient les chapitres les plus faciles ?**
Selon moi, les chapitres 1 et 9 étaient les plus faciles.

– **Quels étaient les chapitres les plus intéressants ?**
J'ai bien aimé le chapitre 3, dans lequel on craque notre propre mot de passe. Je l'avais trouvé très instructif et ludique.

– **Quels étaient les chapitres les moins intéressants ?**
NA

– **Que me suggérez-vous de modifier dans l'ordre des chapitres ?**
NA

– **Qu'est-ce qui est en trop dans le cours ?**
NA

– **Qu'est-ce qui manque dans le cours ?**
Peut-être aborder le multi-threading. C'est un sujet que j'ai vu aborder à plusieurs reprises durant mes recherches.

– **Comment étaient les exercices du point de vue quantité (pas assez, trop).**
Les exercices et les chapitres étaient longs à faire/finir. Cependant, c'est compensé par le fait que le projet est presque totalement traité grâce aux exercices, ce qui réduit le temps passé à travailler sur le projet à la fin du cours.

– **Comment étaient les exercices du point de vue difficulté (trop durs, trop faciles) ?**
Ok.
Les bonus étaient plus compliqués et parfois il était difficile de trouver des informations car les techniques utilisées peuvent l'être à des fins malveillantes.

– **Que donneriez-vous comme conseil à un étudiant qui va suivre le cours ?**
À titre personnel, j'aime travailler sur le cours et les chapitres d'un bloc. J'ai passé environ 6/7 semaines à temps plein sur les chapitres et 2 semaines supplémentaires afin de reprendre et de finaliser le projet, sans compter les relectures avant l'envoi des chapitres durant l'année.

– **Que me donneriez-vous comme conseil en ce qui concerne le cours ?**
NA

Merci beaucoup de votre investissement. On souhaiterait avoir plus de professeurs investis comme vous l'êtes !

– Si vous deviez mettre une note globale au cours, entre 0 et 20, laquelle mettriez-vous ?
20

– Quelles questions manque-t-il pour évaluer correctement le cours (et bien évidemment, quelle réponse vous y apporteriez) ?

NA