



Département sciences du numérique

Systèmes d'exploitation centralisés

L'objectif de ce rapport est de détailler les étapes de conception d'un mini-shell basé sur le shell Linux.

Langage utilisé : C

Code source en annexe

Florent Puy

2023

Partie 1 – Boucle et principes de base

L'objectif de cette partie est de créer un minishell simple basé sur une boucle infinie.

La boucle implémentée est composée d'un **while(1)** , lit ce que l'utilisateur entre en ligne de commande puis crée un processus fils avec **fork()** pour l'exécuter.

Comme l'indique la question 2, avec cette méthode le processus shell lance un fils, puis se met immédiatement en attente de lecture de la prochaine commande.

Par exemple, si nous décidons de rajouter le chemin et le nom de l'utilisateur en début de ligne de commande avec le code suivant avant le **fork()** :

```
//affichage du chemin courant
char* s = malloc(200); //reserve 200 octets pour le chemin
printf("florent:");
printf("%s", getcwd(s,200));
free(s); //libere la memoire
printf(" >>>");
```

Alors l'exécution du processus fils passe après la mise en attente de la prochaine commande :

```
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ ./minishell5
florent:/home/florent/Documents/N7_1A/S6/sec/projet >>>ls
florent:/home/florent/Documents/N7_1A/S6/sec/projet >>>LisezMoi.html  Makefile
minishell5.c  readcmd.c  test_readcmd.c
LisezMoi.md  minishell5  minishell.tar  readcmd.h
```

Le chemin est affiché deux fois consécutives puis pas du tout après ce qui n'était pas l'objectif initial.

Pour palier à ce problème, on rajoute les lignes suivantes dans le cas où le **fork()** s'est bien déroulé :

```
//Attente de la fin du processus fils
int status;
waitpid(pid, &status, 0);
```

On a alors le résultat suivant pour l'exécution de plusieurs **ls** :

```
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ ./minishell5
florent:/home/florent/Documents/N7_1A/S6/sec/projet >>>ls
LisezMoi.html  Makefile  minishell5.c  readcmd.c  test_readcmd.c
LisezMoi.md    minishell5  minishell.tar  readcmd.h
florent:/home/florent/Documents/N7_1A/S6/sec/projet >>>ls
LisezMoi.html  Makefile  minishell5.c  readcmd.c  test_readcmd.c
LisezMoi.md    minishell5  minishell.tar  readcmd.h
```

On désire désormais ajouter les commandes **exit** et **cd** sans lancer de processus fils.

Pour cela, nous devons ajouter les conditions suivantes avant le **fork()** .

```
//exit
if (strcmp(commande, "exit") == 0) {
    break ;
}

//cd
if (strcmp(commande, "cd") == 0) {
```

```

        chdir (getenv ("HOME"));
    }

```

Cette implémentation de la commande `cd` ne permet que de revenir à la racine (**HOME**) car aucun argument n'est pour l'instant traité.

Enfin, nous allons implémenter la possibilité pour l'utilisateur d'exécuter une commande en tâche de fond grâce au caractère `'&'` en fin de ligne de commande.

Pour cela, on vérifie tout d'abord si le dernier caractère est `'&'`:

```

// Vérifier si le dernier caractère est '&'
int bg = 0;
if (taille > 0 && (commande[taille-1] == '&')) {
    bg = 1;
    commande[taille] = '\0';
}

```

Avec `bg` pour background la variable booléenne égale à **1** si la commande doit être exécutée en arrière-plan, **0** sinon et `taille` représentant la taille de la commande :

```
int taille = strlen(commande);
```

Enfin, la commande est exécutée en arrière plan si elle n'est pas prioritaire. On exécute donc le `waitpid` seulement quand la commande n'est pas en tâche de fond :

```

// Attendre la fin du processus fils si la commande n'est pas en tâche de fond
if (bg == 0) {
    int status;
    waitpid(pid, &status, 0);
}

```

Partie 2 – Gestion des processus

Dans cette partie, nous allons implémenter une série de fonctions et commandes ayant pour but de gérer les processus lancés depuis le minishell.

1) Gestion des tâches

J'ai implémenté un type `job` contenant son identifiant local, son PID, si il est actif ou suspendu, si il est mort ou non et la commande qui l'a démarré.

```

struct job {
    int job_id;
    pid_t pid;
    int active;
    char commande[MAX_LIGNE];
};

```

Pour gérer les tâches, j'ai implémenter un tableau de `MAX_JOBS = 100` éléments de type `job`. J'ai implémenté différentes fonctions pour gérer ce tableau :

– `void add_job(pid_t pid, char* commande)` qui ajoute un job. Si le tableau `JOBS` est plein, il regarde si il y a un job mort et le remplace.

- `void stop_job(int job_id)` qui stop un job (`actif→0`) et exécutant `kill(jobs[i].pid, SIGSTOP);`
- `void reprendre_job(int job_id)` qui fait reprendre en tache de fond un job (`actif→1`) et exécute `kill(jobs[i].pid, SIGCONT);`
- `void suppr_job(int job_id)` qui supprime un processus de `jobs` à partie de son `id`.
- `void list_jobs()` qui liste les processus.

Lorsqu'une tache se termine, j'intercepte le signal `SIGCHLD` et exécute la fonction `handler_sigchild` appelant la fonction `suppr_jobs(int job_id)`.

2) La commande `lj` :

Cette commande exécute simplement la fonction `list_jobs()`.

3) La commande `sj` :

Cette commande exécute la fonction `stop_jobs()` exécutant `kill(jobs[i].pid, SIGSTOP);` pour interrompre le processus.

4) La commande `bg`

Cette commande exécute la fonction `reprendre_jobs(int job_id)`.

5) La commande `fg`

Cette commande exécute `kill(jobs[i].pid, SIGCONT);` puis, au moyen d'un `waitpid`, attend que la tache se termine. C'est ce qui caractérise les exécutions au premier plan. Enfin, elle exécute la fonction `suppr_jobs(int job_id)` car la tache sera alors finie.

On peut vérifier quelques unes de ces fonctions avec la commande `sleep` comme ci-dessous :

```
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ gcc minishell.c -o minishell
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ ./minishell
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>sleep 5 &
Tache en arriere plan: 35020      active  sleep
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>lj
Job ID  PID      State  Commande
1       35020    Actif  sleep
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>lj
Job ID  PID      State  Commande
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>
```

`sleep 5` est ici exécuté en arrière plan. Il apparaît dans le tableau des taches pendant 5s et ensuite y est supprimé comme on peut le voir sur le second `lj` exécuté 6s après.

Partie 3 – Gestion des signaux

Nous créons ici une variable globale `int processus_courant` égale à 0 lorsqu'aucun processus n'est en cours d'exécution et sinon égale au PID du processus en cours d'exécution.

7) SIGSTP

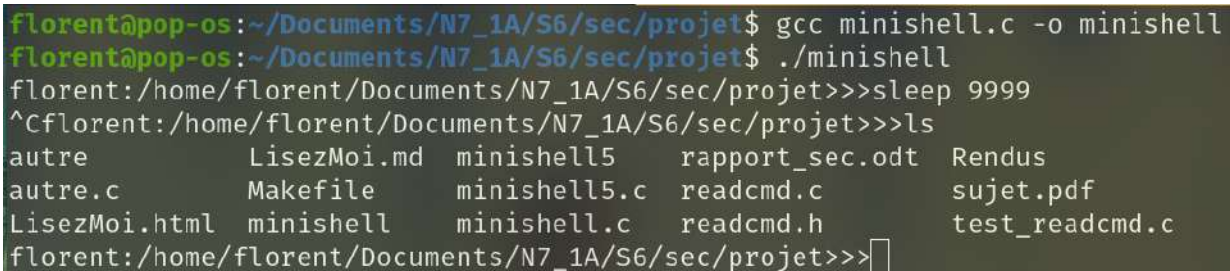
Le signal **SIGSTOP** ou un `ctrl+z` doivent suspendre le processus en cours d'exécution. Pour cela, j'ai choisi de faire ignorer **SIGSTOP** aux fils et ajouter un handler pour ce signal dans le père qui enverra un signal pour suspendre le processus.

La commande `susp` est fonctionnelle en interrompant le minishell. Elle envoie un signal **SIGSTOP** depuis le père :

```
else if (strcmp(commande->seq[0][0], "susp")==0) {
    kill(processus_courant, SIGSTOP);
}
```

8) SIGINT

Pour traiter le `ctrl+C` j'ai écrit un handler au signal **SIGINT** envoyant un signal **SIGKILL** au dernier processus démarré encore actif au premier plan et le supprimant du tableau des tâches avec la fonction `supr_job(int pid)`. Dans le cas où aucun processus n'est en cours d'exécution j'affiche un message disant qu'il faut taper `'exit'` pour quitter le minishell.



```
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ gcc minishell.c -o minishell
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ ./minishell
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>sleep 9999
^Cflorent:/home/florent/Documents/N7_1A/S6/sec/projet>>>ls
autre      LisezMoi.md  minishell5    rapport_sec.odt  Rendus
autre.c     Makefile     minishell5.c  readcmd.c        sujet.pdf
LisezMoi.html minishell    minishell.c   readcmd.h        test_readcmd.c
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>
```

On voit ici l'impact du `ctrl+c` (indiqué par `^C` sur le minishell) que le `sleep` s'interrompt et une nouvelle commande peut être tapée.

Partie 4 – Gestion des redirections

9) Gestion des redirections

A ce stade là, j'ai rencontré un problème. Je n'utilisais pas le code `readcmd.c` donné, j'avais préféré tout faire par moi-même. Cependant, cette étape s'est avérée plus dure que prévue donc j'ai restructuré toute ma gestion de la ligne de commande pour utiliser `readcmd`.

J'ai ensuite implémenté deux fonctions : une pour rediriger l'entrée et l'autre la sortie standard vers un fichier en utilisant `dup2` et `open`:

```
//redirection de la sortie
void rediriger_sortie(char* fichier) {
```

```

int desc, dup;
desc = open (fichier, O_WRONLY| O_CREAT | O_TRUNC, 0640);
if (desc < 0) {
    perror(fichier);
    exit(1);
}
dup = dup2(desc, 1);
if (dup == -1) {
    printf("Erreur dup2 sortie \n");
    exit(1);
}
}

/
//redirection de l'entrée
void rediriger_entree(char* fichier) {
    int desc, dup;
    desc = open (fichier, O_RDONLY);
    if (desc < 0) {
        perror(fichier);
        exit(1);
    }
    dup = dup2(desc, 0);
    if (dup == -1) {
        printf("erreur dup2 entrée \n");
        exit(1);
    }
}
}

```

Ces fonctions sont appelées lorsque **commande -> in** (respectivement **out**) sont non nuls. Voici un test avec la commande **echo** et la commande **cat** pour vérifier le fichier **test.txt**.

```

florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ gcc minishell.c -o minishell
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ ./minishell
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>echo vive mfee > test.txt
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>cat test.txt
vive mfee
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>

```

Partie 5 – Implémentations des tubes

J'ai ici directement implémenté la version « pipelines ». Pour ce faire, je compte le nombre de tubes, je les crée tous : partout sauf aux extrémités : début de la première commande et fin de la dernière. Enfin, j'exécute toutes les commandes et je ferme les pipes avec **close**.

Voici un test avec deux et trois commandes :

```

florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ gcc minishell.c -o minishell
florent@pop-os:~/Documents/N7_1A/S6/sec/projet$ ./minishell
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>ls | wc -l
17
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>cat test.txt | grep int |wc -l
0
florent:/home/florent/Documents/N7_1A/S6/sec/projet>>>

```

Conclusion

Ce projet fut pour moi l'un des plus intéressant de l'année même si je ne suis pas sûr que les tubes et signaux me soient très utiles en passant en MFEE.

Je précise que j'ai laissé en commentaire toute ma gestion de la ligne de commande (gestion des arguments, background et tout ça) que j'utilisais avant de l'abandonner pour le readcmd.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <string.h>
6 #include <errno.h>
7 #include <stdbool.h>
8 #include <fcntl.h>
9 #include <signal.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include "readcmd.h"
13 #include "readcmd.c"
14
15 #define _XOPEN_SOURCE 700
16 #define _POSIX_SOURCE 1
17 #define MAX_LIGNE 100 // Taille max d'une commande
18 #define MAX_JOBS 100 // Nombre max de taches en arriere plan
19
20 // processus courant
21 pid_t processus_courant=0;
22
23 // bools pour ctrl-z et ctrl-c
24 bool ctrl_z=false;
25
26 bool ctrl_c=false;
27
28 struct job { // Structure pour un processus
29     int job_id;
30     pid_t pid;
31     int active;
32     char commande[MAX_LIGNE];
33     //int mort; // 1 si mort, 0 si vivant, si mort, on affiche pas
34 };
35
36 struct job jobs[MAX_JOBS]; // Tableau de taches en arriere plan
37 int num_jobs = 0; // Nombre de taches en arriere plan
38
39
40 // Fonction pour ajouter une tache en arriere plan
41 void add_job(pid_t pid, char* commande) {
42     jobs[num_jobs].job_id = num_jobs + 1;
43     jobs[num_jobs].pid = pid;
44     jobs[num_jobs].active = 1;
45     //jobs[num_jobs].mort = 0;
46     //strcpy(jobs[num_jobs].command, command, MAX_LIGNE);
47     strcpy(jobs[num_jobs].commande, commande);
48     num_jobs++;
49 }
50
51
52 // Fonction pour supprimer une tache en arriere plan
53 void stop_job(int job_id) {
54     for (int i = 0; i < num_jobs; i++) {
55         if (jobs[i].job_id == job_id) {
56             jobs[i].active = 0; // Marquer la tache comme inactive
57             kill(jobs[i].pid, SIGSTOP); // Suspendre le processus
58             break;
59         }
60     }
61 }
62
63 // Fonction pour reprendre une tache en arriere plan
64 void reprendre_job(int job_id) {
65     for (int i = 0; i < num_jobs; i++) {
66         if (jobs[i].job_id == job_id) {
67             jobs[i].active = 1; // Marquer la tache comme active
68             kill(jobs[i].pid, SIGCONT); // Reprendre le processus
69             processus_courant = jobs[i].pid;
70             break;
71         }
72     }
73 }
74
75 // Fonction pour reprendre une tache en arriere plan
76 void suppr_job(int job_id) {
77     processus_courant = 0;
78     for (int i = 0; i < num_jobs; i++) {
79         if (jobs[i].job_id == job_id) {
80             // Décaler toutes les tâches suivantes d'une position vers la gauche
81             for (int j = i; j < num_jobs - 1; j++) {
82                 jobs[j] = jobs[j + 1];
83             }
84             num_jobs--;
85             break;
86         }
87     }
88 }
89
90
91 //redirection de la sortie

```



```

92 void rediriger_sortie(char* fichier) {
93     int desc, dup;
94     desc = open (fichier, O_WRONLY | O_CREAT | O_TRUNC, 0640);
95
96     if (desc < 0) {
97         perror(fichier);
98         exit(1);
99     }
100
101     dup = dup2(desc, 1);
102
103     if (dup == -1) {
104         printf("Erreuuuur dup2 sortie \n");
105         exit(1);
106     }
107 }
108
109 //redirection de l'entr e
110 void rediriger_entree(char* fichier) {
111     int desc, dup;
112     desc = open (fichier, O_RDONLY);
113     if (desc < 0) {
114         perror(fichier);
115         exit(1);
116     }
117     dup = dup2(desc, 0);
118     if (dup == -1) {
119         printf("erreur dup2 entr e \n");
120         exit(1);
121     }
122 }
123
124 /*
125 void suppr_job(int job_id) {
126     //int bool = 0;
127     //if (job_id == 0) {
128     //    jobs[0] = jobs[1]; // Remplacer la tache par la derniere
129     //}
130     for (int i = 0; i < num_jobs; i++) {
131         if (jobs[i].job_id == job_id) {
132             //bool = 1;
133             jobs[i].mort = 1; //On le tue pour ne pas l'afficher
134             //jobs[i] = jobs[num_jobs-1]; // Remplacer la tache par la derniere
135             break;
136         }
137     }
138 }
139 */
140
141
142
143
144
145
146
147
148 // Fonction pour lister les taches en arriere plan
149 void list_jobs() {
150     printf("Job ID\tPID\tState\tCommande\n");
151     for (int i = 0; i < num_jobs; i++) {
152         //if (jobs[i].mort == 0) {
153             char* state = jobs[i].active ? "Actif" : "Suspendu";
154
155             printf("%d\t%d\t%s\t%s\n", jobs[i].job_id, jobs[i].pid, state, jobs[i].commande);
156         // }
157     }
158
159 // Fonction pour interceper le signal SIGCHLD
160 void handler_sigchild(int signal){
161     int wstatus;
162     pid_t pid_child;
163     do {
164         pid_child = (int) waitpid(-1, &wstatus, WNOHANG | WUNTRACED | WCONTINUED);
165         if ((pid_child == -1) && (errno != ECHILD)) {
166             perror("waitpid");
167             exit(-1);
168         } else if (pid_child > 0) {
169             if (WIFSTOPPED(wstatus)) {
170                 //Reprise de processus.*/
171                 for (int k = 0; k < num_jobs; k++) {
172                     if (jobs[k].pid == pid_child) {
173                         int i = k;
174                         jobs[i].active = 1;
175                         break;
176                     }
177                 }
178             } else if (WIFEXITED(wstatus) || WIFSIGNALED(wstatus)) {
179                 //le processus a  t  arret 
180                 for (int l = 0; l < num_jobs; l++) {
181                     if (jobs[l].pid == pid_child) {
182                         suppr_job(jobs[l].job_id);
183                     }
184                 }
185             }
186         }
187     } while (pid_child != -1);
188 }

```

```

184         processus_courant = 0;
185         break;
186     }
187 }
188 }
189 }
190 } while (pid_child > 0);
191 }
192
193 // Fonction pour interceper le signal SIGINT
194
195 void handler_sigint(int signal) {
196     if (processus_courant != 0) {
197         kill(processus_courant, SIGKILL);
198         suppr_job(processus_courant);
199         ctrl_c = true;
200     } else {
201         printf( "\nTapez 'exit' pour quitter.\n" );
202         printf("florent:");
203         char* s = malloc(200); //reserve 200 octets pour le chemin
204         printf("%s", getcwd(s, 200));
205         free(s); //libere la memoire
206         printf(">>>");
207         fflush(stdout);
208     }
209 }
210
211 // Fonction pour interceper le signal SIGTSTP
212 void handler_sigstop(int signal){
213     if (processus_courant != 0) {
214         kill(processus_courant, SIGSTOP);
215         stop_job(processus_courant);
216         processus_courant = 0;
217         //handler_sigint(signal);
218
219         ctrl_z = true;
220     } else {
221         printf( "\nTapez 'exit' pour quitter.\n" );
222         printf("florent:");
223         char* s = malloc(200); //reserve 200 octets pour le chemin
224         printf("%s", getcwd(s, 200));
225         free(s); //libere la memoire
226         printf(">>>");
227         fflush(stdout);
228     }
229 }
230 }
231
232
233
234 int main() {
235     //char commande[MAX_LIGNE + 1];
236     //char *args[64];
237     struct cmdline *commande;
238
239     while (1) {
240
241         //interceper SIGNAUX
242         //FILS
243         signal(SIGCHLD, handler_sigchild);
244         //SIGINT
245         signal(SIGINT, handler_sigint);
246         //SIGTSTP
247         signal(SIGTSTP, SIG_IGN);
248         //signal(SIGSTOP, SIG_IGN);
249
250         //Test pour la Q2: affichage du chemin courant
251         char* s = malloc(200); //reserve 200 octets pour le chemin
252         printf("florent:");
253         printf("%s", getcwd(s, 200));
254         free(s); //libere la memoire
255         printf(">>>");
256         fflush(stdout);
257
258         // Lire une commande depuis l'entr e standard
259         //fgets(commande, MAX_LIGNE, stdin);
260         commande=readcmd();
261         //commande[strlen(commande, "\n")] = '\0'; // Retirer le saut de ligne final
262
263
264         //int taille = strlen(commande); // Taille de la commande
265         //printf("Taille de la commande: %d", taille);
266
267         //Commande en tache de fond
268         // V rifier si le dernier caract re est '&'
269         int bg = 0;
270         if (commande->backgrounded != NULL) { //taille > 0 && (commande[taille-1] == '&')) {
271             bg = 1;
272         }
273
274         //char commande2[MAX_LIGNE + 1];
275         //strcpy(commande2, commande);

```

```

276 //strcpy(commande, args[num_args]);
277 //Gestion des arguments
278 /*
279 int num_args = 0;
280 char *token = strtok(commande, " ");
281 while (token != NULL) {
282     args[num_args++] = token;
283     token = strtok(NULL, " ");
284 }
285 args[num_args] = NULL;
286
287 // args[0] contient la commande
288 //printf("Commande: %s, arg1: %s, arg2: %s", commande, args[0], args[1]);
289 //pid_t pid = fork();
290 */
291
292 if (strcmp(commande->seq[0][0], "cd") == 0) {
293     if (! commande->seq[0][1]) {
294         chdir(getenv("HOME"));
295     } else {
296         chdir(commande->seq[0][1]);
297     }
298 }
299
300 // Question 4
301 // Vérifier si la commande est "exit"
302 else if (strcmp(commande->seq[0][0], "exit") == 0) {
303     break;
304 }
305
306 // Question 6
307 else if (strcmp(commande->seq[0][0], "lj") == 0) {
308     list_jobs();
309 }
310 else if (strcmp(commande->seq[0][0], "sj")==0) {
311     stop_job(atoi(commande->seq[0][1]));
312     // Si args[1] n'est pas le nom d'une tache, il ne se passe rien
313 }
314 else if (strcmp(commande->seq[0][0], "bg")==0) {
315     //reprenre tache suspendue en arriere plan
316     reprendre_job(atoi(commande->seq[0][1]));
317 }
318 else if (strcmp(commande->seq[0][0], "fg")==0) {
319     //reprenre tache suspendue ou en bg au premier plan
320     int a = atoi(commande->seq[0][1]); //convertir string en int
321     //printf("a = %d",a);
322     //system(jobs[a].commande); //executer la commande ???
323     for (int i = 0; i < num_jobs; i++) {
324         if (jobs[i].job_id == a) {
325             jobs[i].active = 1; // Marquer la tache comme active
326             kill(jobs[i].pid, SIGCONT); // Relancer le processus au premier plan
327             processus_courant = jobs[i].pid;
328             int status;
329             waitpid(jobs[i].pid, &status, 0); // Attendre la fin du processus -> fait qu'il s'exécute au premier plan
330             break;
331         }
332     }
333     suppr_job(a); // Supprimer la tache de la liste
334 }
335 else if (strcmp(commande->seq[0][0], "susp")==0) {
336     kill(processus_courant, SIGSTOP);
337 }
338 else if (commande->seq[1] == NULL) {
339     // Créer un processus fils pour exécuter la commande
340     pid_t pid = fork();
341     if (pid == -1) {
342         fprintf(stderr, "Erreur sur fork\n");
343         exit(EXIT_FAILURE);
344     } else if (pid == 0) {
345         // Code du processus fils
346         signal(SIGTSTP, handler_sigstop);
347         // Rediriger l'entrée et la sortie standard
348         if (commande->in != NULL) {
349             rediriger_entree(commande->in);
350         }
351         if (commande->out != NULL) {
352             rediriger_sortie(commande->out);
353         }
354
355         //signal(SIGTSTP, SIG_IGN);
356         // Exécuter la commande
357         //int result = system(commande2);
358         int result = execvp(commande->seq[0][0], commande->seq[0]);
359
360         if (result == -1) { //Si erreur dans fork
361             fprintf(stderr, "Erreur sur execvp\n");
362             exit(EXIT_FAILURE);
363         }
364
365         exit(EXIT_SUCCESS);
366     } else {
367         // Code du processus parent

```

```

368 // code du processus parent
369 // Attendre la fin du processus fils si la commande n'est pas en tâche de fond
370 if (bg == 0) {
371     int status;
372     processus_courant = pid;
373     waitpid(pid, &status, 0);
374     processus_courant = 0;
375 }
376 else {
377     add_job(pid, commande->seq[0][0]);
378     printf("Tache en arriere plan: %d\t%s\t%s\n", pid, "active", commande->seq[0][0]);
379 }
380
381 } else{
382     // Question 10 et 11 - Tubes
383     int nbr_pipes = 0; //le nombre de tubes
384
385     //compter le nombre de pipes
386     while(commande->seq[nbr_pipes+1] != NULL) {
387         nbr_pipes++;
388     }
389     //printf("nbr_pipes = %d\n", nbr_pipes);
390
391     int wstatus;
392     int pid;
393     int tuyau[nbr_pipes*2]; //tableau de tubes
394
395     for (int k = 0; k < nbr_pipes; k++) { //créer les tubes
396         if (pipe(tuyau + k*2) < 0) { //erreur si < 0
397             //printf("
398             perror("Problemeeeeeee pipeeeeeee 1"); //blablabla
399             exit(1);
400         }
401     }
402
403     int acc = 0;
404     int indice_com = 0; // l'indice de la commande
405
406     //créer les fils
407     while (commande->seq[indice_com] != NULL) {
408         pid = fork();
409         if (pid < 0) { //si erreur dans fork
410             perror("Problemeeeeeee fork pipeeeeeee 2"); //blablabla
411             exit(1);
412         } else if (pid == 0) { //processus fils
413             //redirection de l'entr e et de la sortie standards
414             if (indice_com == 0) {
415                 if (commande->in != NULL) {
416                     //printf("redir entree pipe");
417                     rediriger_entree(commande->in); //entr e
418                     //printf("in = %s\n", commande->in);
419                 }
420             }
421             if (indice_com == nbr_pipes) {
422                 if (commande->out != NULL) {
423                     //printf("redir sortie pipe");
424                     rediriger_sortie(commande->out); //sortie
425                     //printf("out = %s\n", commande->out);
426                 }
427             }
428
429             //premiere commande recoit stdin, les autres re oivent la sortie du tube pr c dent
430             //de meme pour la derniere commande qui recoit stdout en sortie
431             if (acc != 0) { //Pas la premiere commande
432                 //printf("acc = %d\n", acc);
433                 if (dup2(tuyau[acc-2], 0) < 0) {
434                     perror("Problemeeeeeee dup2 pipeeeeeee 3"); //blablabla
435                     exit(1);
436                 }
437             }
438
439             //pas la derniere commande
440             if (commande->seq[indice_com+1] != NULL) {
441                 if (dup2(tuyau[indice_com+1], 1) < 0) {
442                     perror("Problemeeeeeee dup2 pipeeeeeee 4"); //blablabla
443                     exit(1);
444                 }
445             }
446
447             //fermeture des tubes
448             for (int i = 0; i < 2*nbr_pipes; i++) {
449                 close(tuyau[i]);
450             }
451             //execution des commandes
452             if (execvp(commande->seq[indice_com][0], commande->seq[indice_com]) < 0) {
453                 perror("Problemeeeeeee exec pipeeeeeee 5"); //blablabla
454                 exit(1);
455             }
456         }
457         indice_com++; acc+=2;
458     }
459 }

```

```
459 //fermeture de la totalité des pipes
460
461 for (int m =0; m < 2*nbr_pipes; m++) {
462
463     close(tuyau[m]);
464 }
465 for (int l =0; l < nbr_pipes +1; l++) {
466     //printf("attente du fils %d\n", l);
467     wait(&wstatus);
468 }
469 }
470
471 return 0;
472 }
473
474 // thanks for reading guyyyys
```