Pierre DAHERON - 20013464

Systèmes d'exploitation - Chapitre 10- Projet Final

June 24, 2022

Contents

Introduction

Ce chapitre constitue le projet final du cours de système d'exploitation de L2 de l'IED Paris 8. Le projet reprend les éléments vus dans les chapitres précédents pour recréeer un shell avec quelques fonctionnalités. Le présent rapport a pour but d'expliciter différents points, en commençant par l'architecture du programme, où je détaillerai la structure de fichiers ainsi que la procédure pour la compilation. Ensuite, vient la liste des fonctionnalités du programme avec les explications et les justifications des choix de ces fonctionnalités, ainsi que leurs spécifications dans ce shell. Ces spécifications seront suivies par des éléments plus techniques (du code commenté et explicité), et enfin les limites du programme.

Le programme en lui-même ainsi que le code dans son intégralité sera envoyé avec ce pdf explicatif.

10-1

Architecture du programme

Le programme complet est assez conséquent, tant au niveau du code du shell lui-même que des fichiers annexes (pages du manuel, commandes externes, etc). J'ai donc regroupé certains fichiers dans des dossiers, et séparé le code dans plusieurs fichiers distincts. Pour être sûr que tous les codes soient bien compilés, j'ai utilisé un Makefile qui reprend tous les fichiers nécessaires.

La compilation du programme se fait donc à l'aide de la commande make sous linux dans un terminal ouvert dans le dossier contenant l'intégralité du programme.

Le dossier principal comprend donc :

- un fichier exécutable : monShell, le fichier permettant de démarrer le programme.
- deux fichiers .c sources : shell-final.c et util.c. Le premier comprend le code principal du shell et dans le second se trouvent toutes les fonctions annexes.
- un fichier d'en-têtes sys.h, qui réunit les librairies à inclure ainsi que les prototypes des fonctions des fichiers .c décrits précédemment.
- un Makefile, pour la compulation
- Deux sous-dossiers : external et man_pages, qui contiennent respectivement les commandes externes et les pages du manuel. Le contenu de ces deux sous-dossiers sera explicité en détail dans le paragraphe suivant (sur les commandes).

Le code du makefile est le suivant :

```
1  all : monShell external
2
3  monShell : shell-final.c util.c
4          gcc -g -Wall shell-final.c -lreadline util.c -o monShell
5
6
7  external : monkill monsleep monMan monCp
8
9  monMan : external/monMan.c
10         gcc -g -Wall external/monMan.c -o external/monMan
11
12 monCp : external/monCp.c
13         gcc -g -Wall external/monCp.c -o external/monCp
14
15 monkill : external/monkill.c
16         gcc -g -Wall external/monkill.c -o external/monkill
17
18 monsleep : external/monsleep.c
19         gcc -g -Wall external/monsleep.c external/f-monsleep.c -o external/monsleep
```

```
20
21 cleanAll :
22         rm -r *.o
```

```
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_10$ make
gcc -g -Wall shell-final.c -lreadline util.c -o monShell
gcc -g -Wall external/monkill.c -o external/monkill
gcc -g -Wall external/monsleep.c external/f-monsleep.c -o external/monsleep
gcc -g -Wall external/monMan.c -o external/monMan
gcc -g -Wall external/monCp.c -o external/monCp
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_10$
```

On peut constater que le makefile compile les deux fichiers .c du dossier principal, ainsi que les fichiers .c des commandes externes. Les options du compilateur sont -g pour le débogage, -Wall pour afficher tous les warnings à la compilation, -o pour spécifier le nom du fichier de sortie (source : manuel linux gcc).

10-2

Fonctionnalités implémentées (choix opérés)

2.1 commandes

On distingue dans ce shell deux types de commandes :

- Les commandes internes

  Ces commandes sont intégrées au code du shell car elles agissent directement sur lui. Elles sont au nombre de 2 : moncd et monexit. La commande moncd permet de changer le répertoire courant traité par le shell. Elle agit donc sur le répertoire courant qui est l'une des variables du shell. C'est l'une des fonctionnalités implémentées lors des précédents chapitres, elle est donc incluse dans ce shell. De la même manière, monexit permet l'arrêt du shell, et en est donc une commande interne. C'est également l'une des fonctionnalités implémentées dans les chapitres précédents et a donc sa place dans le projet final.

```
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_10$ ./monShell
?moncd ..
?pwd
/home/pierred/Documents/ied/L2/SE
?moncd
?pwd
/home/pierred
?moncd Documents/ied/L2/SE
?pwd
/home/pierred/Documents/ied/L2/SE
?monexit
Ciao
```

- Les commandes externes

  Les commandes externes sont au nombre de 4 : monsleep, monkill, monCp et monMan. La commande monsleep a fait l'objet d'un exercice obligatoire au chapitre précédent et les spécifications de cette commande restent inchangées par rapport à cet exercice. La commande monkill, également implémentée dans le chapitre précédent (bonus), permet d'envoyer un signal à un processus donné (ou d'afficher la liste des signaux traités avec l'option -l). La commande monMan est la commande demandée dans ce chapitre, et permet d'afficher la page du manuel créée pour expliciter chacune des commande. La dernière commande, monCp est un ajout personnel pour étoffer un peu la liste des commandes disponibles. Elle permet de copier un fichier ou un dossier.

```
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_10$ ./monShell
?monsleep 2
c'est mon sleep qui a fait ça.
?monsleep 1m &
?monCp c'est mon sleep qui a fait ça.

?pwd
/home/pierred/Documents/ied/L2/SE/chapitre_10
?monCp sys.h ../
erreur en écriture: Is a directory
?monCp sys.h ../sys2.h
?ls ../
'chapitre 1'  'chapitre 2'  'chapitre 4'   chapitre_6   chapitre_8  'Cours_SE V2.pdf'   shell    tata   toto55
 chapitre_10  'chapitre 3'  'chapitre 5'   chapitre_7   chapitre_9   monshell           sys2.h   toto
?monsleep 2
c'est mon sleep qui a fait ça.
?monsleep 1m
c'est mon sleep qui a fait ça.
?monMan monCp
monCp(1)

Nom :
        monCp - copie un fichier

Synopsis :
        monCp [repertoire_fichier/nom_fichier] [nouveau_repertoire/nouveau_nom]

Description :
        Copie un fichier existant dans un nouveau répertoire. Il faut impérativement les deux arguments décrits. Si le
        disponible en lecture, la commande renvoie une erreur. Si le fichier de destination existe, il sera écrasé. S'
        fichier n'est pas disponible en écriture, la commande renvoie une erreur.


        Pas d'options



Auteur :
        Pierre Daheron

Bugs connus :
?
```

2.2 arrière-plan

Le shell offre la possibilité de lancer des processus en arrière-plan. Il suffit pour cela que le symbole & soit présent (seul dans son mot), à la fin d'une commande.

NB : l'arrière plan est incompatible (dans ce shell) avec les pipes, mais pas avec les redirections.

2.3 redirections

Les redirections correspondent à un exercice d'un chapitre précédent. Les redirections implémentées sont les suivantes :

- < :

  permet de modifier le flux d'entrée d'une commande en prenant comme nouvelle entrée un fichier. Si le fichier n'est pas existant, ou est protégé en lecture, une erreur est renvoyée.

- > :

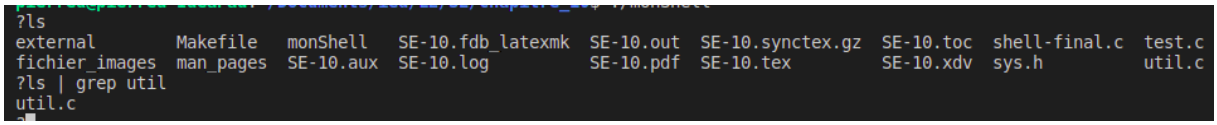
  permet de modifier le flux de sortie d'une commande et d'écrire dans un fichier. Le fichier est créé s'il n'existe pas. Dans le cas contraire, les données du fichier sont écrasées.

- >> :

  redirection similaire à la précédente, mais les données sont ajoutées au fichier.

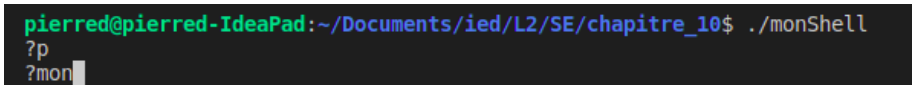
2.4 pipes

Les pipes sont des commandes enchaînées. Le flux de sortie d'une commande correspond au flux d'entrée de la commande suivante. Le présent shell est prévu pour autoriser au maximum trois commandes successives (nombre choisi arbitrairement, qui peut être augmenté assez simplement à l'aide du paramètre global MaxCommandes, cf partie code). Cette partie a été implémentée dans un chapitre précédent et fait donc partie du projet final.



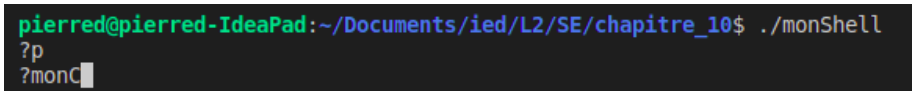
2.5 autocomplétion

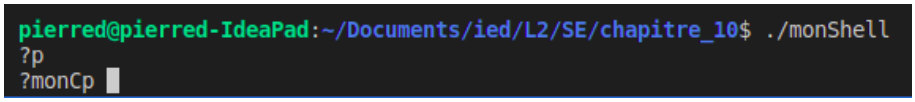
L'auto-complétion correspond au bonus de ce chapitre. L'auto-complétion implémentée permet de compléter automatiquement les chemins vers les adresses des différents fichiers, mais également les commandes spécifiques au shell (voir paragraphe précédent commandes). Il suffit de commencer à taper la commande ou l'adresse dans le shell, puis d'appuyer sur TAB. S'il n'y a qu'une seule possibilité, le nom est automatiquement complété.



Dans l'exemple ci-dessus, si on tape p, puis TAB, rien ne se produit car aucun des fichiers ou dossiers ne commence par p, et les commandes non plus. De la même manière pour mon, car plusieurs commandes commencent par mon.

Voici maintenant un exemple d'autocomplétion : lorsqu'on tape monC + TAB, on obtient :





10-3

Code

3.1 Shell

Le code du shell est situé dans deux fichiers différents : shell-final.c et util.c. Le premier fichier est le fichier principal, qui contient tout ce qui concerne le lancement et l'organisation générale. Le deuxième fichier contient toutes les fonctions annexes, qui permettent de repérer les symboles spéciaux dans une ligne, d'exécuter une commande, ou encore les commandes internes (entre autres).

3.1.1 initialisation du shell

La première partie du main (fichier shell-final.c) contient l'initialisation de toutes les variables utiles au shell.

```
1  int main(int ac, char* ldc[]){
2      //définitions
3      char * ligne;
4      char * mot [MaxMot];
5      char * dirs [MaxDirs];
6      int i, j, nb_args, nb_pipe, arriere_plan, exit,m_if, nb_commandes;
7      int pipe_list [MaxCommandes-1][2];
8
9
10     // Création d'un tableau de commandes et initialisation du tableau.
11     ptr_com commandes[MaxCommandes];
12     for(i =0; i< MaxCommandes; i++){
13         commandes[i] = init_commande();
14     }
15     // initialisation des pipes
16     nb_pipe = 0;
17
18     // Découper une copie de PATH en répertoires
19     int next_dir= decouper(getenv("PATH"),":",dirs,MaxDirs)-1;
20     // ajout du répertoire des commandes externes
21     dirs[next_dir]="external";
22
23     // auto-completion : ajout des noms de fonctions
24     rl_attempted_completion_function = function_name_completion;
```

La variable ligne contiendra la chaîne de caractères lues lors de l'appui sur "entrée". La variable mot contiendra cette même chaîne de caractères séparée en mots. La variable dirs contient les répertoires du PATH. La variable pipe_list contient quant à elle des paires de descripteurs de fichiers qui seront utilisés pour les pipes. Le nombre maximal de pipes correspond au nombre maximal de commandes moins un. La variable rl_attempted_completion_function est propre à la librairie readline. Il s'agit de la fonction permettant l'autocomplétion des différentes noms de commandes.

3.1.2 Boucle de lecture du shell

```
1  //Boucle de lecture et d'interprétation des commandes
2
3      while(1){
4          ligne = readline(PROMPT);
5          //************************** RAZ des variables : ****************************
6          // RAZ des redirections (file descriptors, fd):
7          for(i=0; i< MaxCommandes; i++){
8              commandes[i] -> fd_in = STDIN_FILENO;
9              commandes[i] -> fd_out = STDOUT_FILENO;
10             j=0;
11             while(commandes[i]->ligne[j]){
12                 commandes[i]-> ligne[j] = NULL;
13                 j++;
14             }
15         }
16         // RAZ des pipes
17         for(i=0; i < nb_pipe;i++){
18             pipe_list[i][1]= STDOUT_FILENO;
19         }
20         nb_args = decouper(ligne," \t\n",mot,MaxMot);
21         //gestion de l'arrière-plan
22         arriere_plan = 0;
23         // pipes
24         nb_pipe = 0;
25         // monIf
26         m_if=0;
27
28
29         // ************************ décodage de la ligne : ***************************
30
31         nb_commandes = decode_ligne(mot, commandes,&nb_pipe,pipe_list, &arriere_plan, nb_args, &m_if);
32
33         // ***************************** Exécution de la commande ************************
34
35         if(nb_pipe > 0){
36             execute_commande(*commandes[0], dirs,0);
37             for(i=1;i<nb_commandes;i++){
38                 // en cas de pipe, on ferme l'entrée du pipe précédent
39                 close(pipe_list[i-1][1]);
40                 execute_commande(*commandes[i], dirs,0);
41                 close(pipe_list[i-1][0]);
42             }
43
44         }
45
46         else{
47             // Pour la première commande, on considère l'arrière plan
48             exit = execute_commande(*commandes[0], dirs, arriere_plan);
```

```
49              if(exit == 1) break;
50              // Pour les autres commandes, pas d'arrière plan.
51              for(i=1;i<nb_commandes;i++){
52
53                  execute_commande(*commandes[i], dirs,0);
54
55              }
56          }
57      }
58
59      printf("Ciao\n");
60
61      // libération de l'espace alloué :
62      for(i=0; i< MaxCommandes;i++){
63          free(commandes[i]);
64      }
65      free(ligne);
66      return 0;
```

La boucle de lecture peut être séparée à son tour en plusieurs parties :

- lecture de la ligne

  Contrairement à ce qui est proposé en cours, j'utilise readline, qui possède une autocomplétion déjà implémentée pour les noms de fichiers et qui permet pour la suite de compléter avec d'autres termes en autocomplétion. La commande readline(PROMPT) permet d'afficher le symbole PROMPT (ici : "?"), puis d'attendre l'appui sur la touche de retour chariot avant d'affecter la chaîne de caractères tapée à la variable ligne.

- RAZ :

  Les lignes 7 à 26 correspondent à la remises à 0 des variables du shell.

- décodage de la ligne :

  la fonction decode_ligne permet de séparer la ligne en différentes mots, puis en différentes commandes, tout en déterminant la présences de redirections ou de pipes. Chaque commande est stockée dans le tableau "commandes". La fonction retourne le nombre de commandes.

- Exécution de la commande

  Pour l'exécution de la commande, on utilise la fonction execute_commande. Ensuite, on sépare en deux cas possibles : la présence de pipes nous oblige à effectuer les commandes les unes à la suite des autres, alors que de multiples commandes avec un arrière plan exigent une exécution en parallèle. Il faut donc considérer chacune de ces possibilités.

  Les lignes 36 à 42 correspondent à l'exécution de pipes : on commence par exécuter la première commande (dont le descripteur de fichier de sortie correspond au descripteur de fichier de l'entrée du pipe), puis pour chaque pipe, on commence par fermer l'entrée du pipe, avant d'exécuter la commande, puis de fermer la sortie du pipe.

  Dans l'autre cas, il s'agit de l'exécution simple de la (ou des) commande(s).

- fin de la boucle :

  Les lignes 60 à 67 ne sont plus dans la boucle. Dans ce cas, on affiche le message de fin (Ciao), puis on libère la mémoire allouée : les commandes ainsi que la ligne.

3.1.3 Fonctions utilitaires du shell

- Fonction découper :

  C'est la fonction du cours :

```
1  // Fonction decouper, pour transformer une chaine de caractères en tableau de mots
2  int decouper(char * ligne, char * sep, char * mot[], int mots_max){
3      int i;
4
5      mot[0] = strtok(ligne, sep);
6      for (i=1; mot[i-1] !=0 ; i++){
7          if(i == mots_max){
8              fprintf ( stderr, "Erreur dans la fonction decouper: trop de mots\n");
9              mot[i-1] = 0;
10             break;
11         }
12         mot[i] = strtok(NULL, sep);
13     }
14     return i;
15 }
```

- Fonction decode_ligne

  C'est la fonction principale. En effet, du décodage de la ligne dépend tout le traitement. La fonction décode ligne analyse la liste de mots donnée par la fonction déocuper. La chaîne de caractères est séparée en commandes.

```
1  // Fonction decode_ligne
2  // Rôle : analyser une ligne de code pour pouvoir l'utiliser dans le shell
3  // Arguments : mots : liste de mots de la ligne de codes
4  // commandes : liste de commandes
5  // ptr_stdin : pointeur vers le fd d'entrée
6  // ptr_stdout : pointeur vers le fd de sortie
7  // ptr_pipe : pointeur vers l'int pipe
8  // ptr_arrierePlan : pointeur vers l'int arrierePlan
9  // retour : le nb de commandes
10
11 int decode_ligne(char * mots[], ptr_com commandes[MaxCommandes], int* nb_pipe,int pipeList[
       MaxCommandes-1][2], int* arrierePlan, int nb_args, int* m_if){
12     int nb_commandes =1;
13     int idx_commande=0;
14     char * mots_cherches[] = {"&","<",">",">>","|"};
15     int test_mot, temp;
16     for(int i=0;i<nb_args && mots[i];i++){
17         test_mot = compare_mots(mots[i], 0, mots_cherches,5);
18
19         switch(test_mot){
20             case 0 : // cas de &
21                 if(*nb_pipe !=0) {
22                     perror("& incompatible avec |");
23                     return 0;
```

```
24                  }
25                  if(nb_commandes ==MaxCommandes){
26                      perror("nombre de commandes max atteint ");
27                      return 0;
28                  }
29                  // On augmente le nombre de commandes si le & n'est pas en dernière position
30                  if(i != nb_args-1) {
31                      nb_commandes++;
32                      idx_commande=0;
33                  }
34                  // on fixe l'arrière plan à 1 (true)
35                  *arrierePlan = 1;
36                  // pas d'ajout du mot "&" dans une des commandes.
37                  break;
38              case 1 : // cas de <
39                  commandes[nb_commandes-1] -> fd_in = open(mots[i+1], O_RDONLY);
40                  // test de l'ouverture du fichier
41                  if(commandes[nb_commandes-1] -> fd_in <0) {
42                      // si erreur d'ouverture : on l'affiche
43                      perror("erreur d'ouverture de fichier");
44                      // et on réinitialise le stdin pour éviter les erreurs
45                      commandes[nb_commandes-1] -> fd_in=STDIN_FILENO;
46                  }
47                  // on incrémente i pour supprimer l'adresse
48                  i++;
49                  break;
50              case 2 : // cas de >
51                  // Ce mode doit effacer le fichier s'il existe :
52                  unlink(mots[i+1]);
53                  commandes[nb_commandes-1] -> fd_out = open(mots[i+1], O_WRONLY | O_CREAT,
                        S_IRWXU);
54                  if(commandes[nb_commandes-1] -> fd_out <0) {
55                      // si erreur d'ouverture : on l'affiche
56                      perror("erreur d'ouverture de fichier");
57                      // et on réinitialise le stdin pour éviter les erreurs
58                      commandes[nb_commandes-1] -> fd_out =STDOUT_FILENO;
59                  }
60                  // on incrémente i pour supprimer l'adresse
61                  i++;
62                  break;
63              case 3 : // cas de >>
64                  commandes[nb_commandes-1] -> fd_out = open(mots[i+1], O_WRONLY | O_APPEND |
                        O_CREAT, S_IRWXU);
65                  if(commandes[nb_commandes-1] -> fd_out<0) {
66                      // si erreur d'ouverture : on l'affiche
67                      perror("erreur d'ouverture de fichier");
68                      // et on réinitialise le stdin pour éviter les erreurs
69                      commandes[nb_commandes-1] -> fd_out=STDOUT_FILENO;
70                  }
71
```

```
72                  // on incrémente i pour supprimer l'adresse
73                  i++;
74                  break;
75              case 4 : // cas de |
76                  // conflit avec & :
77                  if(*arrierePlan !=0){
78                      perror("& incompatible avec |");
79                      return 0;
80                  }
81                  // nb max pipes
82                  if(*nb_pipe==MaxCommandes-1){
83                      perror("2 pipes max dans ce shell");
84                      return 0;
85                  }
86                  // création du pipe :
87                  temp = pipe( pipeList[*nb_pipe]);
88                  if(temp !=0){
89                      perror("erreur de création du pipe");
90                      return 0;
91                  }
92                  commandes[nb_commandes-1]-> fd_out = pipeList[*nb_pipe][1];
93                  // on change le numéro de la commande
94                  nb_commandes++;
95                  idx_commande=0;
96                  commandes[nb_commandes-1]-> fd_in = pipeList[*nb_pipe][0];
97
98                  // et on augmente le nombre de pipes
99                  *nb_pipe +=1;
100                 break;
101             default :
102                 // autres cas, on ajoute le mot dans la commande courante.
103                 commandes[nb_commandes-1]->ligne [idx_commande] = mots[i];
104                 idx_commande++;
105         }
106     }
107     return nb_commandes;
108 }
```

La fonction commence par rechercher dans la ligne les caractères spéciaux traduisant la présence de pipes ou de redirections.

Quand l'un de ces caractères trouvés, on applique l'effet (switch ligne 19). Dans le cas des redirections, l'effet se limite à modifier le fd de sortie ou d'entrée de la commande, en fonction des mots situés immédiatement après (une erreur peut être retournée si le fd désigne un fichier inexistant ou interdit en lecture pour l'entrée, ou un fichier non disponible en écriture pour la sortie). Pour l'arrière plan, on incrémente le nombre de commandes, puis on change la commande en cours. Même chose pour les pipes, en y ajoutant aussi le changement de fd de sortie de la commande en cours, puis le fd d'entrée de la commande suivante.

La fonction retourne enfin le nombre de commandes.

- Fonction execute_commande

```c
// Fonction : exécute commande
// Rôle : exécuter une commande
// arguments : commande la commande à exécuter
// Retour : aucun

int execute_commande(commande commande_en_cours, char * dirs[], int arriere_plan){
    char pathname [MaxPathLength];
    int tmp;

    if(commande_en_cours.ligne [0] == 0) //ligne vide
        return 0;

    //implémentation des commandes internes
    if(!strcmp(commande_en_cours.ligne [0],"moncd")){
        // Pour la fonction moncd, on n'a pas besoin d'avoir "NULL" du tableau mot.
        int nbmots = 0;
        while(commande_en_cours.ligne[nbmots]){
            nbmots++;
        }
        // On considère donc les n-1 mots.
        moncd(nbmots,commande_en_cours.ligne);
        return 0;
    }
    if (!strcmp(commande_en_cours.ligne[0],"monexit")){
        return 1;
    }

    tmp = fork(); // création du processus enfant
    if(tmp < 0){
        perror("fork");
        return 0;
    }

    if(tmp != 0){//processus parent
        if(! arriere_plan) while(wait(0) != tmp);
        return 0;
    }

    // Processus enfant, on exécute la commande


    for(int i=0; dirs[i];i++){

        snprintf(pathname, sizeof(pathname), "%s/%s", dirs[i], commande_en_cours.ligne[0]);
        spawn(pathname, commande_en_cours.ligne,commande_en_cours.fd_in,commande_en_cours.fd_out);
    }

    return 0;

```

```
50 }
```

Cette fonction prend en argument un objet commande (c'est à dire avec une ligne de code, ainsi que deux attributs de type int qui correspondent aux fd d'entrée et de sortie). Les deux autres arguments sont le tableau des des répertoires du Path, ainsi que la présence ou non d'arrière plan.

Il y a plusieurs cas possibles lors du traitement de la commande en cours :

- ligne vide
  la fonction retourne alors 0 sans rien exécuter.
- commande interne
  Les deux cas "monexit" et "moncd" sont gérés à ce niveau et la commande renvoie 1 pour "monexit", signifiant ainsi la fin de la boucle de lecture du shell. Pour moncd, on exécute le code contenu dans la suite de la fonction. Toutefois, comme la fonction spawn n'est pas appelée pour résoudre ce cas, le dernier mot de la ligne ne doit pas être NULL. C'est pour cette raison qu'on compte d'abord le nombre de mots de la commande. (cf commandes internes pour une explication plus développée).
- cas normal
  Dans le cas normal, la fonction crée un processus enfant, puis exécute la commande à l'aide de la fonction spawn. S'il y a un arrière-plan, la fonction n'attend pas la réponse du processus enfant pour continuer. Dans le cas contraire, la fonction attend le résultat du processus enfant pour continuer.

- Fonction spawn

```
1  // Fonction spawn (wrapper pour execv)
2  int spawn(char * path, char * args[], int fdIn, int fdOut){
3      redirect(fdIn, STDIN_FILENO);
4      redirect(fdOut, STDOUT_FILENO);
5      //On peut ensuite exécuter avec execv, qui permet d'ajouter un nombre variable d'arguments à exec
6
7      execv(path,args);
8
9      // si l'execv est exécuté, on sort de la fonction spawn, donc pas de retour.
10
11
12     //gestion des erreurs : on n'affiche rien si la commande n'est pas réalisée
13
14     return 1;
15 }
16
17 void redirect(int new, int old){
18     if (old != new){
19         int dup = dup2(new, old);
20         if(dup == -1) perror("erreur de dup");
21     }
22 }
```

La fonction spawn est juste un wrapper pour execv. Elle utilise la fonction redirect, qui change le flux d'entrée ou de sortie dus processus enfant.

- Fonction compare_mot

```
// Fonction compareMots
// Rôle : compare un mot entré à une liste de mot
// arguments : char * mot entré, char * []liste de mots
// Retour : l'index du mot, ou -1 si le mot n'est pas dans la liste.

int compare_mots(char * mot_cherche, int idx, char * listeMots[], int longueur_liste){
    // si on arrive au bout de la liste sans avoir trouvé le mot
    if(idx == longueur_liste) return -1;
    // si le mot suivant de la liste est le mot recherché
    if( strlen(mot_cherche)> 0 && !strcmp(mot_cherche, listeMots[idx])) return idx;
    // Sinon, on lance la récursion
    return compare_mots(mot_cherche, idx+1, listeMots, longueur_liste);
```

Fonction dont le rôle est de rechercher un mot dans une liste de mots (utilisée pour rechercher les symboles spéciaux dans la ligne de commande).

- Fonctions d'initialisation

  Comme le code est en c, il faut allouer la mémoire nécessaire pour la création de la structure "commande". Pour ce faire, on utilise le code suivant :

```
// Fonction : ini_commande
// Rôle : initilisation d'une structure commande pour pouvoir la modifier ensuite.

ptr_com init_commande(){
    ptr_com new_command = (ptr_com) malloc(sizeof(commande));
    new_command -> fd_in = STDIN_FILENO;
    new_command -> fd_out = STDOUT_FILENO;
    *new_command -> ligne = *initArray(MaxMot);
    return new_command;
}


char ** initArray(int array_length){
    static char * array[MaxMot];
    for(int i=0; i< array_length;i++){
        array[i]=NULL;
    }
    return array;
```

- Fonctions d'autocomplétion :

  L'autocomplétion des noms de fichiers se fait automatiquement avec l'utilisation de la librairie readline. Pour l'autocomplétion de mes noms de fonctions, j'ai utilisé le code disponible à cette adresse : https://thoughtbot.com/blog/tab-completion-in-gnu-readline.

```

// Variable des noms de commandes pour l'autocomplétion
char* my_function_names[] = {
        "monCp",
        "monkill",
```

```
6           "monMan",
7           "monsleep",
8           "monexit",
9           "moncd",
10          NULL
11  };
12
13  char ** function_name_completion(const char * text, int start, int end){
14      rl_attempted_completion_over = 1;
15      return rl_completion_matches(text, function_name_generator);
16  }
17
18  char * function_name_generator(const char * text, int state){
19      static int list_index, len;
20      char *name;
21
22      if(! state) {
23          list_index = 0;
24          len = strlen(text);
25      }
26
27      while ((name = my_function_names[list_index++])) {
28          if (strncmp(name, text, len) == 0){
29              return strdup(name);
30          }
31      }
32
33      return NULL;
34  }
```

Ce code se base sur l'utilisation de la librairie readline. Il est accompagné d'une ligne dans le code du fichier principal (afin d'affecter la fonction de comparaison).

```
1   // auto-completion : ajout des noms de fonctions
2   rl_attempted_completion_function = function_name_completion;
```

L'event listener appelle la fonction function_name_completion, qui n'est qu'un wrapper pour rl_completion_matches. Cette dernière regarde dans la liste des noms de commandes (my_function_names) afin de tester la compatibilité du début de mot avec chacun des termes de la liste (le test s'effectue à l'aide de la fonction function_name_generator).

Si aucun cas ne matche, ou si plusieurs cas matchent, l'autocomplétion ne fonctionne pas.

En revanche, si un seul cas matche, il est autocomplété.

3.2 Commandes

3.2.1 Commandes internes

Rappel : les commandes internes sont implémentées dans le fichier util.c (à la fin, section "commandes internes").

- moncd

Suite aux résultats de l'exercice 2.16, on a pu constater que la commande moncd ne doit pas être exécutée comme les autres commandes dans un processus enfant, car elle modifierait alors l'environnement du processus enfant, mais pas celui du processus parent (celui qui nous intéresse). Le code de la fonction moncd doit donc être exécuté dans le shell, ce qui signifie une implémentation directe dans le code même du shell (section "commandes internes" à la fin du code du shell). Le code de la fonction reste quasiment inchangé. Le code du shell toutefois va être modifié. En effet, on va différencier les commandes internes des commandes externes. Les commandes internes ne nécessitent pas la création d'un fork. Elles doivent donc être gérées avant cette création, sur le même modèle que la gestion d'une ligne vide.

```
1  // Fonction moncd pour changer de répertoire
2  int moncd(int ac, char * commande[]){
3      char * dir;
4      int t;
5
6      // traitement des arguments
7      if(ac < 2){
8          dir = getenv("HOME");
9          if (dir == 0)
10             dir = "\temp";
11     }
12     else if (ac > 2){
13         fprintf ( stderr, "usage: %s [dir] ne doit avoir qu'un argument\n", commande[0]) ;
14         return 1;
15     }
16     else{
17         dir = commande[1];
18     }
19
20     // changement de répertoire
21     t = chdir(dir) ;
22     if ( t < 0){
23         perror(dir) ;
24         return 1;
25     }
26     return 0;
27 }
```

```
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre 2$ shell/c-main.2
?pwd
/home/pierred/Documents/ied/L2/SE/chapitre 2
?moncd shell
?pwd
/home/pierred/Documents/ied/L2/SE/chapitre 2/shell
?moncd ..
?pwd
/home/pierred/Documents/ied/L2/SE/chapitre 2
?moncd
?pwd
/home/pierred
?moncd documents
documents: No such file or directory
?pwd
/home/pierred
?moncd Documents deuxieme_argument
usage: moncd [dir] ne doit  avoir qu'un argument
?
```

Les tests réalisés ont tous la même structure : affichage du répertoire courant avec la commande pwd, changement de répertoire avec moncd et vérification du nouveau répertoire courant avec pwd.

On peut voir dans le premier exemple que moncd fonctionne pour aller vers un répertoire fils. Le deuxième exemple montre que la commande fonctionne pour aller vers le répertoire parent. Le troisième exemple montre que sans argument, la commande redirige vers le répertoire de l'environnement $HOME. Le quatrième exemple montre la redirection vers un répertoire inexistant et le dernier exemple utilise une commande incorrecte avec deux arguments au lieu d'un seul. Les résultats obtenus paraissent conformes aux spécifications demandées.

- monexit

  La commande monexit n'est pas une fonction à proprement parler. Elle est gérée par la fonction execute_commande, sous forme d'une simple condition (ligne 24 de la fonction à la section précédente).

```
1  if (!strcmp(commande_en_cours.ligne[0],"monexit")){
2          return 1;
3      }
```

```
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre 2/shell$ ./c-main.2
?echo test de la commande monexit
test de la commande monexit
?monexit
Ciao
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre 2/shell$ ./c-main.2
?monexit avec des arguments après
Ciao
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre 2/shell$ ./c-main.2
?arguments avant monexit
?monexit
Ciao
```

3.2.2 Commandes externes

Les commandes externes sont situées dans le dossier external du projet. Ce dossier a été ajouté à la liste des dossiers du PATH dans la fonction principale pour que le shell puisse y avoir accès. En outre, les exécutables de ces commandes externes sont gérées par le makefile.

- monsleep

  Il s'agit du programme du cours (qui fait appel à une fonction définie en local et non à la fonction bash sleep).La fonction n'est donc pas de moi, mais de vous. A priori, il n'y a donc pas besoin de commentaires sur la réalisation.

```
1  /* monsleep.c
2  Une re-implementation de la commande sleep
3  v2.0 : accepte plusieurs arguments.
4  Auteur : Pierre DAHERON
5  Contexte :cours de système d'exploitation IED P8
6  */
7
8  // Compilation : gcc -g -Wall monsleep.c f-monsleep.c
9
10 # include <stdio.h>
11 # include <stdlib.h>
12 # include <ctype.h>
13 # include <unistd.h>
14
15 static void invalid( char * prog, char * s) {
16     fprintf ( stderr, "%s: invalid time interval %s\n", prog, s) ;
17     exit (1) ;
18 }
19
20 int main(int ac, char * av[]) {
21     int i ;
22     int nsec;
23     if ( ac < 2){
24         fprintf ( stderr, "usage: %s N\n", av[0]);
25         return 1;
26     }
27     for(int j=1;j<ac;j++){
28
29
30         for( i = 0; isdigit ( av[1][ i ]) ; i ++)
31         ;
32         if (! isdigit ( av[j][0]) ) invalid( av[0], av[j]) ;
33         nsec = atoi ( av[j]) ;
34         switch(av[j][i ]) {
35             default:
36                 invalid( av[0], av[j]) ;
37             case 'd':
38                 nsec *= 24*3600;
39             case 'h':
40                 nsec *= 3600;
41             case 'm':
42                 nsec *= 60;
43             case 's' :
44             case 0:
45             ;
```

```
46          }
47          sleep(nsec);
48      }
49      return 0;
50  }
```

La fonction sleep locale (appelée dans la compilation) est la suivante (également celle du cours):

```
1   /* f-monsleep.c
2   Une re-implementation de la fonction sleep
3   Attention time() est invalide en C99
4   */
5
6
7
8   # include <stdio.h>
9   # include <unistd.h>
10  # include <signal.h>
11  #include <time.h>
12
13  void dring(int n){
14  /* rien a faire */
15  }
16
17  unsigned int sleep(unsigned int nsec){
18      int avant, apres;
19      avant = time(0);
20      signal( SIGALRM, dring);
21      alarm(nsec);
22      pause();
23      apres = time(0);
24      return ( avant + nsec) - apres;
25  }
```

Résultats obtenus :

```
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_10$ ./monShell
?monsleep
usage: monsleep N
?monsleep 2
?monsleep 1m
```

- monkill

```
1   #include <sys/types.h>
2   #include <signal.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6
```

```c
typedef char* string ;
// Quelques uns des signaux utilisés :
typedef struct {string nom; int num;} Signal;
Signal SIGNAUX[8] = {{"SIGHUP",1}, {"SIGINT",2},{"SIGQUIT",3}, {"SIGKILL",4},{"SIGABRT",6}, {"SIGKILL",9},{"SIGALARM",14},{"SIGTERM",15} };


// Prototypes :
int usage(string);
int option(string, int*);


int main(int ac, string argv[]){
    if (ac <2) usage("usage : kill [pid]");
    int i=1;
    int signal = 15; // défaut : SIGTERM
    while (i < ac){
        if(argv[i][0]=='-') option(argv[i], &signal);
        else{
            if (atoi(argv[i])==0){
                // si le pid n'est pas un nombre
                usage("usage : les pid doivent être des nombres");
            }
            else{
                kill(atoi(argv[i]), signal);
            }
        }
        i++;
    }




    return 0;
}

int usage(string message){
    printf("%s\n", message);
    exit(1);
}

int option(string option, int* signal){
    if(! strcmp(option,"-l")){
        for(int i=0;i<8;i++){
            printf("%d) %s\n", SIGNAUX[i].num, SIGNAUX[i].nom);
        }
    }
    else{
        if(atoi(option+1) !=0){
            *signal = atoi(option+1);
```

```
56              return 0;
57          }
58          for(int i=0; i< 8; i++){
59              if(!strcmp(option+1, SIGNAUX[i].nom)){
60                  *signal = SIGNAUX[i].num;
61                  return 0;
62              }
63          }
64          usage("les options disponibles sont -l pour lister les fonctions, ou -[nom_du_signal], ou -[
                num_du_signal]");
65      }
66      return 0;
67  }
```

Résultats obtenus :

```
?monsleep 1 2
?monkill -l
1) SIGHUP
2) SIGINT
3) SIGQUIT
4) SIGKILL
6) SIGABRT
9) SIGKILL
14) SIGALARM
15) SIGTERM
?ps 6391
    PID TTY      STAT   TIME COMMAND
   6391 ?        Sl     0:37 /usr/lib/firefox/firefox -new-window
?monkill 6391
?ps 6391
    PID TTY      STAT   TIME COMMAND
?monkill 6391
?monkill processus
usage : les pid doivent être des nombres
?
```

On peut constater ici que la commande sans option arrête bien le processus demandé, et que la commande avec l'option -l donne la liste des signaux implémentés. Ces exemples ont été réalisés dans le shell.

Pour vérifier le bon fonctionnement de la commande, il est préférable de sortir du shell afin de vérifier les différents appels système (ce qui est tout à fait faisable, avec des commandes externes) :

```
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_9$ ./monkill -9 14658
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_9$ ps 14658
    PID TTY      STAT   TIME COMMAND
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_9$ ps 15389
    PID TTY      STAT   TIME COMMAND
  15389 pts/2    S+     0:00 top
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_9$ strace ./monkill -SIGKILL 15389
execve("./monkill", ["./monkill", "-SIGKILL", "15389"], 0x7ffef38ab000 /* 53 vars */) = 0
brk(NULL)                               = 0x55670aa22000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc3b2d9870) = -1 EINVAL (Argument invalide)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (Aucun fichier ou dossier de ce type)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=116911, ...}) = 0
mmap(NULL, 116911, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f150aec1000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360A\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\237\333t\347\262\27\320l\223\27*\202C\370T\177"...
fstat(3, {st_mode=S_IFREG|0755, st_size=2029560, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f150aebf000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\237\333t\347\262\27\320l\223\27*\202C\370T\177"...
mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f150accd000
mmap(0x7f150acef000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x
mmap(0x7f150ae67000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19a000) = 0
mmap(0x7f150aeb5000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1
mmap(0x7f150aebb000, 13920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0)
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f150aec0540) = 0
mprotect(0x7f150aeb5000, 16384, PROT_READ) = 0
mprotect(0x556708cd5000, 4096, PROT_READ) = 0
mprotect(0x7f150af0b000, 4096, PROT_READ) = 0
munmap(0x7f150aec1000, 116911)          = 0
kill(15389, SIGILL)                     = 0
exit_group(0)                           = ?
+++ exited with 0 +++
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_9$ ps
    PID TTY          TIME CMD
   8906 pts/1    00:00:00 bash
  15558 pts/1    00:00:00 ps
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_9$ ps 15389
    PID TTY      STAT   TIME COMMAND
pierred@pierred-IdeaPad:~/Documents/ied/L2/SE/chapitre_9$
apitre_9)
```

Sur ces deux exemples, on constate que l'appel avec l'option -9 arrête également le processus. Dans le dernier exemple, j'ai utilisé strace pour bien montrer que c'est le signal SIGKILL et non SIGTERM (par défaut) qui est appelé. On peut constater à l'antépénultième ligne du résultat de strace que l'appel système kill est bien effectué avec SIGKILL.

- monMan

  La commande monMan permet d'afficher les pages du manuel situées dans le dossier man_pages.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
```

```
4
5  int main(int argc, char * argv[]){
6      if (argc !=2) perror("usage : monMan [nom_de_commande] - Commandes documentées : moncd,
           monCp, monexit, monkill, monMan, monsleep");
7      char * man_title="man_pages/man_";
8      size_t taille = strlen(man_title)+strlen(argv[1]);
9      char * man_file =(char *) malloc(taille);
10     strcpy(man_file, man_title);
11     strcat(man_file, argv[1]);
12
13     FILE *f;
14     char c;
15     f=fopen(man_file,"rt");
16
17     if (f==NULL) perror("erreur d'ouverture de fichier");
18     while((c=fgetc(f))!=EOF){
19         printf("%c",c);
20     }
21
22     free(man_file);
23     fclose(f);
24     return 0;
25
26 }
```

La fonction principale ouvre le fichier à la demande. Les fichiers textes ont tous la même racine : ils sont situés dans le répertoire man_pages et leurs noms commencent par man_ suivis du nom de la commande.

En cas d'erreur de l'ouverture du fichier, la fonction s'arrête et renvoie une erreur. Dans le cas contraire, on affiche le contenu du fichier.

```
?monMan Man
erreur d'ouverture de fichier: No such file or directory
?monMan exit
erreur d'ouverture de fichier: No such file or directory
?monMan monMan
monMan(1)

Nom :
        monMan - accède à la page du manuel correspondante

Synopsis :
        monMan [commande]

Description :
        Accède à la page du manuel correspondante. Un et un seul argument attendu.


        Pas d'options (les pages des fonctions ou de signaux systèmes ne sont pas documentées).



Auteur :
        Pierre Daheron

Bugs connus :
?
```

Ici, on a deux exemples d'erreur, ainsi que l'exemple d'ouverture réussie de la page monMan

- monCp

  La commande monCp copie un fichier d'un répertoire vers un autre. Le premier argument est le chemin du fichier à copier, le second étant le nom du fichier copié, avec son chemin.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char * argv[]){
6      //usage :
7      if(argc !=3) perror("usage : monCp [path_to_copy] [path_to_paste], le fichier à copier doit être existant"
           );
8
9      // variables
10     FILE *f_in, *f_out;
11     char c;
12
13     //ouverture du fichier à lire
14     f_in=fopen(argv[1],"rt");
15     // test de l'ouverture
16     if (f_in==NULL) {
17         perror("erreur d'ouverture de fichier");
18         exit(1);
19         }
20
21     // ouverture du fichier à écrire
22     f_out = fopen(argv[2],"w");
23     // test de l'ouverture
24     if(f_out==NULL){
25         perror("erreur en écriture");
26         exit(1);
27     }
28
29     // Boucle de copie d'un fichier à l'autre
30     while((c=fgetc(f_in))!=EOF){
31         fprintf(f_out,"%c",c);
32     }
33
34     // Fermeture des deux fichiers.
35     fclose(f_in);
36     fclose(f_out);
37     return 0;
38 }
```

Le code est assez simple : on ouvre un fichier en écriture et un en lecture. On recopie le fichier en lecture dans celui en écriture, puis on ferme les deux flux.

Des erreurs peuvent apparaître si le fichier à copier n'existe pas ou si le fichier destination n'est pas autorisé en écriture.

```
?ls
external        Makefile   monShell   SE-10.fdb_latexmk  SE-10.out  SE-10.synctex.gz  SE-10.toc  shell-final.c  util.c
fichier_images  man_pages  SE-10.aux  SE-10.log          SE-10.pdf  SE-10.tex         SE-10.xdv  sys.h
?ls external
f-monsleep.c  monCp  monCp.c  monkill  monkill.c  monMan  monMan.c  monsleep  monsleep.c  test  test2
?monCp external/monCp.c test.c
?ls
external        Makefile   monShell   SE-10.fdb_latexmk  SE-10.out  SE-10.synctex.gz  SE-10.toc  shell-final.c  test.c
fichier_images  man_pages  SE-10.aux  SE-10.log          SE-10.pdf  SE-10.tex         SE-10.xdv  sys.h          util.c
?monCp monCp.c external/test.c
erreur d'ouverture de fichier: No such file or directory
?monCp test.c external
erreur en écriture: Is a directory
?
```

3.3 Gestion de la mémoire

Le programme étant implémenté en c, il convient de faire attention à la gestion de la mémoire. A cette fin, les structures de type "commande" qui utilisent de la mémoire allouée (malloc) doivent être libérés, de même que le buffer contenant la ligne à décoder.

Afin de vérifier que la mémoire est bien libérée, j'ai utilisé le logiciel valgrind, qui détermine les fuites de mémoire d'un programme

Le résultat obtenu est le suivant :

```
==22297== LEAK SUMMARY:
==22297==    definitely lost: 0 bytes in 0 blocks
==22297==    indirectly lost: 0 bytes in 0 blocks
==22297==      possibly lost: 0 bytes in 0 blocks
==22297==    still reachable: 200,556 bytes in 214 blocks
==22297==         suppressed: 0 bytes in 0 blocks
```

On peut constater qu'aucune fuite de mémoire définitive n'est à déplorer. La mémoire "still reachable" perdue est due à l'utilisation de readline. C'est en réalité de la mémoire qui n'est pas libérée à la fin du programme et qui pourra l'être lorsque le terminal sera fermé. Il ne s'agit pas a proprement parler d'une fuite de mémoire comme l'indique cette discussion (qui porte sur le même problème et où l'on peut constater que la quantité de mémoire indiquée est du même ordre de grandeur) : https://stackoverflow.com/questions/55196451/gnu-readline-enormous-memory-leak.

10-4

Difficultés rencontrées

Je n'ai pas l'impression d'avoir rencontré beaucoup de difficultés lors de ce projet, sûrement parce qu'il s'étale sur une grande partie de l'année. Le chapitre sur les pipes m'a posé plus de problèmes : j'avais d'abord testé avec des fichiers tampons, avant de me rabattre sur l'idée des fd, plus facile à mettre en oeuvre.

La dernière difficulté que je n'ai pas su résoudre, c'est le "monIf". En effet, cette commande fonctionne dans le bash en différenciant les textes et les commandes (à l'aide des guillemets). Le shell que nous avons créé ne prend pas en compte cette différence.

10-5

Conclusion et perspectives

En conclusion, je peux dire que j'ai compris beaucoup de choses sur le fonctionnement du shell bash en essayant d'en recréer un. Toutefois, après tout ce travail, j'ai bien conscience que le shell que nous avons réalisé reste limité.

En premier lieu, le shell ne fait pas la différence entre le texte et les objets, ce qui peut poser problème. Le shell bash permet de faire la différence entre les deux via les guillemets.Ensuite, dans notre shell, même si on a produit quelques commandes, on n'a aucun appel système, et que très peu de gestion des signaux.

Ce shell reste donc rudimentaire, mais il est fonctionnel et pourrait facilement être étoffé par plusieurs commandes externes. L'implémentation de l'évaluation du texte fourni (différenciation du texte et des commandes) pourrait être un plus. On pourrait également améliorer la gestion des signaux.