Project Report

Minishell

Table of Contents

Prepared by:
Ralph Khairallah

Date: June 4, 2025

1 Introduction

During this semester, I worked on building a **Minishell**. The idea is simple: to recreate, in a very lightweight way, what a Unix shell does every day: read a line, interpret it, execute the requested programs, then wait for the next one.

Why create a shell? Because it forces us to work with low-level system calls: `fork` to duplicate a process, `exec` to change the program, `wait` to retrieve the return code, `pipe` and `dup2` to connect input/output streams, and also signal handling (`SIGINT`, `SIGTSTP`, ...).

How does it work? The Minishell:

a) displays a prompt and reads the typed line;

b) parses this line to separate the command, its arguments, redirections (`<`, `>`, `»`) and pipes (`|`);

c) creates one or more child processes with `fork()`, replaces their code with the right command using `execvp()`, and connects inputs/outputs using `dup2()` when needed;

d) intercepts keyboard signals (Ctrl-C, Ctrl-Z) to properly terminate or suspend;

e) waits for child termination (`waitpid()`) before displaying the prompt again.

Project breakdown The work was divided across five lab sessions:

- **Lab 1**: execute a simple command;
- **Lab 2**: intercept basic signals;
- **Lab 3**: handle these signals when multiple processes run in parallel;
- **Lab 4**: add input/output redirections;
- **Lab 5**: connect multiple commands using pipes.

Report contents For each lab, you will find:

- a reminder of the objectives;
- the implementation in code;
- the tests performed and the results obtained.

2 Lab 1: Processes and Sequential Execution

2.1 Steps to Follow

1. **Step 1 — Test the program**
   Compile the original minishell with `make`, run it (`./minishell`), and check that it simply displays the typed line and then the prompt. No code changes are needed at this stage.

2. **Step 2 — Launching a command**
   Add a `fork()` and, in the child, an `execvp()` to replace the process image with the requested command. The parent does nothing else, and the prompt reappears immediately.

3. **Step 3 — Sequential chaining**
   Insert a `waitpid()` in the parent to wait for the child to finish before reading the next line. This gives the sequence: child creation, *exec*, wait, then return to prompt.

4. **Step 4 — Background task**
   Detect the presence of a `&` using the `commande->backgrounded` field. If this field is not `NULL`, the parent *does not wait*; it simply displays a message with the child's `pid`.

2.2 Implementation and Code

```
while (!fini) {
    printf("> ");
    struct cmdline *commande = readcmd();

    int indexseq = 0;
    char **cmd;
    while ((cmd = commande->seq[indexseq])) {
        if (cmd[0] && strcmp(cmd[0], "exit") == 0) {
            fini = true;
            printf("Goodbye ...\n");
        } else if (cmd[0]) {
            pid_t pid_fork = fork();
            if (pid_fork == -1) {
                perror("fork");
                exit(EXIT_FAILURE);
            } else if (pid_fork == 0) {          // CHILD
                execvp(cmd[0], cmd);             // Step 2
                perror(cmd[0]);                  // if exec fails
                exit(EXIT_FAILURE);
            } else {                             // PARENT
                if (commande->backgrounded == NULL) { // Step 3
                    int status;
                    if (waitpid(pid_fork, &status, 0) != -1) {
                        if (WIFEXITED(status))
                            printf("Child %d ended (code %d)\n",
                                   pid_fork, WEXITSTATUS(status));
                        else if (WIFSIGNALED(status))
                            printf("Child %d killed by signal %d\n",
                                   pid_fork, WTERMSIG(status));
                    }
                } else {                         // Step 4
                    printf("Launching in background (pid %d)\n",
                           pid_fork);
                }
            }
```

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

2.3 Tests and Results

1. **Step 1: Basic launch**
   Type any command and check that the shell simply re-displays the line.

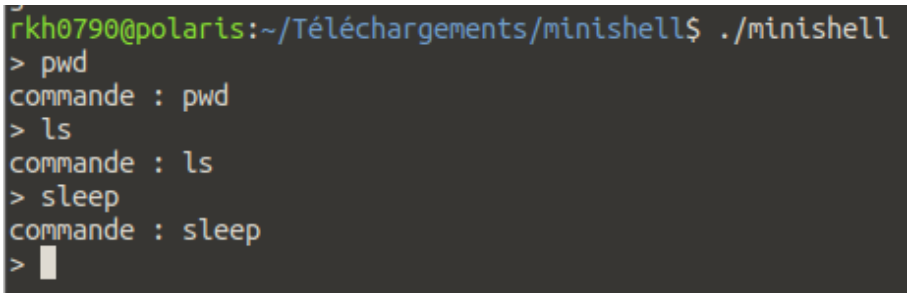

Figure 1: Initial behavior (Step 1)

2. **Step 2: Command execution**
   `ls -l` should execute in the child, and the prompt returns immediately.

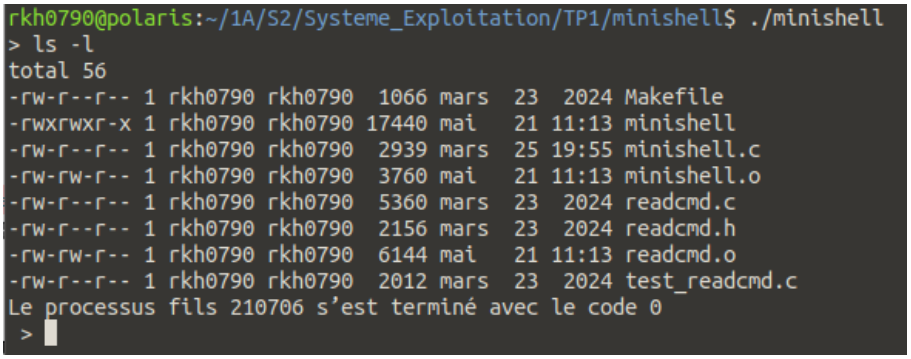

Figure 2: `ls -l` launched via fork/exec

3. **Step 3: Blocking sequence**
   Run `echo OK` followed by `date`: the date only appears after "OK".

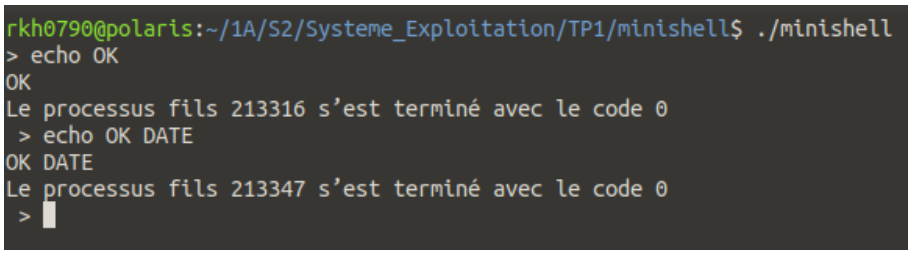

Figure 3: Waiting using `waitpid`

4. **Step 4: Background task**
   `sleep 5 &` should immediately return control and display the message "Launching in background".

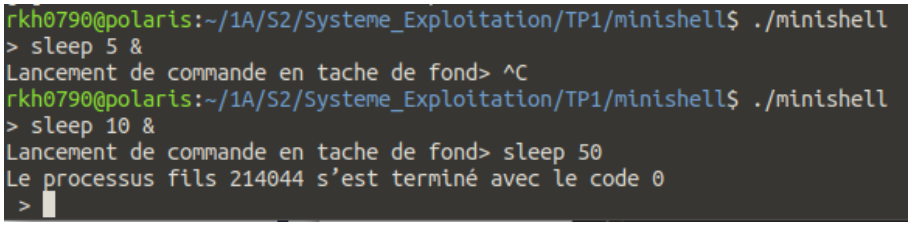

Figure 4: Command launched in background

Summary of Lab 1

This first lab mostly taught me how to launch a program from my own shell. I learned how to:

- read the line typed by the user;
- duplicate the process with `fork()`;
- replace the child's code using `execvp()`;
- wait for the child to finish with `waitpid()`, or let it run in the background using `&`.

The trickiest part was clearly separating the "parent" and the "child" after the `fork()` and checking the return value of each system call.

At this stage, the Minishell can already execute simple commands, display the return code, and launch a background process. This provides a solid foundation for the next steps: signal handling, redirections, and pipelines.

3 Lab 2: Signals and Process Management

3.1 Steps to Follow

1. **Step 5 — Handling `SIGCHLD`**
   Install a handler using `sigaction()` that displays a message when a child changes its state.

2. **Step 6 — Retrieving child status**
   In the handler, loop with `waitpid(-1, status, WNOHANG | WUNTRACED | WCONTINUED)` to handle *all* terminated, stopped, or resumed child processes.

3. **Step 7 — Waiting via `pause()`**
   For a foreground command, replace the blocking `waitpid()` with a simple `pause()`: the parent sleeps until `SIGCHLD` is received.

4. **Step 8 — Stop/Continue on a background child**
   Test sending `SIGSTOP` then `SIGCONT` (with `kill`) to a process launched with `&`.

5. **Step 9 — Detailed messages**
   In the handler: distinguish normal exit, termination by signal, suspension, and resume using the macros `WIFEXITED`, `WIFSIGNALED`, `WIFSTOPPED`, `WIFCONTINUED`.

3.2 Implementation and Code

```c
// Handling the SIGCHLD signal
void traitement(int sig)
{
    int status;
    pid_t pid_fork;
    // Retrieve all children that changed state
    while ((pid_fork = waitpid(-1, &status,
            WNOHANG | WUNTRACED | WCONTINUED)) > 0) {
        if (WIFEXITED(status))
            printf("Child process %d exited with code %d\n",
                   pid_fork, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("Child process %d was terminated by signal %d\n",
                   pid_fork, WTERMSIG(status));
        else if (WIFSTOPPED(status))
            printf("Child process %d is suspended (signal %d)\n",
                   pid_fork, WSTOPSIG(status));
        else if (WIFCONTINUED(status))
            printf("Child process %d resumed\n", pid_fork);
    }
}

int main(void)
{
    // Step 5, handler for SIGCHLD
    struct sigaction action;
    action.sa_handler = traitement;
    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_RESTART;
    sigaction(SIGCHLD, &action, NULL);

    bool fini = false;
    while (!fini) {
        printf("> ");
```

```
35          struct cmdline *commande = readcmd();
36
37          int indexseq = 0;
38          char **cmd;
39          while ((cmd = commande->seq[indexseq])) {
40
41              // Exit if the user types 'exit'
42              if (cmd[0] && strcmp(cmd[0], "exit") == 0) {
43                  fini = true;
44                  printf("Goodbye ...\n");
45              }
46              // Otherwise, run the command
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) {        // CHILD
53                      execvp(cmd[0], cmd);
54                      perror(cmd[0]);                // exec failed
55                      exit(EXIT_FAILURE);
56                  } else {                           // PARENT
57                      if (commande->backgrounded == NULL) {
58                          pause();                   // Step 7: wait for
59                                                     // a SIGCHLD
60                      } else {
61                          printf("Launching background command\n");
62                      }
63                  }
64              }
65              indexseq++;
66          }
67      }
68      return EXIT_SUCCESS;
69  }
```

3.3 Tests and Results

1. **Step 5: Message at the end of a child process**
   Launch `sleep` in both foreground and background; a message should appear when the `sleep` finishes.

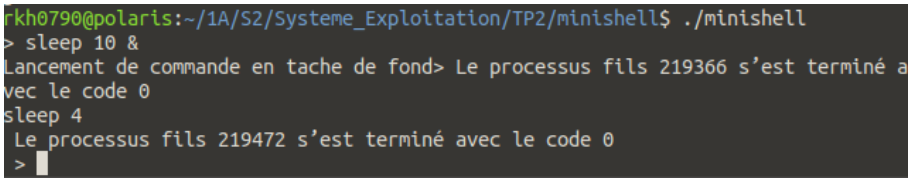

Figure 5: `SIGCHLD` signal captured

2. **Step 6: Multiple children**
   Launch three `sleep 5 &` commands in a row; all three terminations should be detected with no zombie processes.

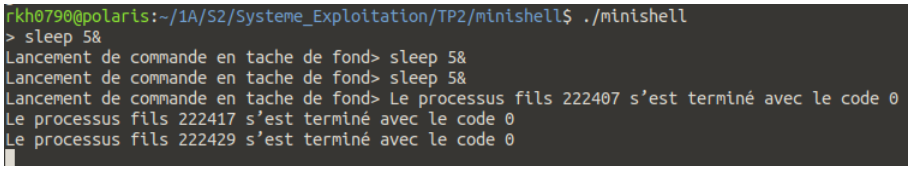

Figure 6: Loop on non-blocking `waitpid`

3. **Step 7: pause for the foreground**
   Run `sleep 10&`; the background launch message appears immediately.

   Run `sleep 5` (without `&`); the shell blocks, then returns to the prompt after 5 seconds.

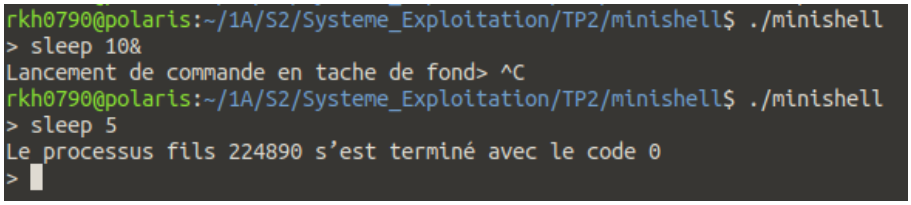

Figure 7: `pause()` unblocked by `SIGCHLD`

4. **Step 8: stop / continue**
   Run `sleep 50 &`; the shell immediately displays `[BG] pid 232514`. `kill -STOP 232514` suspends the process: the handler prints "Child process 232514 is suspended (signal 19)". `kill -CONT 232514` resumes it: the handler prints "Child process 232514 resumed".

```
rkh0790@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 and resume of process 232514

5. **Step 9: Detailed messages**
   The capture clearly shows both handler messages (*suspended* then *resumed*), confirming correct handling of `SIGSTOP` and `SIGCONT`.

```
rkh0790@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: Display of the "suspended" then "resumed" states

Summary of Lab 2

In this lab, I discovered signal handling:

- installing a handler using `sigaction`;
- using non-blocking `waitpid` to retrieve all state changes;
- replacing blocking `waitpid` with `pause` for foreground tasks;
- testing `SIGSTOP` and `SIGCONT` signals on background processes.

The shell now handles termination, suspension, and resumption of its processes without leaving zombies.

4 Lab 3: Advanced Signal Handling

4.1 Steps to Follow

1. **Step 12 — Test key presses Ĉ / Ẑ**
   Verify the default behavior: `sleep 10` (foreground) then `sleep 10 &` (background), and press Ĉ / Ẑ.

2. **Step 13 — Protect the minishell**

   a) **13.1** Install a handler for `SIGINT` and `SIGTSTP` that simply displays "Control + C ignored".

   b) **13.2** (Option SIG_IGN) — not used, since the handler is kept.

   c) **13.3** Mask `SIGINT` and `SIGTSTP` via `sigprocmask` so that none of these signals reach the parent or the child.

3. **Step 14 — Detach background tasks**
   Place processes launched with `&` in a *separate group* using `setpgrp()` so they no longer receive keyboard signals sent to the minishell's group.

4.2 Implementation and Code

```c
pid_t pid_global = -1;            // foreground child process

// SIGCHLD Handler
void traitement(int sig)
{
    int status;
    pid_t pid_fork;
    while ((pid_fork = waitpid(-1, &status,
            WNOHANG | WUNTRACED | WCONTINUED)) > 0) {

        if (pid_fork == pid_global)        // foreground child has exited
            pid_global = -1;

        ...
    }
}

// Ignore handler for SIGINT / SIGTSTP
void ignorer(int sig) {
    printf(" Control + C ignored, ");
}

int main(void)
{
    // 13.1: assign signal handlers
    struct sigaction action;
    action.sa_handler = traitement;
    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_RESTART;
    sigaction(SIGCHLD, &action, NULL);

    action.sa_handler = ignorer;
    sigaction(SIGINT,  &action, NULL);
    sigaction(SIGTSTP, &action, NULL);

```

```
36      // 13.3: mask SIGINT and SIGTSTP
37      sigset_t masque;
38      sigemptyset(&masque);
39      sigaddset(&masque, SIGINT);
40      sigaddset(&masque, SIGTSTP);
41      sigprocmask(SIG_BLOCK, &masque, NULL);
42
43      // Main loop
44      if (pid_fork == 0) {                    // CHILD
45          if (commande->backgrounded != NULL)
46              setpgrp();                      // 14: detach background job
47          execvp(cmd[0], cmd);
48          ...
49      } else {                                // PARENT
50          if (commande->backgrounded == NULL) {
51              pid_global = pid_fork;          // store foreground child PID
52              while (pid_global != -1)        // wait for SIGCHLD
53                  pause();
54          } else {
55              printf("Launching background command (pid %d)\n",
56                     pid_fork);
57          }
58      }
59  }
```

4.3 Tests and Results

1. **Step 12 — Initial behavior**
   Before modifications, Ĉ kills the minishell; Ẑ suspends it.

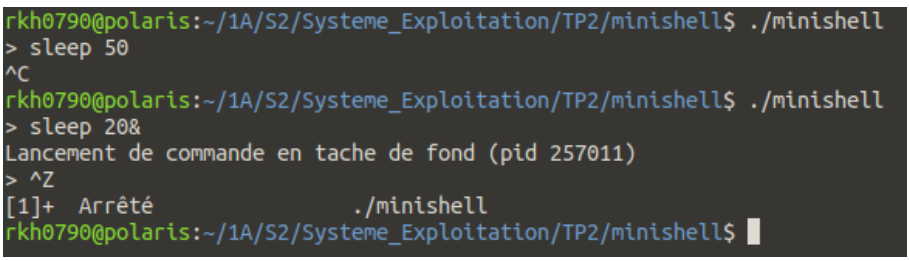

Figure 10: Default effect of Ĉ / Ẑ key presses

2. **Step 13 — Signals blocked**
   After installing the handler *and* blocking `SIGINT`/`SIGTSTP`: pressing Ĉ or Ẑ no longer interrupts the minishell or the foreground process.

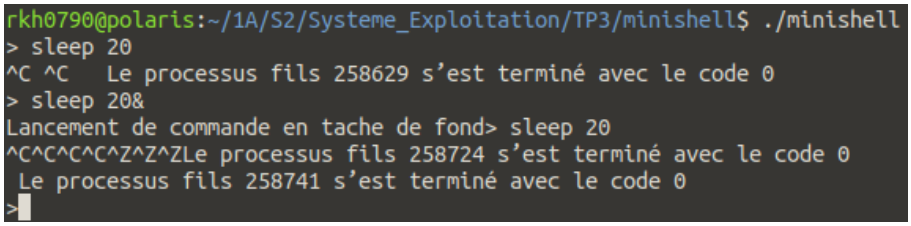

Figure 11: No visible reaction to Ĉ / Ẑ

3. **Step 14 — Detaching background tasks**
   Modifications on the child side (required before step 14).

   In the `fork()`, we first restore `SIGINT`/`SIGTSTP` to their default behavior, then place background tasks in a separate process group:

```
1   else if (pid_fork == 0) {          // Child
2
3       // 1) Unblock SIGINT / SIGTSTP inherited from parent
4       sigprocmask(SIG_UNBLOCK, &masque, NULL);
5       signal(SIGINT,  SIG_DFL);
6       signal(SIGTSTP, SIG_DFL);
7
8       // 2) If command launched with &, detach it from shell group
9       if (commande->backgrounded != NULL)
10          setpgrp();
11
12      execvp(cmd[0], cmd);
13      perror(cmd[0]);
14      _exit(EXIT_FAILURE);
15  }
```

   This block ensures that:

   - the *foreground* process does receive Ĉ/Ẑ (signals unblocked);
   - *background* commands are isolated in another group; they will no longer inherit keyboard signals.

   a) Run `sleep 20` in the foreground, then press Ĉ. The handler displays: "Child process *268281* was terminated by signal 2"; the minishell continues running.

   b) Then start `sleep 500 &`. The shell prints "Launching background command (pid *268390*)".

   c) From another terminal: `kill -INT 268390`. Only the background `sleep` receives the SIGINT and dies; the minishell remains at the prompt.

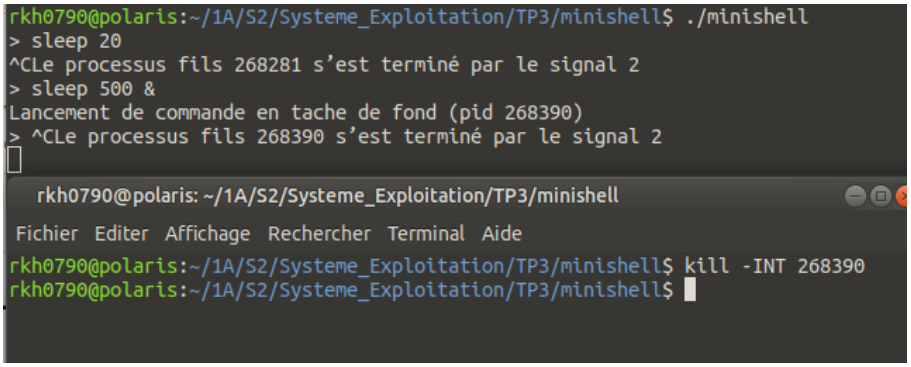

Figure 12: `SIGINT` signal sent to background process (PID 268390): it terminates, the minishell stays alive thanks to `setpgrp()`.

Summary of Lab 3

In this lab, I mainly explored advanced signal handling:

- installing multiple handlers using `sigaction()`;
- blocking – then unblocking – `SIGINT` and `SIGTSTP` using `sigprocmask()`;
- properly waiting for the foreground process using a global variable and `pause()`;
- placing background tasks in their own group via `setpgrp()` so they no longer inherit Ĉ / Ẑ;
- correctly forwarding keyboard signals to the foreground process only.

The minishell can now ignore Ĉ / Ẑ for itself, while allowing the user to control each launched command.

5 Lab 4: Redirections and File Management

5.1 Steps to Follow

1. **Step 13 — Redirections < and >**
   Associate a file with standard input or output before calling `execvp()`.
2. **Step 14 — Directory navigation**
   Implement the built-in command `cd`.
3. **Step 15 — Directory listing**
   Add the built-in command `dir` (displays the contents of a directory).

5.2 Implementation and Code

```
1 // redirection in the child process
2 if (commande->in != NULL) {                    // < file
3     int fd_in = open(commande->in, O_RDONLY);
4     if (fd_in < 0) { perror(commande->in); _exit(EXIT_FAILURE); }
```

```
    dup2(fd_in, STDIN_FILENO);
    close(fd_in);
}
if (commande->out != NULL) {                    // > file
    int fd_out = open(commande->out,
                      O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd_out < 0) { perror(commande->out); _exit(EXIT_FAILURE); }
    dup2(fd_out, STDOUT_FILENO);
    close(fd_out);
}

// restore signals (professor's note)
signal(SIGINT,  SIG_DFL);
signal(SIGTSTP, SIG_DFL);

// built-in commands
void commande_cd(char **cmd) {                  // Step 14
    char *chemin = (cmd[1] ? cmd[1] : getenv("HOME"));
    if (chdir(chemin) < 0) perror("cd");
}

void commande_dir(char **cmd) {                 // Step 15
    char *chemin = (cmd[1] ? cmd[1] : ".");
    DIR *rep = opendir(chemin);
    if (!rep) { perror("dir"); return; }
    struct dirent *ent;
    while ((ent = readdir(rep))) printf("%s\n", ent->d_name);
    closedir(rep);
}
```

5.3 Tests and Results

1. **Output redirection**
   `echo Hello > f.txt` creates the file, then `cat f.txt` displays "Hello".

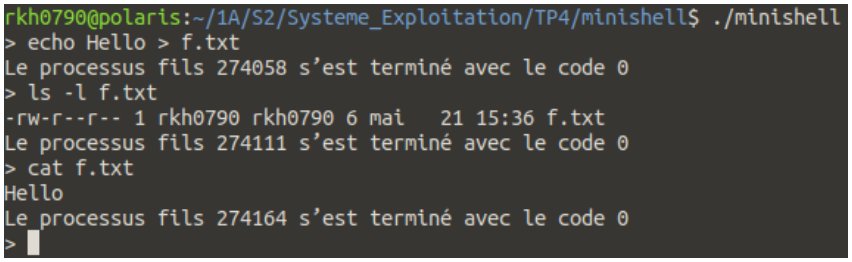

Figure 13: `>` : output redirected to *f.txt*

2. **Input redirection**
   `wc -w < f.txt` returns "1".

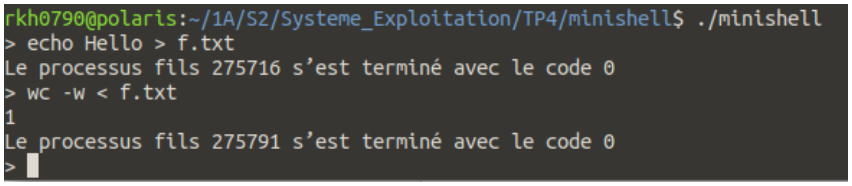

Figure 14: `<` : standard input comes from *f.txt*

3. **Built-in `cd` command**
   `cd /tmp` followed by `pwd`: the current directory becomes `/tmp`.

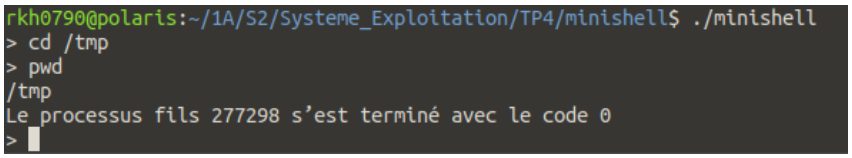

Figure 15: Changing directory using `cd`

4. **Built-in `dir` command**
   `dir /etc` lists the contents of `/etc`.

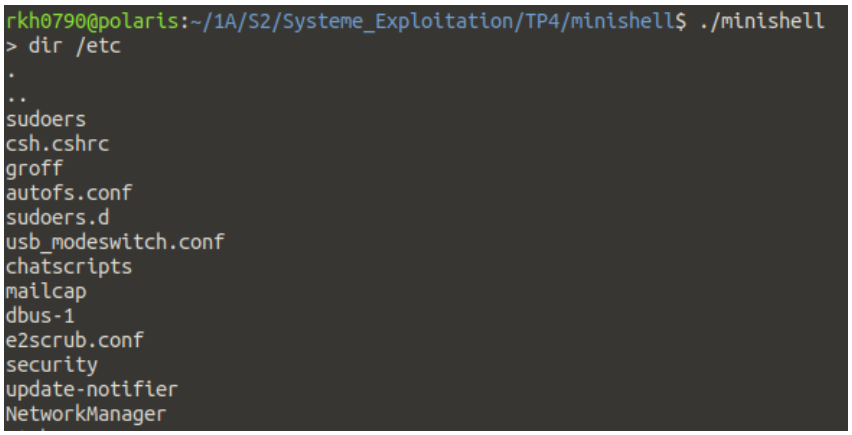

Figure 16: Displaying folder contents using `dir`

Summary of Lab 4

- implemented redirections using `dup2()`;
- added the built-in commands `cd` and `dir`;
- restored `SIGINT`/`SIGTSTP` handling in the child (fix suggested during code review): it is now possible to interrupt or suspend a foreground command;
- the minishell now handles files and directory navigation.

6 Lab 5: Pipes and Pipelines

6.1 Steps to Follow

1. **Step 16 — Pipeline with Two Commands**
   Handle `command1  command2|` and also account for possible input/output redirections `<` and `>`.

2. **Step 17 — Pipeline with n Commands**
   Extend the previous step to support chaining an arbitrary number of filters; example: `cat f.txt  grep int | wc -l|`.

6.2 Implementation and Code

```c
// Detect a simple two-command pipeline
if (commande->seq[1] != NULL && commande->seq[2] == NULL) {
    int tube[2];
    if (pipe(tube) < 0) { perror("pipe"); continue; }

    // 1st CHILD: left-hand command
    if (fork() == 0) {
        signal(SIGINT,  SIG_DFL);
        signal(SIGTSTP, SIG_DFL);
        close(tube[0]);                     // close read end

        if (commande->in) {                 // input redirection <
            int fd = open(commande->in, O_RDONLY);
            if (fd < 0) { perror("open in"); exit(EXIT_FAILURE); }
            dup2(fd, STDIN_FILENO); close(fd);
        }

        dup2(tube[1], STDOUT_FILENO);       // stdout → pipe
        close(tube[1]);

        execvp(commande->seq[0][0], commande->seq[0]);
        perror("execvp 1"); exit(EXIT_FAILURE);
    }

    // 2nd CHILD: right-hand command
    if (fork() == 0) {
        signal(SIGINT,  SIG_DFL);
        signal(SIGTSTP, SIG_DFL);
        close(tube[1]);                     // close write end

        if (commande->out) {                // output redirection >
            int fd = open(commande->out,
                          O_WRONLY|O_CREAT|O_TRUNC, 0644);
            if (fd < 0) { perror("open out"); exit(EXIT_FAILURE); }
            dup2(fd, STDOUT_FILENO); close(fd);
        }

```

```
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      // Parent: close pipe and wait
46      close(tube[0]);
47      close(tube[1]);
48      wait(NULL);
49      wait(NULL);
50      continue;
51  }
```

6.3 Tests and Results

1. **Pipeline + Redirections**
   Input file: `input.txt`. `cat < input.txt  grep bonjour > output.txt|` should produce an `output.txt` file containing only the lines where "bonjour" appears.

```
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: Two-command pipeline with input and output redirection

Code Evolution for Step 17 The "two-command" implementation of step 16 has been **commented out** and replaced by a loop capable of chaining an arbitrary number of filters:

- **Detection**: if `commande->seq[1] != NULL`, we know there's at least one pipe;
- **Loop** over each stage `i`:
  a) create a pipe with `pipe(tube)` (except for the last command);
  b) `fork()`: the child:
     – reads from the previous pipe's read end (except for the first command);
     – writes to the current pipe's write end (except for the last command);
     – applies `<` only to the first command;
     – applies `>` only to the last command;

– executes `execvp(maillon[0], maillon)`.

c) the parent closes unused file descriptors and prepares `in_fd` for the next iteration.

- finally, the parent loops with `wait(NULL)` to clean up all children.

```c
// Detection of an n-command pipeline
int in_fd = STDIN_FILENO;
for (int i = 0; commande->seq[i]; i++) {
    int tube[2], use_pipe = (commande->seq[i+1] != NULL);
    if (use_pipe && pipe(tube) < 0) { perror("pipe"); break; }

    if (fork() == 0) {                      // CHILD
        ...
        execvp(commande->seq[i][0], commande->seq[i]);
        perror("execvp"); _exit(1);
    }
    // PARENT: close in_fd and set up for next
    if (in_fd != STDIN_FILENO) close(in_fd);
    if (use_pipe) { close(tube[1]); in_fd = tube[0]; }
}
while (wait(NULL) > 0);
```

(a) **Three-command pipeline**
`cat toto.c lulu.c` grep int | wc -l| returns 3.
`cat toto.c lulu.c` grep int | wc -l > n.txt| creates *n.txt* containing 3.

(b) **Four-command pipeline**
`cat toto.c lulu.c` tr a-z A-Z | grep INT | sort | uniq|
outputs uppercase lines, sorted and with duplicates removed.

```
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "int a = 0;"           >  toto.c
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "float b = 1.0;"       >> toto.c
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "int main() { return 0; }" >> toto.c
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "double x = 3.14;"     >  lulu.c
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "int count = 5;"       >> lulu.c
rkh0790@newton:~/1A/S2/Systeme_Exploitation/TP5/minishell$ echo "/* commentaire */"    >> lulu.c
rkh0790@newton:~/1A/S2/Systeme_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: Examples of pipelines satisfying Step 17

Summary of Lab 5

The minishell now supports a simple pipeline:

- creating a `pipe()`;
- two `fork()` calls for left/right commands;
- appropriate use of `dup2()`, closing unused file descriptors;
- handling redirections `<` and `>` in the context of pipelines.

Step 17 (arbitrary-length pipelines) remains to be implemented.

7 Conclusion

This *Minishell* project truly allowed me to put into practice— step by step—everything we covered in our Operating Systems course. Throughout the five labs, we progressed from a simple `readcmd()` that just printed the typed line, to a full-featured mini-shell capable of handling processes, signals, redirections, and pipelines of arbitrary length.

I particularly appreciated the format where we worked on the project step-by-step during supervised lab sessions: we would code, ask the instructor our questions directly, submit an archive, and then receive clear feedback before the next session. This approach forced me to correct my mistakes as I went along (for example, restoring default signals in child processes, or handling redirections in pipelines), instead of discovering all issues at the end.

The points that were initially a bit unclear for me:

- properly separating parent and child processes after each `fork()`, especially when chaining two or more filters;
- handling `SIGINT` and `SIGTSTP`: blocking them for the minishell but letting them reach foreground processes;
- remembering to close the correct pipe descriptors, to avoid blocking on a `read()` or leaking file descriptors.

In the end, the minishell runs most common commands, supports `cd` and `dir`, handles redirections, background tasks, and pipelines.