



Rapport de Projet

Minishell

Sommaire

1	Introduction	2
2	TP1 : Processus et exécution séquentielle	2
2.1	Étapes à réaliser	2
2.2	Implémentation et code	3
2.3	Tests et résultats	4
3	TP2 : Signaux et gestion de processus	6
3.1	Étapes à réaliser	6
3.2	Implémentation et code	6
3.3	Tests et résultats	8
4	TP3 : Signaux avancés	9
4.1	Étapes à réaliser	9
4.2	Implémentation et code	10
4.3	Tests et résultats	11
5	TP4 : Redirections et manipulation de fichiers	13
5.1	Étapes à réaliser	13
5.2	Implémentation et code	13
5.3	Tests et résultats	14
6	TP5 : Tubes et pipelines	16
6.1	Étapes à réaliser	16
6.2	Implémentation et code	16
6.3	Tests et résultats	17
7	Conclusion	19

Réalisé par :
Ralph Khairallah

Date : June 4, 2025

1 Introduction

Pendant ce semestre j'ai travaillé sur la construction d'un **Minishell**. L'idée est simple : recréer, en très léger, ce qu'un shell Unix fait tous les jours : lire une ligne, la comprendre, lancer les programmes qu'elle demande, puis revenir attendre la suivante.

Pourquoi faire un shell ? Parce qu'il oblige à toucher aux appels système de base : **fork** pour dupliquer un processus, **exec** pour changer de programme, **wait** pour récupérer le code retour, **pipe** et **dup2** pour raccorder les entrées-sorties, sans oublier la gestion des signaux (**SIGINT**, **SIGTSTP**, ...).

Comment ça marche ? Le Minishell :

- a) affiche un prompt et lit la ligne tapée ;
- b) découpe cette ligne pour séparer la commande, ses arguments, les redirections (<, >, ») et les tubes (|) ;
- c) crée un ou plusieurs fils avec **fork()**, remplace leur code par la bonne commande grâce à **execvp()**, et connecte les entrées/sorties avec **dup2()** quand il faut ;
- d) intercepte les signaux clavier (Ctrl-C, Ctrl-Z) pour arrêter ou suspendre proprement ;
- e) attend la fin des fils (**waitpid()**) avant de ré-afficher le prompt.

Découpage du projet Le travail a été réparti sur cinq TPs :

- **TP1** : exécuter une commande simple ;
- **TP2** : intercepter les signaux de base ;
- **TP3** : gérer ces signaux quand plusieurs processus tournent en parallèle ;
- **TP4** : ajouter les redirections d'entrée/sortie ;
- **TP5** : relier plusieurs commandes avec les tubes.

Contenu du rapport Pour chaque TP, on trouvera :

- un rappel de ce qui était demandé ;
- L'implantation en code ;
- les tests effectués et les résultats obtenus.

2 TP1 : Processus et exécution séquentielle

2.1 Étapes à réaliser

1. Étape 1 — Tester le programme

Compiler le minishell d'origine avec **make**, le lancer (**./minishell**) et vérifier qu'il se contente d'afficher la ligne tapée puis le prompt. Aucun changement de code à cette étape.

2. Étape 2 — Lancement d'une commande

Ajouter un `fork()` puis, dans le fils, un `execvp()` pour remplacer l'image du processus par la commande demandée. Le père ne fait rien d'autre et le prompt réapparaît immédiatement.

3. Étape 3 — Enchaînement séquentiel

Insérer, côté père, un `waitpid()` pour attendre la fin du fils avant de lire la ligne suivante. On obtient donc la séquence : création du fils, *exec*, attente, puis retour au prompt.

4. Étape 4 — Tâche de fond

Détecter la présence d'un `&` grâce au champ `commande->backgrounded`. Si ce champ est différent de `NULL`, le père *n'attend pas* ; il affiche simplement un message avec le `pid` du fils.

2.2 Implémentation et code

```

1  while (!fini) {
2      printf("> ");
3      struct cmdline *commande = readcmd();
4
5      int indexseq = 0;
6      char **cmd;
7      while ((cmd = commande->seq[indexseq])) {
8          if (cmd[0] && strcmp(cmd[0], "exit") == 0) {
9              fini = true;
10             printf("Au revoir ...\n");
11         } else if (cmd[0]) {
12             pid_t pid_fork = fork();
13             if (pid_fork == -1) {
14                 perror("fork");
15                 exit(EXIT_FAILURE);
16             } else if (pid_fork == 0) {          // FILS
17                 execvp(cmd[0], cmd);           // Étape 2
18                 perror(cmd[0]);                // si exec échoue
19                 exit(EXIT_FAILURE);
20             } else {                            // PÈRE
21                 if (commande->backgrounded == NULL) { // Étape 3
22                     int status;
23                     if (waitpid(pid_fork, &status, 0) != -1) {
24                         if (WIFEXITED(status))
25                             printf("Fin fils %d (code %d)\n",
26                                   pid_fork, WEXITSTATUS(status));
27                         else if (WIFSIGNALED(status))
28                             printf("Fils %d tué par signal %d\n",
29                                   pid_fork, WTERMSIG(status));
30                     }
31                 } else {                        // Étape 4
32                     printf("Lancement en tâche de fond (pid %d)\n",
33                             pid_fork);
34                 }
35             }
36         }
37     }
38 }

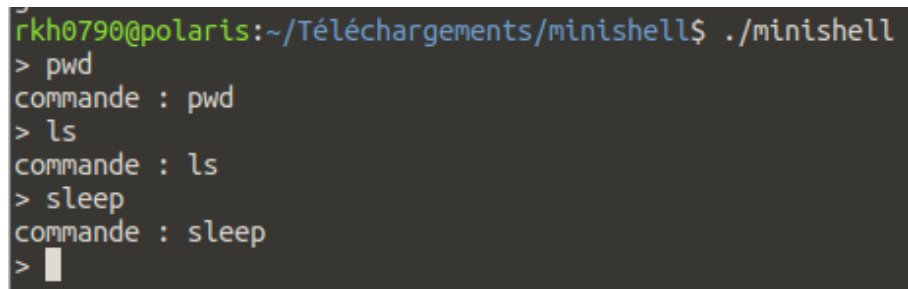
```

```
35         }  
36     }  
37     indexseq++;  
38 }  
39 }
```

2.3 Tests et résultats

1. Étape 1 : lancement basique

On tape une commande quelconque on vérifie que le shell ré-affiche simplement la ligne.

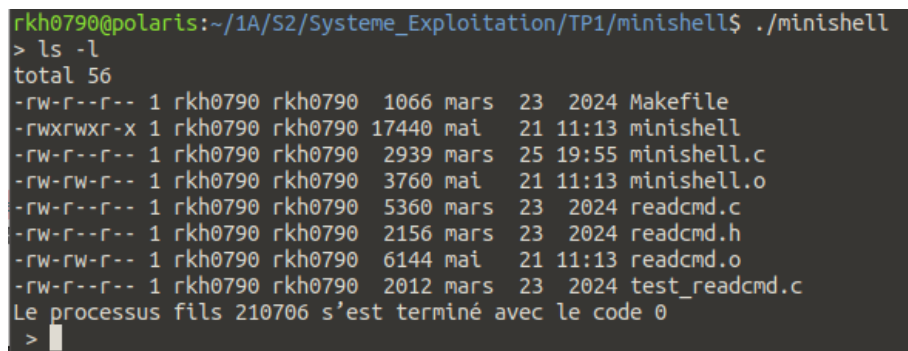


```
rkh0790@polaris:~/Téléchargements/minishell$ ./minishell  
> pwd  
commande : pwd  
> ls  
commande : ls  
> sleep  
commande : sleep  
> █
```

Figure 1: Comportement initial (étape 1)

2. Étape 2 : exécution d'une commande

`ls -l` doit s'exécuter dans le fils, le prompt revient immédiatement.



```
rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP1/minishell$ ./minishell  
> ls -l  
total 56  
-rw-r--r-- 1 rkh0790 rkh0790 1066 mars 23 2024 Makefile  
-rwxrwxr-x 1 rkh0790 rkh0790 17440 mai 21 11:13 minishell  
-rw-r--r-- 1 rkh0790 rkh0790 2939 mars 25 19:55 minishell.c  
-rw-rw-r-- 1 rkh0790 rkh0790 3760 mai 21 11:13 minishell.o  
-rw-r--r-- 1 rkh0790 rkh0790 5360 mars 23 2024 readcmd.c  
-rw-r--r-- 1 rkh0790 rkh0790 2156 mars 23 2024 readcmd.h  
-rw-rw-r-- 1 rkh0790 rkh0790 6144 mai 21 11:13 readcmd.o  
-rw-r--r-- 1 rkh0790 rkh0790 2012 mars 23 2024 test_readcmd.c  
Le processus fils 210706 s'est terminé avec le code 0  
> █
```

Figure 2: `ls -l` lancé via `fork/exec`

3. Étape 3 : séquence bloquante

On lance `echo OK` suivi de `date` : la date ne s'affiche qu'après « OK ».

```
rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP1/minishell$ ./minishell
> echo OK
OK
Le processus fils 213316 s'est terminé avec le code 0
> echo OK DATE
OK DATE
Le processus fils 213347 s'est terminé avec le code 0
> █
```

Figure 3: Attente avec `waitpid`

4. Étape 4 : arrière-plan

`sleep 5 &` doit rendre la main aussitôt et afficher le message « Lancement en tâche de fond ».

```
rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP1/minishell$ ./minishell
> sleep 5 &
Lancement de commande en tâche de fond> ^C
rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP1/minishell$ ./minishell
> sleep 10 &
Lancement de commande en tâche de fond> sleep 50
Le processus fils 214044 s'est terminé avec le code 0
> █
```

Figure 4: Commande en tâche de fond

Bilan du TP1

Ce premier TP m'a surtout appris à lancer un programme depuis mon propre shell. J'ai vu comment :

- lire la ligne tapée par l'utilisateur ;
- copier le processus avec `fork()` ;
- remplacer le code du fils avec `execvp()` ;
- attendre la fin du fils avec `waitpid()` ou, si on met un `&`, le laisser tourner en tâche de fond.

Le plus piégeux a été de bien séparer le « père » et le « fils » après le `fork()` et de tester la valeur de retour de chaque appel système.

À la fin, le Minishell exécute déjà des commandes simples, affiche le code retour, et sait lancer un programme en arrière-plan. C'est une bonne base pour la suite : signaux, redirections, puis tubes.

3 TP2 : Signaux et gestion de processus

3.1 Étapes à réaliser

1. **Étape 5 — Traitement de SIGCHLD**

Installer un handler avec `sigaction()` qui affiche qu'un fils vient de changer d'état.

2. **Étape 6 — Récupération du statut des fils**

Dans le handler, boucler sur `waitpid(-1, status, WNOHANG | WUNTRACED | WCONTINUED)` afin de gérer *tous* les fils terminés, stoppés ou relancés.

3. **Étape 7 — Attente via pause()**

Pour une commande au premier plan, remplacer le `waitpid()` bloquant par un simple `pause()` : le père dort jusqu'à réception de SIGCHLD.

4. **Étape 8 — Stop/Continue sur un fils en arrière-plan**

Tester l'envoi de SIGSTOP puis SIGCONT (avec `kill`) à un processus lancé avec `&`.

5. **Étape 9 — Messages détaillés**

Dans le handler : distinguer fin normale, fin par signal, suspension et reprise grâce aux macros `WIFEXITED`, `WIFSIGNALED`, `WIFSTOPPED`, `WIFCONTINUED`.

3.2 Implémentation et code

```
1 //Gestion du signal SIGCHLD
2 void traitement(int sig)
3 {
4     int status;
5     pid_t pid_fork;
6     // Récupère tous les fils ayant changé d'état
7     while ((pid_fork = waitpid(-1, &status,
8         WNOHANG | WUNTRACED | WCONTINUED)) > 0) {
9         if (WIFEXITED(status))
10             printf("Le processus fils %d s'est terminé avec le code %d\n",
11                 pid_fork, WEXITSTATUS(status));
12         else if (WIFSIGNALED(status))
13             printf("Le processus fils %d s'est terminé par le signal %d\n",
14                 pid_fork, WTERMSIG(status));
15         else if (WIFSTOPPED(status))
16             printf("Le processus fils %d est suspendu (signal %d)\n",
17                 pid_fork, WSTOPSIG(status));
18         else if (WIFCONTINUED(status))
19             printf("Le processus fils %d reprend\n", pid_fork);
20     }
21 }
22
23 int main(void)
24 {
25     // Etape 5 , handler SIGCHLD (Étape 5)
26     struct sigaction action;
27     action.sa_handler = traitement;
```

```

28     sigemptyset(&action.sa_mask);
29     action.sa_flags = SA_RESTART;
30     sigaction(SIGCHLD, &action, NULL);
31
32     bool fini = false;
33     while (!fini) {
34         printf("> ");
35         struct cmdline *commande = readcmd();
36
37         int indexseq = 0;
38         char **cmd;
39         while ((cmd = commande->seq[indexseq])) {
40
41             // Quitter si l'utilisateur tape 'exit'
42             if (cmd[0] && strcmp(cmd[0], "exit") == 0) {
43                 fini = true;
44                 printf("Au revoir ...\n");
45             }
46             // Sinon on lance la commande
47             else if (cmd[0]) {
48                 pid_t pid_fork = fork();
49                 if (pid_fork == -1) {
50                     perror("fork");
51                     exit(EXIT_FAILURE);
52                 } else if (pid_fork == 0) {           // FILS
53                     execvp(cmd[0], cmd);
54                     perror(cmd[0]);                 // exec a échoué
55                     exit(EXIT_FAILURE);
56                 } else {                             // PÈRE
57                     if (commande->backgrounded == NULL) {
58                         pause();                     // Étape 7 : on attend
59                                                         un SIGCHLD
60                     } else {
61                         printf("Lancement de commande en tache de fond\n");
62                     }
63                 }
64             }
65             indexseq++;
66         }
67     }
68     return EXIT_SUCCESS;
69 }

```

3.3 Tests et résultats

1. Étape 5 : message à la fin d'un fils

On lance des `sleep` en avant et arrière plans ; un message doit apparaître quand le `sleep` se termine.

```
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 10 &
Lancement de commande en tache de fond> Le processus fils 219366 s'est terminé a
vec le code 0
sleep 4
Le processus fils 219472 s'est terminé avec le code 0
>
```

Figure 5: Signal SIGCHLD capturé

2. Étape 6 : plusieurs fils

On lance trois `sleep 5 &` d'affilée ; les trois fins doivent être détectées sans zombie.

```
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 5&
Lancement de commande en tache de fond> sleep 5&
Lancement de commande en tache de fond> sleep 5&
Lancement de commande en tache de fond> Le processus fils 222407 s'est terminé avec le code 0
Le processus fils 222417 s'est terminé avec le code 0
Le processus fils 222429 s'est terminé avec le code 0
>
```

Figure 6: Boucle sur `waitpid` non bloquant

3. Étape 7 : pause sur le 1^{er} plan

On lance `sleep 10&` ; le message de lancement en tache de fond s'affiche immédiatement

On lance `sleep 5` (sans `&`) ; le shell se bloque puis revient au prompt après 5 s.

```
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 10&
Lancement de commande en tache de fond> ^C
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 5
Le processus fils 224890 s'est terminé avec le code 0
>
```

Figure 7: `pause()` débloquée par SIGCHLD

4. Étape 8 : stop / continue

On lance `sleep 50 &` ; le shell indique aussitôt [BG] pid 232514. `kill -STOP 232514` suspend le processus : le handler affiche « Le processus fils 232514 est suspendu (signal 19) ». `kill -CONT 232514` le relance : le handler affiche « Le processus fils 232514 reprend ».


```
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 50&
Lancement de commande en tache de fond (pid 232514)
> kill -STOP 232514
Le processus fils 232514 est suspendu (signal 19)
> Le processus fils 232589 s'est terminé avec le code 0
kill -CONT 232514
Le processus fils 232514 reprend
> Le processus fils 232644 s'est terminé avec le code 0
Le processus fils 232514 s'est terminé avec le code 0
```

Figure 8: Suspension puis reprise du processus 232514

5. Étape 9 : messages détaillés

La capture montre bien que les deux messages du handler (*suspendu* puis *reprend*) apparaissent, confirmant la bonne gestion de SIGSTOP et SIGCONT.

```
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 50&
Lancement de commande en tache de fond (pid 232514)
> kill -STOP 232514
Le processus fils 232514 est suspendu (signal 19)
> Le processus fils 232589 s'est terminé avec le code 0
kill -CONT 232514
Le processus fils 232514 reprend
> Le processus fils 232644 s'est terminé avec le code 0
Le processus fils 232514 s'est terminé avec le code 0
```

Figure 9: Affichage des états « suspendu » puis « reprend »

Bilan du TP2

Dans ce TP j'ai découvert la gestion des signaux :

- installation d'un handler avec `sigaction` ;
- usage de `waitpid` non bloquant pour récupérer tous les changements d'état ;
- remplacement du `waitpid` bloquant par `pause` pour le premier plan ;
- test des signaux SIGSTOP et SIGCONT sur un processus en arrière-plan.

Le shell gère maintenant la fin, la suspension et la reprise de ses processus sans laisser de zombies.

4 TP3 : Signaux avancés

4.1 Étapes à réaliser

1. Étape 12 — Test des frappes \hat{C} / \hat{Z}

Vérifier le comportement par défaut: `sleep 10` (avant-plan) puis `sleep 10 &` (arrière-plan), et taper \hat{C} / \hat{Z} .

2. Étape 13 — Protéger le minishell

- a) **13.1** Installer un handler pour SIGINT et SIGTSTP qui se contente d'afficher « Control + C ignoré ».
- b) **13.2** (Option SIG_IGN) — non retenu, car on garde le handler.
- c) **13.3** Masquer SIGINT et SIGTSTP via sigprocmask afin qu'aucun de ces signaux n'atteigne le père ni le fils.

3. Étape 14 — Détacher les tâches de fond

Mettre les processus lancés avec & dans un *autre groupe* grâce à setpggrp(), pour qu'ils ne reçoivent plus les signaux clavier destinés au groupe du minishell.

4.2 Implémentation et code

```
1  pid_t pid_global = -1;                // fils au 1 plan
2
3  // Handler SIGCHLD
4  void traitement(int sig)
5  {
6      int status;
7      pid_t pid_fork;
8      while ((pid_fork = waitpid(-1, &status,
9          WNOHANG | WUNTRACED | WCONTINUED)) > 0) {
10
11          if (pid_fork == pid_global)    // fils 1 plan terminé
12              pid_global = -1;
13
14          ...
15      }
16  }
17
18  // Handler ignorer pour SIGINT / SIGTSTP
19  void ignorer(int sig) {
20      printf(" Control + C ignoré, ");
21  }
22
23  int main(void)
24  {
25      // 13.1 : associer les handlers
26      struct sigaction action;
27      action.sa_handler = traitement;
28      sigemptyset(&action.sa_mask);
29      action.sa_flags = SA_RESTART;
30      sigaction(SIGCHLD, &action, NULL);
31
32      action.sa_handler = ignorer;
33      sigaction(SIGINT, &action, NULL);
34      sigaction(SIGTSTP, &action, NULL);
35  }
```

```

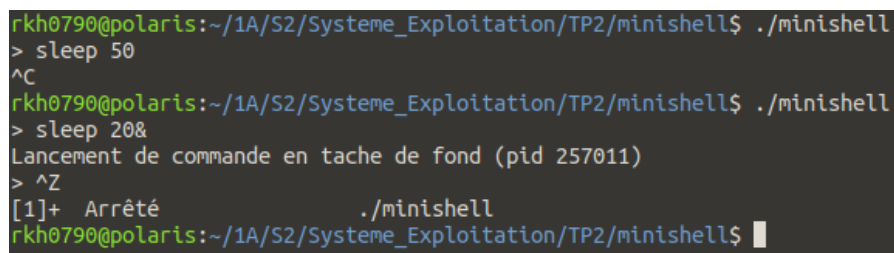
36 // 13.3 : masquer SIGINT et SIGTSTP
37 sigset_t masque;
38 sigemptyset(&masque);
39 sigaddset(&masque, SIGINT);
40 sigaddset(&masque, SIGTSTP);
41 sigprocmask(SIG_BLOCK, &masque, NULL);
42
43 // Boucle principale
44 if (pid_fork == 0) { // FILS
45     if (commande->backgrounded != NULL)
46         setpggrp(); // 14 : détache BG
47     execvp(cmd[0], cmd);
48     ...
49 } else { // PÈRE
50     if (commande->backgrounded == NULL) {
51         pid_global = pid_fork; // garde le PID 1 plan
52         while (pid_global != -1) // attend SIGCHLD
53             pause();
54     } else {
55         printf("Lancement de commande en tache de fond (pid %d)\n",
56             pid_fork);
57     }
58 }
59 }

```

4.3 Tests et résultats

1. Étape 12 — comportement initial

Avant les modifications, \hat{C} tue le minishell; \hat{Z} le suspend.



```

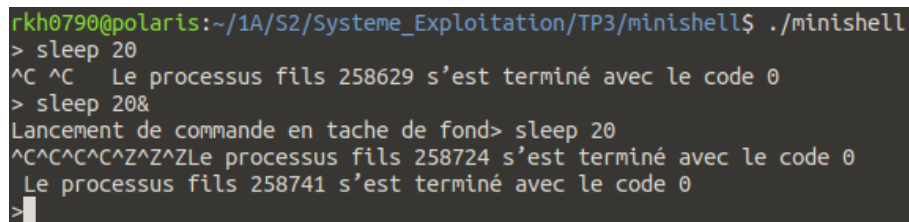
rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 50
^C
rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$ ./minishell
> sleep 20&
Lancement de commande en tache de fond (pid 257011)
> ^Z
[1]+  Arrêté ./minishell
rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP2/minishell$

```

Figure 10: Effet par défaut des frappes \hat{C} / \hat{Z}

2. Étape 13 — signaux bloqués

Après installation du handler *et* blocage de SIGINT/SIGTSTP : les frappes \hat{C} ou \hat{Z} n'interrompent plus ni le minishell ni le processus au premier plan.



```
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP3/minishell$ ./minishell
> sleep 20
^C ^C Le processus fils 258629 s'est terminé avec le code 0
> sleep 20&
Lancement de commande en tache de fond> sleep 20
^C^C^C^C^Z^Z^ZLe processus fils 258724 s'est terminé avec le code 0
Le processus fils 258741 s'est terminé avec le code 0
>
```

Figure 11: Plus aucune réaction visible à \hat{C} / \hat{Z}

3. Étape 14 — détachement des tâches de fond

Modifications côté fils (nécessaires avant l'étape 14).

Dans le `fork()`, on remet d'abord SIGINT/SIGTSTP à leur comportement normal, puis on place les tâches de fond dans un groupe séparé :

```
1  else if (pid_fork == 0) {           // Fils
2
3      // 1) Débloquent SIGINT / SIGTSTP hérités du père
4      sigprocmask(SIG_UNBLOCK, &masque, NULL);
5      signal(SIGINT, SIG_DFL);
6      signal(SIGTSTP, SIG_DFL);
7
8      // 2) Si commande lancée avec &, détache-la du groupe du shell
9      if (commande->backgrounded != NULL)
10         setpgrp();
11
12     execvp(cmd[0], cmd);
13     perror(cmd[0]);
14     _exit(EXIT_FAILURE);
15 }
```

Ce bloc assure que :

- le processus *avant-plan* reçoit bien les \hat{C} / \hat{Z} (signaux débloqués) ;
 - les commandes *arrière-plan* sont isolées dans un autre groupe ; elles n'hériteront donc plus des signaux clavier.
- a) On lance `sleep 20` au premier plan, puis on tape \hat{C} . Le handler annonce : « Le processus fils 268281 s'est terminé par le signal 2 » ; le minishell continue.
 - b) On démarre ensuite `sleep 500 &`. Le shell indique « Lancement de commande en tache de fond (pid 268390) ».
 - c) Dans un autre terminal : `kill -INT 268390`. Seul le `sleep` en arrière-plan reçoit le SIGINT et meurt ; le minishell reste au prompt.

```
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP3/minishell$ ./minishell
> sleep 20
^CLe processus fils 268281 s'est terminé par le signal 2
> sleep 500 &
Lancement de commande en tache de fond (pid 268390)
> ^CLe processus fils 268390 s'est terminé par le signal 2
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP3/minishell$ kill -INT 268390
rkho790@polaris:~/1A/S2/Systeme_Exploitation/TP3/minishell$
```

Figure 12: Signal SIGINT envoyé au processus BG (PID 268390) : il se termine, le minishell reste actif grâce à `setpgrp()`.

Bilan du TP 3

Dans ce TP j'ai surtout découvert la gestion des signaux :

- installer plusieurs handlers avec `sigaction()`;
- bloquer – puis débloquent – SIGINT et SIGTSTP grâce à `sigprocmask()`;
- attendre proprement le processus avant-plan avec une variable globale et `pause()`;
- mettre les tâches de fond dans leur propre groupe via `setpgrp()` pour qu'elles n'héritent plus des \hat{C} / \hat{Z} ;
- propager correctement les signaux clavier au seul processus avant-plan.

Le minishell sait maintenant ignorer les frappes \hat{C} / \hat{Z} pour lui-même, tout en laissant l'utilisateur contrôler chaque commande lancée.

5 TP4 : Redirections et manipulation de fichiers

5.1 Étapes à réaliser

1. **Étape 13 — redirections < et >**
Associer un fichier à l'entrée ou la sortie standard avant l'`execvp()`.
2. **Étape 14 — déplacement dans l'arborescence**
Implémenter la commande interne `cd`.
3. **Étape 15 — listage de répertoire**
Ajouter la commande interne `dir` (affiche le contenu d'un dossier).

5.2 Implémentation et code

```
1 // redirection dans le fils
2 if (commande->in != NULL) {                                // < fichier
3     int fd_in = open(commande->in, O_RDONLY);
4     if (fd_in < 0) { perror(commande->in); _exit(EXIT_FAILURE); }
```

```

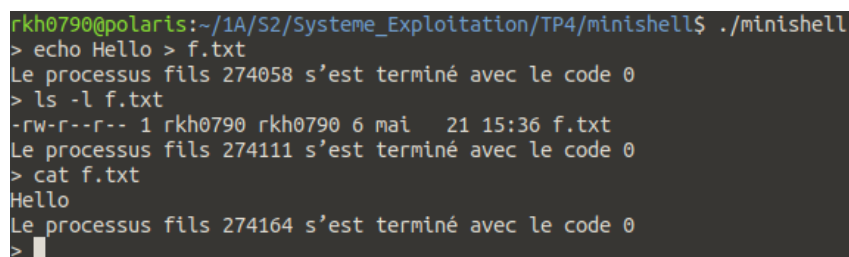
5     dup2(fd_in, STDIN_FILENO);
6     close(fd_in);
7 }
8 if (commande->out != NULL) {           // > fichier
9     int fd_out = open(commande->out,
10                        O_WRONLY | O_CREAT | O_TRUNC, 0644);
11     if (fd_out < 0) { perror(commande->out); _exit(EXIT_FAILURE); }
12     dup2(fd_out, STDOUT_FILENO);
13     close(fd_out);
14 }
15
16 // restauration des signaux (remarque du prof)
17 signal(SIGINT, SIG_DFL);
18 signal(SIGTSTP, SIG_DFL);
19
20 // commandes internes
21 void commande_cd(char **cmd) {         // Étape 14
22     char *chemin = (cmd[1] ? cmd[1] : getenv("HOME"));
23     if (chdir(chemin) < 0) perror("cd");
24 }
25
26 void commande_dir(char **cmd) {        // Étape 15
27     char *chemin = (cmd[1] ? cmd[1] : ".");
28     DIR *rep = opendir(chemin);
29     if (!rep) { perror("dir"); return; }
30     struct dirent *ent;
31     while ((ent = readdir(rep))) printf("%s\n", ent->d_name);
32     closedir(rep);
33 }

```

5.3 Tests et résultats

1. Redirection de sortie

echo Hello > f.txt crée le fichier puis cat f.txt affiche « Hello ».



```

rkh0790@polaris:~/1A/S2/Systeme_Exploitation/TP4/minishell$ ./minishell
> echo Hello > f.txt
Le processus fils 274058 s'est terminé avec le code 0
> ls -l f.txt
-rw-r--r-- 1 rkh0790 rkh0790 6 mai 21 15:36 f.txt
Le processus fils 274111 s'est terminé avec le code 0
> cat f.txt
Hello
Le processus fils 274164 s'est terminé avec le code 0
>

```

Figure 13: > : la sortie est redirigée vers f.txt

2. Redirection d'entrée

`wc -w < f.txt` retourne « 1 ».

```
rk0790@polaris:~/1A/S2/Systeme_Exploitation/TP4/minishell$ ./minishell
> echo Hello > f.txt
Le processus fils 275716 s'est terminé avec le code 0
> wc -w < f.txt
1
Le processus fils 275791 s'est terminé avec le code 0
>
```

Figure 14: `<` : l'entrée standard provient de *f.txt*

3. Commande interne `cd`

`cd /tmp` puis `pwd` : le répertoire courant devient `/tmp`.

```
rk0790@polaris:~/1A/S2/Systeme_Exploitation/TP4/minishell$ ./minishell
> cd /tmp
> pwd
/tmp
Le processus fils 277298 s'est terminé avec le code 0
>
```

Figure 15: Changement de répertoire avec `cd`

4. Commande interne `dir`

`dir /etc` liste le contenu de `/etc`.

```
rk0790@polaris:~/1A/S2/Systeme_Exploitation/TP4/minishell$ ./minishell
> dir /etc
.
..
sudoers
csh.cshrc
groff
autofs.conf
sudoers.d
usb_modeswitch.conf
chatscripts
mailcap
dbus-1
e2scrub.conf
security
update-notifier
NetworkManager
```

Figure 16: Affichage du contenu d'un dossier avec `dir`

Bilan du TP4

- mise en place des redirections avec `dup2()`;
- ajout des commandes internes `cd` et `dir`;
- restauration de `SIGINT`/`SIGTSTP` dans le fils (correction B-) : on peut maintenant couper ou suspendre une commande avant-plan ;
- minishell gère désormais les fichiers et la navigation dans l'arborescence.

6 TP5 : Tubes et pipelines

6.1 Étapes à réaliser

1. **Étape 16 — pipeline à deux commandes**
Gérer `commande1` `commande2` et prendre en compte les redirections éventuelles `<` et `>`.
2. **Étape 17 — pipeline à n commandes**
Étendre l'étape précédente pour pouvoir enchaîner un nombre arbitraire de filtres ;
exemple : `cat f.txt | grep int | wc -l`.

6.2 Implémentation et code

```

1 // Détection d'un pipeline simple
2 if (commande->seq[1] != NULL && commande->seq[2] == NULL) {
3     int tube[2];
4     if (pipe(tube) < 0) { perror("pipe"); continue; }
5
6     // 1 FILS : commande de gauche
7     if (fork() == 0) {
8         signal(SIGINT, SIG_DFL);
9         signal(SIGTSTP, SIG_DFL);
10        close(tube[0]); // ferme lecture
11
12        if (commande->in) { // redirection <
13            int fd = open(commande->in, O_RDONLY);
14            if (fd < 0) { perror("open in"); exit(EXIT_FAILURE); }
15            dup2(fd, STDIN_FILENO); close(fd);
16        }
17
18        dup2(tube[1], STDOUT_FILENO); // stdout → pipe
19        close(tube[1]);
20
21        execvp(commande->seq[0][0], commande->seq[0]);
22        perror("execvp 1"); exit(EXIT_FAILURE);
23    }
24
25    // 2 FILS : commande de droite
26    if (fork() == 0) {
27        signal(SIGINT, SIG_DFL);
28        signal(SIGTSTP, SIG_DFL);
29        close(tube[1]); // ferme écriture
30
31        if (commande->out) { // redirection >
32            int fd = open(commande->out,
33                          O_WRONLY|O_CREAT|O_TRUNC, 0644);
34            if (fd < 0) { perror("open out"); exit(EXIT_FAILURE); }
35            dup2(fd, STDOUT_FILENO); close(fd);
36        }
37

```



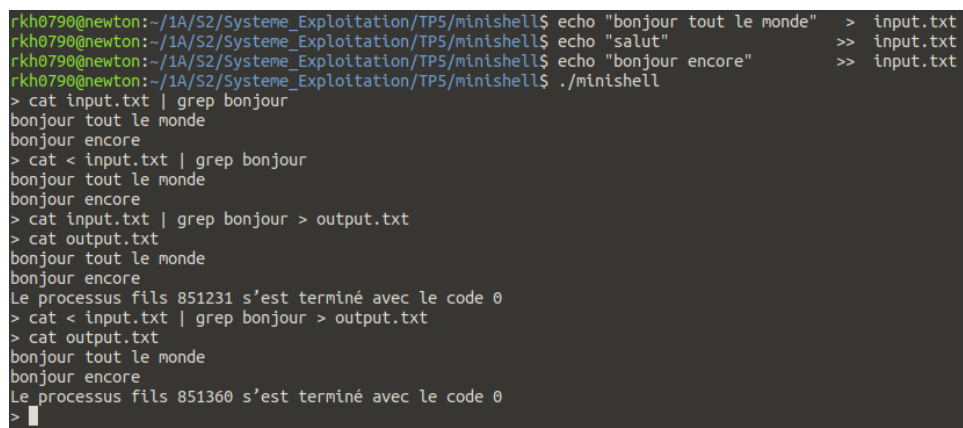
```

38     dup2(tube[0], STDIN_FILENO);           // stdin ← pipe
39     close(tube[0]);
40
41     execvp(commande->seq[1][0], commande->seq[1]);
42     perror("execvp 2"); exit(EXIT_FAILURE);
43 }
44
45 / Père : ferme le tube et attend
46 close(tube[0]);
47 close(tube[1]);
48 wait(NULL);
49 wait(NULL);
50 continue;
51 }
```

6.3 Tests et résultats

1. Pipeline + redirections

Fichier d'entrée : `input.txt`. `cat < input.txt grep bonjour > output.txt` doit produire un fichier `output.txt` ne contenant que les lignes où apparaît « bonjour ».



```

rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "bonjour tout le monde" > input.txt
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "salut" >> input.txt
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "bonjour encore" >> input.txt
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ ./minishell
> cat input.txt | grep bonjour
bonjour tout le monde
bonjour encore
> cat < input.txt | grep bonjour
bonjour tout le monde
bonjour encore
> cat input.txt | grep bonjour > output.txt
> cat output.txt
bonjour tout le monde
bonjour encore
Le processus fils 851231 s'est terminé avec le code 0
> cat < input.txt | grep bonjour > output.txt
> cat output.txt
bonjour tout le monde
bonjour encore
Le processus fils 851360 s'est terminé avec le code 0
>
```

Figure 17: pipeline à deux commandes avec redirection d'entrée et de sortie

Évolution du code pour l'étape 17 La gestion « deux commandes » de l'étape 16 a été **commentée** et remplacée par une boucle capable d'enchaîner un nombre arbitraire de filtres :

- **détection** : si `commande->seq[1] != NULL`, on sait qu'il existe au moins un tube ;
- **boucle** sur chaque maillon `i` :
 - a) création d'un pipe `pipe(tube)` sauf pour le dernier ;
 - b) `fork()` : le fils
 - lit sur l'extrémité lecture du pipe précédent (sauf pour le tout premier) ;
 - écrit sur l'extrémité écriture du pipe courant (sauf pour le dernier) ;

- applique < uniquement sur la première commande ;
 - applique > uniquement sur la dernière commande ;
 - exécute `execvp(maillon[0], maillon)`.
- c) le père ferme les descripteurs devenus inutiles et prépare `in_fd` pour le tour suivant.
- enfin, le père boucle sur `wait(NULL)` pour récupérer tous les fils.

```

1 // Détection d'un pipeline à n commandes
2 int in_fd = STDIN_FILENO;
3 for (int i = 0; commande->seq[i]; i++) {
4     int tube[2], use_pipe = (commande->seq[i+1] != NULL);
5     if (use_pipe && pipe(tube) < 0) { perror("pipe"); break; }
6
7     if (fork() == 0) { // FILS
8         ...
9         execvp(commande->seq[i][0], commande->seq[i]);
10        perror("execvp"); _exit(1);
11    }
12    // PÈRE : ferme in_fd et prépare le suivant
13    if (in_fd != STDIN_FILENO) close(in_fd);
14    if (use_pipe) { close(tube[1]); in_fd = tube[0]; }
15 }
16 while (wait(NULL) > 0);

```

(a) Pipeline à trois commandes

`cat toto.c lulu.c grep int | wc -l` retourne 3.

`cat toto.c lulu.c grep int | wc -l > n.txt` crée *n.txt* contenant 3.

(b) Pipeline à quatre commandes

`cat toto.c lulu.c tr a-z A-Z | grep INT | sort | uniq`

produit les lignes en majuscules, triées et sans doublons.

```

rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$ echo "int a = 0;" > toto.c
rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$ echo "float b = 1.0;" >> toto.c
rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$ echo "int main() { return 0; }" >> toto.c
rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$
rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$ echo "double x = 3.14;" > lulu.c
rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$ echo "int count = 5;" >> lulu.c
rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$ echo "/* commentaire */" >> lulu.c
rkh0790@newton:~/1A/S2/Système_Exploitation/TP5/minishell$ ./minishell
> cat toto.c lulu.c | grep int | wc -l
3
Le processus fils 857799 s'est terminé avec le code 0
> cat toto.c lulu.c | grep int | wc -l > n.txt
> cat n.txt
3
Le processus fils 857872 s'est terminé avec le code 0
> cat < toto.c lulu.c | grep int | wc -l
1
> cat toto.c lulu.c | tr a-z A-Z | grep INT | sort | uniq
INT A = 0;
INT COUNT = 5;
INT MAIN() { RETURN 0; }
>

```

Figure 18: Exemples de pipelines validant l'étape 17

Bilan du TP 5

Le minishell gère désormais un tube simple :

- création d'un `pipe()` ;
- deux `fork()` pour les commandes gauche / droite ;
- `dup2()` appropriés, `close()` descripteurs inutiles ;
- prise en compte des redirections `<` et `>` dans le contexte du pipeline.

L'étape 17 (pipeline de taille arbitraire) reste à implémenter.

7 Conclusion

Ce projet *Minishell* m'a vraiment permis de mettre en pratique, pas à pas, tout ce qu'on a vu en cours de Systèmes d'exploitation. Au fil des cinq TP, on est partis d'un simple `readcmd()` qui affichait la ligne tapée pour arriver à un mini-shell capable de gérer les processus, les signaux, les redirections et un pipeline de longueur arbitraire.

J'ai particulièrement apprécié le format où l'on travaillait sur les étapes du projet pendant les séances encadrées de TPs: on codait, on posait nos questions au prof directement, puis on remettait une archive et on recevait un retour clair avant la séance suivante. Ça m'a forcé à corriger mes erreurs (par exemple remettre les signaux par défaut dans les fils, ou penser aux redirections dans le pipeline) au fur et à mesure, au lieu de tout découvrir à la fin.

Les points qui étaient un peu flous pour moi :

- bien séparer le père et le fils après chaque `fork()`, surtout quand il fallait enchaîner deux puis plusieurs filtres ;
- gérer `SIGINT` et `SIGTSTP` : bloquer pour le minishell mais laisser passer pour le processus avant-plan ;
- ne pas oublier de fermer les bons descripteurs de tube, sous peine de rester bloqué sur un `read()` ou de créer des fuites de FD.

Au final, le minishell exécute la plupart des commandes courantes, supporte `cd` et `dir`, les redirections, les tâches de fond et les pipelines.