



Systèmes d'Exploitations Centralisés

(mini)Projet (mini)Shell

Rapport

1ère Année, Département Sciences du Numérique

Younes Saoudi

2019-2020

Contents

1	Rapport miniShell	2
2	Code Source	6
2.1	Module myJobs	6
2.1.1	myJobs Header myJobs.h	6
2.1.2	myJobs Corps myJobs.c	8
2.2	Main	11

Rapport miniShell

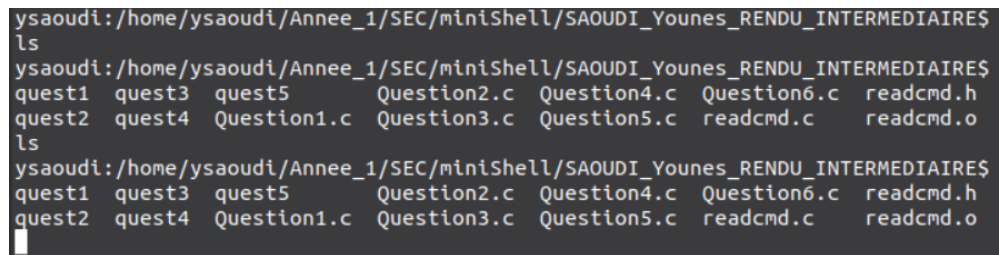
Question 01

Réalisation de la boucle de base de l'interpréteur, en se limitant à des commandes simples (pas d'opérateurs de composition), sans motifs pour les noms de fichiers.

L'implantation de cette question était assez simple: Il fallait d'abord une boucle infinie (`while(1)`) au sein de laquelle la création des processus était exécutée. En testant le retour des appels systèmes, et en ajoutant, dans la partie du fils, les commandes

```
execvp(cmd->seq[0][0], cmd->seq[0]); //Executer la commande
exit(EXIT_SUCCESS); /* Terminaison normale (0 = sans erreur) */
```

Question 02



(a) Question 2

Figure 1.1: Mise en exergue du fait que l'affichage de l'invite se mêle à l'exécution du processus fils.

L'ajout des commandes suivantes

```
printf("ysaoudi:");
printf("%s\n", getcwd(s,200));
```

avant le `fork()` engendre le problème remarqué dans la figure ci-dessus.

Question 03

Modification du code afin qu'il attende la fin de la dernière commande lancée avant de passer à la lecture de la ligne suivante.

Il suffisait d'ajouter `wait(NULL)` dans la partie exécutée par le père.

Question 04

Ajout de deux commandes internes, exécutées directement par l'interpréteur sans lancer de processus fils : `cd` et `exit`.

Pour ce faire, il fallait contrôler la saisie de ces commandes avant même la création du processus fils avec une simple structure `if.. else if...`

Question 05

Ajout de la possibilité du lancement de commandes en tâche de fond, spécifié par un `&` en fin de ligne. Pour achever cela, il fallait n'exécuter `wait(NULL)` que si l'attribut `backgrounded` de la structure `cmdline` était `NULL`, i.e., si la commande n'était pas en tâche de fond.

Question 06

Pour l'implantation de la commande `list`, j'ai dû créer un module `myJobs` d'une liste *dynamique* (et non une liste chaînée comme vu dans les cours de programmation). Ce module est en charge de l'ajout de processus, la mise à jour de leur état, leur suppression ainsi que leur affichage.

Tout d'abord, je définis 2 nouveaux types qui sont:

- `enum pStatus`
- `struct pList`

L'énumération `pStatus` contient les éléments `Done`, `Running` et `Stopped`, question d'imiter le vrai shell. La structure `pList` contient 4 attributs qui sont:

- La liste de PID `int* pIDs`
- La taille de la liste `int size` (bonne pratique, cf. *cours Langage C S6*)
- La liste des états de chaque processus `pStatus* pStatuses`
- La liste des commandes ayant engendré chaque processus `char** commands`

Ce module contient plusieurs procédures pour l'ajout, la suppression; etc. Pour imiter le comportement de la primitive `jobs` du shell, la procédure d'affichage dans le module `myJobs` est `void myJobs(pList processList)`.

Cette procédure affiche tous les processus. Si un processus a fini son exécution, il est signalé comme `Done` et son affichage avec la procédure `Jobs` le supprime de la liste de processus. C'est-à-dire, si je lance `sleep 5&` et `sleep 30&`, alors la saisi de `list` après 5 secondes donne:

```
[1]      5994      Done      sleep 5
[2]      5995    Running    sleep 30
```

Une deuxième exécution de `list` après que les 30 secondes soient écoulées affichera:

```
[2]      5995      Done      sleep 30
```

Les processus finis sont ainsi affichés une seule fois pour que l'utilisateur le sache puis supprimés de la liste de processus.

Pour l'exécution de `list`, `fg`, `bg` et `stop`, il fallait d'abord les ajouter dans la structure `if.. else if..` de la Question 4, s'assurer que chaque processus lancé an arrière plan est ajouté à liste de processus puis ajouter un handler du signal `SIGCHLD`, `suivi_fils` (Inspiré du tutoriel de Mr. Hamrouni sur les Signaux).

Ce dernier met à jour la l'état des processus dans la liste de commandes (`Done` si `WIFEXITED`, `Stopped` si `WIFSTOPPED` ou `WIFSIGNALED`; etc.)

Pour ce qui est de la commande `stop`, on envoie le signal `SIGSTOP` au processus.

Pour reprendre un signal arrêté dans l'arrière-plan avec la commande `bg`, on envoie le signal `SIGCONT` sans

attendre le fils.

Puisque l'attente du fils imite son exécution en avant-plan, c'est ce que j'ai fait pour implanter le processus `bg` avec la commande `wait(PID, 0, 0)`.

Puisque le signal `SIGTSTP` doit arrêter le processus en avant-plan sans quitter le minishell, j'ai fait ignorer le signal `SIGTSTP` aux fils avec `signal(SIGTSTP, SIG_IGN)` et j'ai ajouté un autre handler de signal `suivi_pere` pour `SIGTSTP` qui, à la réception de ce signal, envoie le signal `SIGSTOP` au processus. Sauf que puisque le père attend la fin d'un processus quelconque à cause de la commande `wait(NULL)` (Question 5) et que le processus est suspendu, nous ne revenons jamais au miniShell.

J'ai donc été obligé de créer un fils à la réception du signal `SIGTSTP` que je tue immédiatement. Ceci libère le père de la commande `wait(NULL)`, et l'envoi du signal `SIGSTOP` au processus en avant-plan le suspend puis nous fait revenir au miniShell.

Question 07

Comme pour le signal `SIGTSTP`, j'ai fait ignorer aux fils le signal `SIGINT` avec la commande `signal(SIGINT, SIG_IGN)` et j'y ai assigné le handler `suivi_pere` (où je teste bien sûr si le signal vient d'un `CTRL+Z` ou `CTRL+C`). Le processus fils reçoit alors le signal `SIGKILL` et est supprimé de la liste de processus.

Question 08

Pour cette question, il fallait utiliser les attributs `in` et `out` de la structure du module `readcmd`. L'implantation, qui a été faite dans la partie du fils (après le `fork`), était assez simple:

```
if in != NULL && err == NULL
    rediriger stdin vers le descripteur du fichier in
if out != NULL && err == NULL
    rediriger stdout vers le descripteur du fichier out
```

Il faut bien sûr aussi tester les retours des `dup2` et des `open` si jamais ces commandes engendrent une erreur. L'exécution de la commande est assuré par l'ancien `execvp`.

Question 09

Puisque, pour cette question, on suppose qu'il n'y a qu'un seul pipe, il fallait alors, après avoir testé l'existence de redirections, vérifier si `cmd->seq[1]` était `NULL`; i.e., vérifier s'il y avait un pipe dans la commande.

Le cas échéant, il fallait créer un petit fils qui se chargera d'exécuter la première partie de la commande en redirigeant son résultat vers un pipe hérité de son père, puis rediriger `stdin` vers l'entrée du pipe dans la partie du père et exécuter la seconde partie de la commande.

Question 10

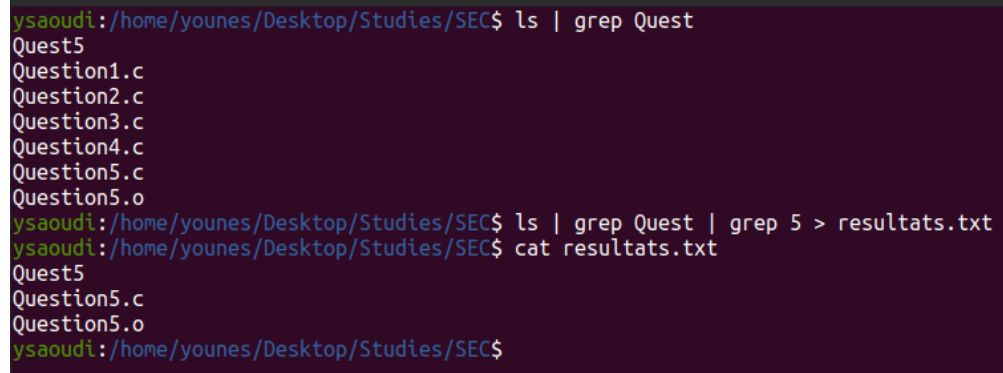
Dans cette partie, je vérifie d'abord s'il y a un seul pipe et compte le nombre de pipes le cas échéant. Je crée alors un tableau de pipes dont j'ouvre et ferme ceux dont j'ai besoin au sein d'une boucle `while`. Tant qu'on n'est toujours pas arrivé à la dernière commande, un nouveau petit fils est créé. Si on est à la première itération (première commande), on ne redirige pas `stdin`, sinon on le redirige vers le résultat de la commande précédente.

Si on est à la dernière commande, on ne redirige pas `stdout`, sinon on le redirige vers l'entrée de la commande suivante.

Après ces redirections en utilisant `dup2`, je ferme tous les pipes et exécute la commande dans la partie du petit fils. Toujours dans la boucle mais dans la partie du fils (père du petit fils), j'incréménte le compteur de commandes et celui des pipes.

Une fois toutes les commandes exécutées, je ferme tous les pipes et attend que tous les petits fils finissent

leur tâches avant de tuer leur père pour revenir au miniShell.

A terminal window with a dark purple background and light green text. The prompt is 'ysaoudi:/home/younes/Desktop/Studies/SEC\$'. The first command is 'ls | grep Quest', which lists 'Quest5', 'Question1.c', 'Question2.c', 'Question3.c', 'Question4.c', 'Question5.c', and 'Question5.o'. The second command is 'ls | grep Quest | grep 5 > resultats.txt'. The third command is 'cat resultats.txt', which outputs 'Quest5', 'Question5.c', and 'Question5.o'. The prompt returns to 'ysaoudi:/home/younes/Desktop/Studies/SEC\$'.

```
ysaoudi:/home/younes/Desktop/Studies/SEC$ ls | grep Quest
Quest5
Question1.c
Question2.c
Question3.c
Question4.c
Question5.c
Question5.o
ysaoudi:/home/younes/Desktop/Studies/SEC$ ls | grep Quest | grep 5 > resultats.txt
ysaoudi:/home/younes/Desktop/Studies/SEC$ cat resultats.txt
Quest5
Question5.c
Question5.o
ysaoudi:/home/younes/Desktop/Studies/SEC$
```

(a) Démonstration

Figure 1.2: Résultat de l'implantation des questions 8 à 10.

Comme on peut le remarquer, les tubes et redirections fonctionnent correctement. Ceci marque alors la fin du projet minishell.

Code Source

2.1 Module myJobs

2.1.1 myJobs Header myJobs.h

```
1 #include <stdbool.h>
2 #ifndef _MYJOBS_H
3 #define _MYJOBS_H
4
5
6     enum pStatus{Done, Running, Stopped};
7     /**
8      * Les différents status d'un processus quelconque
9      * */
10    typedef enum pStatus pStatus;
11
12
13    struct pList{
14        int* pIDs;
15        int size;
16        pStatus* pStatuses;
17        char** commands;
18    };
19    /**
20     * La table de processus et leur informations
21     * */
22    typedef struct pList pList;
23
24    /**
25     * Initialiser la liste de processus
26     * @param processList La future liste de processus
27     * */
28    void initializeList(pList* processList);
29
30    /**
31     * Tester l'existence d'un processus dans le liste
32     * @param processList La liste des processus
33     * @param pID Le processus dont on veut vérifier l'existence
34     */
35    bool pExists(pList processList, int pID);
36
37    /**
38     * Ajouter un processus à la liste de processus
39     * @param processList La liste de processus
40     * @param newPID L'ID du nouveau processus
41     * @param newCommand La commande qui a engendré ce processus
42     * */
43    void addCommand(pList* processList, int newPID, char* newcmd);
44
45    /**
```



```

46     * Supprimer un processus de la liste de processus
47     * @param processList La liste de processus
48     * @param deadID L'ID du nouveau processus
49     * */
50     void popCommand(pList* processList, int deadID);
51
52     /**
53     * Supprimer les processus terminés de la liste de processus
54     * @param processList La liste de processus
55     * */
56     void popDoneCommands(pList* processList);
57
58     /**
59     * Modifier l'état d'un processus dans la liste de processus
60     * @param processList
61     * */
62     void updateCommand(pList* processList, int processID, pStatus newStatus);
63
64     /**
65     * Mimiquer la commande jobs du shell (afficher les processus)
66     * */
67     void jobs(pList* processList);
68
69 #endif

```

2.1.2 myJobs Corps myJobs.c

```
1 #include <stdio.h> /* entrées sorties */
2 #include <stdlib.h> /* exit */
3 #include <string.h>
4 #include "myJobs.h"
5 /**
6  * @author Younes SAOUDI 1SN-D
7  * Tout ce qui est relatif au stockage des informations des processus et leur affichage.
8  */
9
10 void initializeList(pList* processList){
11     processList->size = 0;
12     processList->pIDs = malloc(sizeof(int));
13     processList->pStatuses = malloc(sizeof(pStatus));
14     char ** array=(char **)malloc(sizeof(*array) );
15     processList->commands = malloc(sizeof(*array));
16 }
17
18 bool pExists(pList processList, int pID){
19     int index = 1;
20     while (index <= processList.size){
21         if(processList.pIDs[index] == pID){
22             return true;
23         }
24         index ++;
25     }
26     return false;
27 }
28
29 void addCommand(pList* processList, int newpID, char* newcmd){
30     int* newList = (int *) realloc(processList->pIDs, (processList->size + 2) * sizeof(int)
31 );
32
33     pStatus* newStatuses = realloc(processList->pStatuses, (processList->size + 2) * sizeof(
34 pStatus));
35
36     char** array = processList->commands;
37     char** newCommands = (char **) realloc(processList->commands, (processList->size + 2) *
38 sizeof(*array));
39
40     if (newList && newStatuses && newCommands){
41         processList->pIDs = newList;
42         processList->size ++;
43         processList->pIDs[processList->size] = newpID;
44
45         processList->pStatuses = newStatuses;
46         processList->pStatuses[processList->size] = Running;
47
48         processList->commands = newCommands;
49         processList->commands[processList->size] = newcmd;
50     }
51 }
52
53 void updateCommand(pList* processList, int processID, pStatus newStatus){
54     int index = 0;
55     while (processList->pIDs[index] != processID){
56         index ++;
57     }
58     processList->pStatuses[index] = newStatus;
59 }
60
61 void popCommand(pList *processList, int deadID){
62     if(processList->size == 1){
```

```

60     initializeList(processList);
61 } else{
62     if (processList -> pIDs[processList->size] != deadID){
63
64         int index = 1;
65         while (index < processList->size && processList->pIDs[index] != deadID){
66             index ++;
67         }
68
69         while(index < processList->size){
70             processList->pIDs[index] = processList->pIDs[index + 1];
71             processList->commands[index] = processList->commands[index + 1];
72             processList->pStatuses[index] = processList->pStatuses[index + 1];
73             index ++;
74         }
75     }
76
77     int* newList = (int *) realloc(processList->pIDs, (processList->size - 1) * sizeof(
78 int));
79
80     pStatus* newStatuses = realloc(processList->pStatuses, (processList->size - 1) *
81 sizeof(pStatus));
82
83     char** array = processList->commands;
84     char** newCommands = (char **) realloc(processList->commands, (processList->size) *
85 sizeof(*array));
86
87     if(newList && newStatuses && newCommands){
88         processList->pIDs = newList;
89         processList->size --;
90
91         processList->pStatuses = newStatuses;
92
93         processList->commands = newCommands;
94     }
95 }
96
97 void popDoneCommands(pList* processList){
98     int index = 1;
99     while (index <= processList->size){
100         pStatus statusEnum = processList->pStatuses[index];
101         char *status;
102         switch (statusEnum){
103             case Done: status = "Done";break;
104             case Running: status = "Running"; break;
105             case Stopped: status = "Stopped"; break;
106         }
107         if (strcmp(status, "Done") == 0){
108             popCommand(processList, processList->pIDs[index]);
109         }
110         index ++;
111     }
112 }
113
114 void jobs(pList* processList){
115     int size = processList->size;
116
117     for (int i = 1; i <= size; i++){
118
119         int processID = processList->pIDs[i];
120         pStatus statusEnum = processList->pStatuses[i];
121         char *command = processList->commands[i];
122         //printf("%s\n", command);

```

```

122     //printf("OK COMMAND\n");
123     char *status;
124
125     switch (statusEnum){
126         case Done: status = "Done";break;
127         case Running: status = "Running"; break;
128         case Stopped: status = "Stopped"; break;
129     }
130     printf("[%d]\t %d\t %s\t ", i, processID, status);
131     if (strcmp(status, "Done") == 0){
132         printf("\t ");
133     }
134     printf("%s\n", command);
135 }
136 popDoneCommands(processList);
137 }

```

2.2 Main

```
1 #include <stdio.h> /* entrées sorties */
2 #include <unistd.h> /* primitives de base : fork, ...*/
3 #include <stdlib.h> /* exit */
4 #include <signal.h> /* traitement des signaux */
5 #include "readcmd.h"
6 #include <sys/wait.h> /* wait */
7 #include <string.h> /* opérations sur les chaînes */
8 #include <errno.h>
9 #include <fcntl.h> /* opérations sur les fichiers */
10 #include "myJobs.h" //Contient tout ce qui est relatif au stockage des processus, leur
    informations ainsi que leur affichage
11
12 /**
13  * @author Younes SAOUDI 1SN-D
14  * miniShell
15  */
16
17 pList processList; //La liste des eventuels processus
18 bool executionFG; //Verifier si l'exécution se fait en FG
19 int pidCHILDFG; //le PID du fils executant en FG
20 char *commandFG; //Commande exécutée en FG
21
22 void suivi_pere(int sig)
23 {
24     //Si l'exécution se fait en FG et que CTRL Z est tapé
25     if (executionFG && sig == SIGTSTP)
26     {
27         //executionFG = false;
28         printf(" Suspension du processus\n");
29         int fils = fork(); //Le père attend qu'un fils quelconque finisse, on va donc en
    créer un pour le terminer immédiatement (sinon le père attendre pidCHILDFG alors qu'il
    sera suspendu)
30         if (fils == 0)
31         { //Nouveau Fils
32             exit(EXIT_SUCCESS);
33         }
34         else if (fils > 0)
35         { //Père
36             kill(fils, SIGKILL);
37             kill(pidCHILDFG, SIGSTOP); //Suspension du processus
38         }
39         else
40         {
41             /* échec du fork */
42             printf("Erreur fork\n");
43             /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
44             exit(1);
45         }
46     }
47     //Si l'exécution se fait en FG et que CTRL C est tapé
48     else if (executionFG && sig == SIGINT)
49     {
50         printf(" Terminaison du processus\n");
51         executionFG = false;
52         kill(pidCHILDFG, SIGKILL);
53         if (pExists(processList, pidCHILDFG))
54         {
55             popCommand(&processList, pidCHILDFG); //Suppression de processus
56         }
57     }
58     //Si l'exécution ne se fait pas en FG et que CTRL C est tapé
59     else if (!executionFG && sig == SIGINT)
```

```

60     {
61         NULL; //Ne rien faire, puisque le miniShell est libre!
62     }
63 }
64
65 void suivi_fils(int sig)
66 {
67     int etat_fils, pid_fils;
68
69     do
70     {
71
72         pid_fils = (int)waitpid(-1, &etat_fils, WNOHANG | WUNTRACED | WCONTINUED);
73
74         if ((pid_fils == -1) && (errno != ECHILD))
75         {
76
77             perror("waitpid");
78
79             exit(EXIT_FAILURE);
80         }
81         else if (pid_fils > 0)
82         {
83
84             if (WIFCONTINUED(etat_fils))
85             {
86                 /* traiter la reprise */
87                 if (pExists(processList, pid_fils))
88                 {
89                     updateCommand(&processList, pid_fils, Running);
90                 }
91             }
92             else if (WIFSTOPPED(etat_fils) || WIFSIGNALED(etat_fils))
93             {
94                 /* trailer l'arrêt */
95                 if (pExists(processList, pid_fils))
96                 {
97                     updateCommand(&processList, pid_fils, Stopped);
98                 }
99             }
100             else if (WIFEXITED(etat_fils))
101             {
102                 /* traiter l'arrêt normal */
103                 if (pExists(processList, pid_fils))
104                 {
105                     updateCommand(&processList, pid_fils, Done);
106                 }
107             }
108         }
109     }
110
111     } while (pid_fils > 0);
112
113     /* autres actions après le suivi des changements d'état */
114 }
115 int main()
116 {
117     char s[200]; // répertoire courant
118     int retour;
119     struct cmdline *cmd; //ligne de commande
120
121     signal(SIGTSTP, suivi_pere); //handler CTRL Z
122     signal(SIGINT, suivi_pere); // handle CTRL C
123
124     initializeList(&processList); //initialisation de la liste de processus

```

```

125 signal(SIGCHLD, suivi_fils); //handler SIGCHLD
126 int desc_ent, desc_res, dupdesc; /* descripteurs de fichiers */
127
128
129 while (1)
130 {
131     //MODIFICATION QUESTION 2
132     printf("\033[0;32m"); //VERT
133     printf("ysaoudi");
134     printf("\033[0m"); //DEFAULT
135     printf(":");
136     printf("\033[0;34m"); //BLEU
137     printf("%s", getcwd(s, 200)); //Repertoire de travail
138     printf("\033[0m"); //DEFAULT
139     printf("$ ");
140     //FIN QUESTION 2
141     cmd = readcmd();
142
143     //MODIFICATION QUESTION 4 ET 6
144     if (strcmp(cmd->seq[0][0], "cd") == 0)
145     {
146         chdir(cmd->seq[0][1]); //changer de repertoire
147         continue;
148     }
149     else if (strcmp(cmd->seq[0][0], "exit") == 0)
150     {
151         exit(0);
152     }
153     else if (strcmp(cmd->seq[0][0], "list") == 0)
154     {
155         jobs(&processList); //afficher la liste de processus
156         continue;
157     }
158     else if (strcmp(cmd->seq[0][0], "stop") == 0)
159     {
160         if (cmd->seq[0][1] == NULL)
161         {
162             printf("Il faut préciser le numéro du processus!\n");
163             continue;
164         }
165         else
166         {
167             char *strPID = malloc(sizeof(cmd->seq[0][1]));
168             strcpy(strPID, cmd->seq[0][1]);
169             int PID;
170             sscanf(strPID, "%d", &PID); //récuper le PID integer
171
172             if (pExists(processList, PID))
173             {
174                 //stop le processus
175                 kill(PID, SIGSTOP);
176                 continue;
177             }
178             else
179             {
180                 printf("Ce processus n'existe pas!\n");
181                 continue;
182             }
183         }
184     }
185     else if (strcmp(cmd->seq[0][0], "bg") == 0)
186     {
187         if (cmd->seq[0][1] == NULL)
188         {
189             printf("Il faut préciser le numéro du processus!\n");

```

```

190         continue;
191     }
192     else
193     {
194         char *strPID = malloc(sizeof(cmd->seq[0][1])); //pid String
195         strcpy(strPID, cmd->seq[0][1]);
196         int PID;
197         sscanf(strPID, "%d", &PID); //conversion du pid String en pid int
198
199         if (pExists(processList, PID))
200         {
201
202             //reprendre le processus mais en arrière plan
203             kill(PID, SIGCONT);
204             continue;
205         }
206         else
207         {
208             printf("Ce processus n'existe pas!\n");
209             continue;
210         }
211     }
212 }
213 else if (strcmp(cmd->seq[0][0], "fg") == 0)
214 {
215     if (cmd->seq[0][1] == NULL)
216     {
217         printf("Il faut préciser le numéro du processus!\n");
218         continue;
219     }
220     else
221     {
222         char *strPID = malloc(sizeof(cmd->seq[0][1]));
223         strcpy(strPID, cmd->seq[0][1]);
224         int PID;
225         sscanf(strPID, "%d", &PID);
226
227         if (pExists(processList, PID))
228         {
229
230             //reprendre le processus mais en avant plan
231             //waitpid(PID, &etat_fils, WNOHANG | WUNTRACED | WCONTINUED);
232             //if ( WIFSTOPPED(etat_fils) || WIFSIGNALED(etat_fils)) {
233             kill(PID, SIGCONT);
234             waitpid(PID, 0, 0);
235             updateCommand(&processList, PID, Done);
236             //}
237             continue;
238         }
239         else
240         {
241             printf("Ce processus n'existe pas!\n");
242             continue;
243         }
244     }
245 }
246
247 //FIN QUESTION 4 ET 6
248
249 retour = fork();
250
251 /* Bonne pratique : tester systématiquement le retour des appels système */
252 if (retour < 0)
253 { /* échec du fork */
254     printf("Erreur fork\n");

```



```

255     /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
256     exit(1);
257 }
258
259 /* fils */
260 if (retour == 0)
261 {
262     signal(SIGTSTP, SIG_IGN);    //Ignorer CTRL Z
263     signal(SIGINT, SIG_IGN);    //Ignorer CTRL C
264     signal(SIGCHLD, suivi_fils); //Attribuer la handler suivi_fils au signal SIGCHLD
265
266     //S'il y a redirection de stdin ou stdout
267     if (cmd->in != NULL || cmd->out != NULL)
268     {
269         if (cmd->in != NULL && cmd->err == NULL) //S'il y a redirection de stdin
vers un fichier
270     {
271         /*ouvrir le fichier des entrées */
272         desc_ent = open(cmd->in, O_RDONLY);
273         /* traiter systématiquement les retours d'erreur des appels */
274         if (desc_ent < 0)
275         {
276             fprintf(stderr, "Erreur ouverture %s\n", cmd->in);
277             exit(1);
278         }
279         //rediriger stdin vers le fichier de desc_ent
280         dupdesc = dup2(desc_ent, 0);
281         if (dupdesc == -1)
282         {
283             /* échec du dup2 */
284             perror("Erreur dup2\n");
285             exit(5);
286         }
287     }
288     if (cmd->out != NULL && cmd->err == NULL) //S'il y a redirection de stdout
vers un fichier
289     {
290
291         /* ouvrir le fichier résultats */
292         desc_res = open(cmd->out, O_WRONLY | O_CREAT | O_TRUNC, 0640);
293         /* traiter systématiquement les retours d'erreur des appels */
294         if (desc_res < 0)
295         {
296             fprintf(stderr, "Erreur ouverture %s\n", cmd->out);
297             exit(2);
298         }
299         //rediriger stdout vers le fichier de desc_res
300         dupdesc = dup2(desc_res, 1);
301         if (dupdesc == -1)
302         {
303             /* échec du dup2 */
304             perror("Erreur dup2\n");
305             exit(5);
306         }
307     }
308 }
309
310 //S'il y a au moins un pipe
311 if (cmd->seq[1] != NULL)
312 {
313
314     int nbr_pipes = 0; //nombre de pipes
315
316     while (cmd->seq[nbr_pipes + 1] != NULL)
317     {

```

```

318         nbr_pipes++;
319     }
320
321     int pipes[nbr_pipes*2]; //pipes
322
323     for (int j = 0; j < nbr_pipes; j++)
324     {
325         int retour_pipe = pipe(pipes + j*2); //création de pipes
326         if (retour_pipe == -1)
327         {
328             perror("Erreur pipe\n");
329             exit(1);
330         }
331     }
332     int i = 0;
333     int cmd_nbr = 0; //numéro de la commande à exécuter
334
335     while(cmd_nbr < nbr_pipes +1)
336     {
337         int fils = fork();
338         if (fils == -1)
339         {
340             //erreur fork
341             perror("Erreur fork\n");
342             exit(1);
343         }
344         else if (fils == 0)
345         {
346             //fils
347             if (cmd_nbr > 0) //si ce n'est pas la première commande
348             {
349                 //rediriger stdin vers le résultat de la commande précédente
350                 dupdesc = dup2(pipes[i - 2], 0);
351                 if (dupdesc == -1)
352                 {
353                     perror("Erreur dup2 Lecture");
354                     fprintf(stderr, "iteration %d\n", cmd_nbr);
355                     exit(1);
356                 }
357             }
358
359             if (cmd_nbr != nbr_pipes) //si ce n'est pas la dernière commande
360             {
361                 //rediriger stdout vers l'entrée de la commande suivante
362                 dupdesc = dup2(pipes[i +1], 1);
363                 if (dupdesc == -1)
364                 {
365                     perror("Erreur dup2 Ecriture");
366                     fprintf(stderr, "iteration %d\n", cmd_nbr);
367                     exit(1);
368                 }
369             }
370
371             //fermer les pipes inutiles
372             for (int k = 0; k < 2*nbr_pipes; k++)
373             {
374                 close(pipes[k]);
375             }
376
377             //execution de la commande actuelle
378             if (execvp(cmd->seq[cmd_nbr][0], cmd->seq[cmd_nbr]) < 0)
379             {
380                 printf("%s: Unknown Command!\n", cmd->seq[cmd_nbr][0]);
381                 exit(EXIT_FAILURE);
382             }

```

```

383         else
384         {
385             exit(EXIT_SUCCESS);
386         }
387     }
388     else
389     {
390         //père
391         cmd_nbr++; //pour exécuter la commande suivante
392         i+=2; //pour lire et écrire dans les bons pipes
393     }
394 }
395 //fermer les pipes puisque la commande a été exécutée correctement
396 for (int k = 0; k < 2*nbr_pipes; k++)
397 {
398     close(pipes[k]);
399 }
400
401 //attendre que tous les fils finissent
402 int status;
403 for(i = 0; i < nbr_pipes + 1; i++)
404 {
405     wait(&status);
406 }
407 exit(EXIT_SUCCESS); //revenir au minishell!
408 }
409 //Execution de la commande (n'arrive jamais ici dans le cas d'un pipe)
410 if (execvp(cmd->seq[0][0], cmd->seq[0]) < 0)
411 {
412     printf("%s: Unknown Command!\n", cmd->seq[0][0]);
413     exit(EXIT_FAILURE);
414 }
415 else
416 {
417     exit(EXIT_SUCCESS); /* Terminaison normale (0 = sans erreur) */
418 }
419 //execvp(cmd->seq[0][0], cmd->seq[0]); //Executer la commande
420 //exit(EXIT_SUCCESS); /* Terminaison normale (0 = sans erreur)
421 */
422 }
423
424 /* pere */
425 else
426 {
427     if (cmd->backgrounded == NULL)
428     {
429         executionFG = true;
430         pidCHILDFG = retour;
431         char *commandFG = malloc(sizeof(cmd->seq[0][0]));
432         strcpy(commandFG, cmd->seq[0][0]);
433         addCommand(&processList, retour, commandFG);
434         updateCommand(&processList, retour, Done);
435         wait(NULL);
436     }
437     else
438     {
439         executionFG = false;
440         char *newCommand = malloc(sizeof(cmd->seq[0][0]));
441         strcpy(newCommand, cmd->seq[0][0]);
442         addCommand(&processList, retour, newCommand); //Ajout de la commande à liste
443         de processus
444     }
445 }

```

```
446     return EXIT_SUCCESS;
447 }
```