Minishell

1. Introduction

La méthodologie de test a été d'interagir avec le shell et d'observer son comportement, en vérifiant que celui-ci était conforme aux spécifications.

La structure et les choix d'immplémentations seront ensuite expliqués au fur et à mesure du code, en se basant sur le principe du *literate programming*.

Tout le minishell est compris dans un seul fichier.

2. Procédure des tests

- Question 1

```
/home/ewen/enseeiht/os/projects
🐚 ls
...
```

- Question 3

```
/home/ewen/enseeiht/os/projects
🐚 sleep 3

(3 secondes plus tard)

/home/ewen/enseeiht/os/projects
🐚 ls
```

- Question 4

```
/home/ewen/enseeiht/os/projects
🐚 cd ~

/home/ewen/
🐚 exit

(le shell termine)
```

- Question 5 et 6

```
/home/ewen/enseeiht/os/projects
🐚 sleep 600 &

(immédiatement)

/home/ewen/enseeiht/os/projects
🐚 lj
🔴 1 (26625) sleep

/home/ewen/enseeiht/os/projects
🐚 sj 1

/home/ewen/enseeiht/os/projects
🐚 fg 1
(600 secondes après ...)

/home/ewen/enseeiht/os/projects
🐚
```

- Question 7

```
/home/ewen/enseeiht/os/projects
🐚 sleep 600

(on envoie ^Z)

/home/ewen/enseeiht/os/projects
🐚 lj
🔴 1 (26625) sleep
```

- Question 8

```
/home/ewen/enseeiht/os/projects
🐚 sleep 600

(on envoie ^C)

/home/ewen/enseeiht/os/projects
🐚 lj
(rien)
```

- Question 9

```
/home/ewen/enseeiht/os/projects
🐚 echo quoi > feur

/home/ewen/enseeiht/os/projects
🐚 cat < feur
quoi
```

- Question 10 et 11

```
/home/ewen/enseeiht/os/projects
🐚 ls | wc -l
23

/home/ewen/enseeiht/os/projects
🐚 wget https://loca7.fr/appartements.json -O- | grep -o hasBicycleParking | uniq | wc -l
1
```

3. Importations

On commence par imporeter toutes les dépendances.

```c
#define _POSIX_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdarg.h>
#include "readcmd.h"
```

4. Fonctions utilitaires

Ensuite, on définit une macro spéciale permettant de checker la valeur de retour d'une expression, et d'afficher une erreur en indiquant d'où elle provient avec, puis ensuite de quitter le programme avec `EXIT_FAILURE`. Le nom est inspiré

de la méthode `.unwrap()` en Rust.

```
#define UNWRAP(expression)   \
    if ((expression) < 0)    \
    {                        \
        perror(#expression); \
        exit(EXIT_FAILURE);  \
    }
```

Ces deux enums, ainsi que la fonction `style`, permettent de mettre des couleurs et du formattage au texte du terminal, en utilisant le standard des séquences d'échappement ANSI

```
enum style_tag
{
    RESET,
    BOLD,
    DIM,
    ITALIC,
    UNDERLINED,
    SLOW_BLINK,
    FAST_BLINK,
    INVERTED,
    HIDDEN,
    CROSSED_OUT,
};

enum color_tag
{
    BLACK,
    RED,
    GREEN,
    YELLOW,
    BLUE,
    MAGENTA,
    CYAN,
    WHITE,
};

void style(enum style_tag style_tag, enum color_tag color_tag)
{
    // TRACE("set style style=%d color=%d", style_tag, color_tag);
    printf("\033[3%dm", color_tag);
    printf("\033[%dm", style_tag);
}
```

On a ensuite une méthode `TRACE` permettant d'afficher des lignes de debug quand trace est activé (grâce à la commande `trace`).

```
static bool tracing = false;
```

L'activation ou non est contrôlée par cette variable globale.

```
void TRACE(const char *format, ...)
{
    if (tracing)
    {
        printf("\033[0m\033[2m[trace] ");
        va_list args;
        va_start(args, format);
        vprintf(format, args);
        va_end(args);
        printf("\033[0m\n");
    }
}
```

Enfin, la fonction suivante permet de supporter le raccourci `~` dans les chemins, qui sera utile pour l'implémentation de `cd`.

```
char *expand_home_prefix(char *path)
{
    if (path[0] == '~')
    {
        char *home = getenv("HOME");
        char *new_path = malloc(strlen(home) + strlen(path) + 1);
        strcpy(new_path, home);
        strcat(new_path, path + 1);
        return new_path;
    }

    return path;
}
```

5. Gestion des jobs

Ensuite, il nous faut un moyen de stocker les jobs en cours.
On stocke leur état avec un booléen `running`, et leur id interne au shell correspond simplement à l'indice du job dans l'array.

```
struct job
{
    pid_t pid;
    bool running;
    char *command;
};

static struct job jobs[999];
```

`streq` est une simple fonction utilitaire pour tester l'égalité entre deux chaînes de caractères.

```
bool streq(char *a, char *b)
{
    return strcmp(a, b) == 0;
}
```

Les fonctions suivantes sont responsables de la gestion des jobs: leur ajout, suppression, trouver leur index par leur PID, et récupérer le dernier job. Un job est considéré comme existant si sa commande n'est pas `NULL`.
On pense donc bien à mettre `.command` à `NULL` quand on supprime un job.

```c
int last_job_index()
{
    TRACE("get last job index");
    int i = 0;
    while (jobs[i].command != NULL)
    {
        i++;
    }
    TRACE("last job index is %d", i - 1);
    return i - 1;
}

int job_index_by_pid(int pid)
{
    TRACE("get job index by pid %d", pid);
    int i = 0;
    while (jobs[i].command != NULL)
    {
        if (jobs[i].pid == pid)
        {
            TRACE("job index is %d", i);
            return i;
        }
        i++;
    }
    TRACE("job index not found");
    return -1;
}

// returns the last job index if NULL is given
int get_job_index_from_arg(char *arg)
{
    if (arg == NULL)
    {
        return last_job_index();
    }

    return atoi(arg) - 1;
}

void add_job(int pid, char ***commandline)
{
    int newJobIndex = last_job_index() + 1;
    jobs[newJobIndex].pid = pid;
    jobs[newJobIndex].running = false;
    jobs[newJobIndex].command = malloc(sizeof(char) * (strlen(commandline[0][0]) + 1));
    strcpy(jobs[newJobIndex].command, commandline[0][0]);
    TRACE("added jobs[%d] (%d) %s", newJobIndex, pid, jobs[newJobIndex].command);
}

int remove_job(int pid)
{
    TRACE("remove job %d", pid);
    int i = job_index_by_pid(pid);
    if (i == -1)
    {
        TRACE("job %d not found", pid);
        return 0;
    }

    free(jobs[i].command);
    jobs[i].command = NULL;
    for (int j = i + 1; j <= last_job_index(); j++)
    {
        jobs[j - 1] = jobs[j];
    }
    return 0;
}
```

6. Prompt

Une partie essentielle d'un shell, c'est ce qui apparaît avant l'entrée utilisateur quand le shell attend une commande de lui. C'est l'occasion d'afficher des informations utiles comme le répertoire de travail actuel, ou encore l'état d'un repo git si on se trouve à l'intérieur d'un, le niveau de batterie, etc. Il existe même des logiciels dédiés qui ne s'occupe que de

la partie *affichage du prompt*, tel que Starship

```
static bool got_signal;
```

Ce booléen global permet de faire la différence entre un arrêt de `readcmd()` par un signal reçu et une erreur lors de la lecture de la ligne de commande, car `readcmd()` renvoie `-1` dans les deux cas.

La fonction `prompt()` est responsable de l'affichage du prompt, et de la lecture de la ligne de commande tapée par l'utilisateur.

Elle est donc également responsable de traiter les éventuelles erreurs lors de `readcmd()`, et d'attendre que l'on ne soit plus en train de traiter un signal avant de s'afficher.

On a aussi décider de gérer la commande `exit` ici, puisque son traitement est le même que dans le cas ou `readcmd()` renvoie `NULL`.

```
struct cmdline *prompt()
{
    struct cmdline *commandline;
    do
    {
        do
        {
            char working_directory[9999];
            getcwd(working_directory, 9999);
            style(ITALIC, MAGENTA);
            printf("\n\n%s", working_directory);
            style(RESET, BLACK);
            printf("\n🐚 ");
            style(BOLD, CYAN);
            fflush(stdin);

            got_signal = false;
            commandline = readcmd();
            style(RESET, BLACK);
        } while (got_signal); // attendre de ne plus être en train de traiter un signal

        if (commandline == NULL)
        {
            return NULL;
        }

        if (commandline->err)
        {
            printf("error: %s\n", commandline->err);
            return NULL;
        }
    } while (*(commandline->seq) == NULL); // si on appuie sur rentrer sans rien rentrer, un nouveau prompt
apparaît.

    if (streq(**(commandline->seq), "exit"))
    {
        return NULL;
    }

    return commandline;
}
```

7. Gestion du processus en premier plan

On stocke tout d'abord le PID du processus en premier plan dans une variable globale.

```
static int current_pid = 0;
```

Ensuite, il nous faut une fonction pour envoyer un signal au processus en premier plan. On a des cas particulier, avec `SIGTSTP` qui devient `SIGSTOP` et `SIGINT` qui devient `SIGTERM`.

La raison pour ce changement est que `SIGTSTP` pourrait être ignoré par le processus auquel on envoie le signal, mais `SIGSTOP` ne peut pas l'être.

De la même façon, `SIGINT` est la version "envoyée par le terminal" de `SIGTERM`. `SIGINT` peut être ignoré, mais `SIGTERM` ne peut pas l'être.

L'utilisation du `-` dans le premier argument de `kill` signifie que l'on envoie le signal à tout les processus du *groupe*

du PID donné.

On travaille sur des groupes pour envoyer le signal aux potentiels fils du processus, même lorsque celui-ci redéfinit le(s) handler(s) du signal que l'on envoie.

```
void send_signal_to_current_process(int signal)
{
    if (signal == SIGTSTP)
    {
        kill(-current_pid, SIGSTOP);
    }
    else if (signal == SIGINT)
    {
        kill(-current_pid, SIGTERM);
    }
    else
    {
        kill(-current_pid, signal);
    }

    got_signal = true;
}
```

Cette fonction change le processus en premier plan. Cela implique de:

1. Mettre à jour notre variable globale `current_pid`
2. Signaler ce changement au système via `tcsetpgrp` (ce qui met à jour le groupe de processus en premier plan)
3. Attendre que l'on reçoive un signal
4. De nouveau `tcstepgrp`, pour notifier au système que le shell est de nouveau en premier plan. Cela permet au shell d'accéder de nouveau à l'entrée et aux sorties standards.

```
void switch_current_process(int pid)
{
    current_pid = pid;
    tcsetpgrp(STDERR_FILENO, pid);
    do
    {
        pause();
    } while (current_pid > 0);
    tcsetpgrp(STDERR_FILENO, getpgrp());
}
```

8. Gestion des fils

Gérer les fils implique principalement deux choses:

- Les démarrer (`start_child`)
- Attendre qu'ils se terminent et réagir en fonction (`child_handler_action`)

La fonction suivante permet de démarrer les fils.

```
int start_child(int *group_pid, int child_stdin, int child_stdout, char **args)
{
    int pid;
    struct sigaction child_handler;
    sigemptyset(&child_handler.sa_mask);
    child_handler.sa_flags = 0;
```

On a choisit de créer les pipes de proche en proche: un enfant est démarré avec une entrée et une sortie souhaitée. Si la sortie n'est pas `stdout`, on crée un pipe et on écrit dedans. On passe ensuite le descripteur de lecture de ce pipe comme valeur de retour, afin que le père puisse passer ce descripteur au fils suivant.

Le fait d'avoir ouvert un pipe pour le fils prochain est stocké dans le booléen `outputs_to_pipe`.

Le premier fils a donc `stdin` comme entrée, et le dernier a `stdout` comme sortie.

```
    int pipe_to_next_child[2];
    bool outputs_to_pipe = child_stdout < 0;
    if (outputs_to_pipe)
    {
        style(RESET, BLACK);
        UNWRAP(pipe(pipe_to_next_child));
        child_stdout = pipe_to_next_child[1];
    }
```

On crée le fils ici, après avoir ouvert le pipe (si nécéssaire).

```
    UNWRAP(pid = fork());

    if (pid == 0)
    {
```

On ferme le côté lecture du pipe que l'on va donner au prochain fils, car l'on n'a pas besoin de le lire depuis ce fils-ci.

```
        if (outputs_to_pipe)
        {
            UNWRAP(close(pipe_to_next_child[0]));
            child_stdout = pipe_to_next_child[1];
        }
```

On change le groupe du fils pour le rattacher au groupe de toute la pipeline.

```
        UNWRAP(setpgid(getpid(), *group_pid));
```

On change les handlers de `SIGTSTP`, `SIGINT`, `SIGTTOU` et `SIGTTIN` pour les remettre à leur handler par défaut.

```
        child_handler.sa_handler = SIG_DFL;
        sigaction(SIGTSTP, &child_handler, NULL);
        sigaction(SIGINT, &child_handler, NULL);
        sigaction(SIGTTOU, &child_handler, NULL);
        sigaction(SIGTTIN, &child_handler, NULL);
```

On change les entrées et sorties du fils pour qu'il utilise les descripteurs de fichiers que l'on a donné en paramètre (que ce soit des pipes ou stdin/out).

```
        UNWRAP(dup2(child_stdin, STDIN_FILENO));
        UNWRAP(dup2(child_stdout, STDOUT_FILENO));
```

Enfin, on exécute la commande.

```
        execvp(args[0], args);
        perror("execvp");
        exit(EXIT_FAILURE);
    }
```

Depuis le père, on met à jour le PID du groupe passé en argument lorsque celui-ci n'était pas connu.
On ferme ensuite les descripteurs:

- l'entrée du fils (sauf si celle-ci est l'entrée standard)
- le pipe ou le fichier (si il y a redirection dans un fichier), sauf si la sortie est la sortie standard.

Enfin, on renvoie le descripteur d'entrée du prochain fils, qui est soit le pipe (si on a écrit dans un pipe), soit l'entrée standard.

```
    if (*group_pid == 0)
    {
        *group_pid = pid;
    }

    if (child_stdin != STDIN_FILENO)
    {
        UNWRAP(close(child_stdin));
    }

    if (outputs_to_pipe)
    {
        UNWRAP(close(pipe_to_next_child[1]));
        return pipe_to_next_child[0];
    }
    else if (child_stdout != STDOUT_FILENO)
    {
        UNWRAP(close(child_stdout));
    }

    return STDIN_FILENO;
}
```

La fonction suivante est executée quand un fils est créé depuis notre shell, via l'assignation de la fonction comme handler de `SIGCHLD`.

On attend que le fils se termine avec `waitpid`, et on agit en fonction du résultat:

- Si il a été stoppé (avec `WIFSTOPPED`), on met à jour le statut du job;
- Si il a été terminé, on affiche le code de retour, et on supprime le job de la liste des jobs.

De plus, si le fils était le processus en premier plan, il faut mettre à jour `current_pid` pour le remettre à `0` (signifiant que le shell est en premier plan).

Enfin, si `waitpid` a échoué, et que l'erreur n'est pas `ECHILD` (signifiant qu'il n'y a pas de fils à attendre), on affiche l'erreur.

```
void child_handler_action()
{
    int pid;
    int status;
    got_signal = true;

    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        if (WIFSTOPPED(status))
        {
            jobs[job_index_by_pid(pid)].running = false;
            TRACE("job %d stopped", pid);
        }
        else
        {
            UNWRAP(remove_job(pid));
            TRACE("job pid=%d removed", pid);
            if (WEXITSTATUS(status) == 0) {
                // printf("✅ 0");
            } else {
                printf("❌ %d", WEXITSTATUS(status));
            }
        }

        if (pid == current_pid)
        {
            current_pid = 0;
        }
    }

    if (pid < 0 && errno != ECHILD)
    {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }
}
```

9. Boucle principale

Le point d'entrée du shell. La boucle "infinie" qui va demander à l'utilisateur d'entrer une commande, et l'exécuter.

```
int main()
{
```

On initialise les handlers de signaux:

- `SIGCHLD` est handle par `child_handler_action`
- `SIGTSTP`, `SIGINT`, `SIGTTOU` et `SIGTTIN` sont ignorés.

```
    struct sigaction signals_handler;
    sigemptyset(&signals_handler.sa_mask);
    signals_handler.sa_flags = 0;

    signals_handler.sa_handler = child_handler_action;
    sigaction(SIGCHLD, &signals_handler, NULL);

    signals_handler.sa_handler = SIG_IGN;
    sigaction(SIGTSTP, &signals_handler, NULL);
    sigaction(SIGINT, &signals_handler, NULL);
    sigaction(SIGTTOU, &signals_handler, NULL);
    sigaction(SIGTTIN, &signals_handler, NULL);
```

On commence la boucle principale. On stocke le résultat du prompt (c'est-à-dire la ligne de commande entrée par l'utilisateur) dans `commandline`.

```
    struct cmdline *commandline;

    while ((commandline = prompt()))
    {
```

Malheureusement, réinitialiser le texte à la couleur par défaut ne fonctionne pas sans faire de retour à la ligne. Mais, juste après le prompt, je trouve un retour à la ligne vraiment pas esthétique. J'ai essayé un `fflush(stdin)`, qui n'a pas non plus réglé le problème.

```
        style(RESET, BLACK);
```

On commence par gérer les commandes "spéciales" que le shell va interpréter directement, sans lancer de fils: `cd`, `lj`, `sj`, `susp`, `trace`, `fg` `et` `bg`.

```
        char **first_command_args = *(commandline->seq);
        char *first_command = first_command_args[0];

        if (streq(first_command, "cd"))
        {
            if (chdir(expand_home_prefix(first_command_args[1])) < 0)
            {
                perror("chdir");
            }
            continue;
        }
        if (streq(first_command, "lj"))
        {
            TRACE("listing jobs");
            for (int i = 0; i <= last_job_index(); i++)
            {
                printf("%s %d (%d) %s\n", jobs[i].running ? "🏃" : "🛑", i + 1, jobs[i].pid, jobs[i].command);
            }
            continue;
        }

        if (streq(first_command, "sj"))
        {
            int job_index = get_job_index_from_arg(first_command_args[1]);
            TRACE("stopping job %d, pid=", job_index, jobs[job_index].pid);
            kill(-jobs[job_index].pid, SIGSTOP);
            continue;
        }

        if (streq(first_command, "susp"))
        {
            kill(getpid(), SIGSTOP);
        }

        if (streq(first_command, "trace"))
        {
            tracing = !tracing;
            style(ITALIC, YELLOW);
            printf("trace is %s.\n", tracing ? "on" : "off");
            style(RESET, BLACK);
            continue;
        }
```

Les commandes `fg` et `bg` sont les mêmes: on envoie dans les deux cas un `SIGCONT` aux processus dans le groupe du job choisi. La différence est que `fg` va changer le processus en premier plan.

```
        if (streq(first_command, "fg") || streq(first_command, "bg"))
        {
            int job_index = get_job_index_from_arg(first_command_args[1]);
            int pid = jobs[job_index].pid;

            kill(-pid, SIGCONT);

            if (streq(first_command, "fg"))
            {
                jobs[job_index].running = true;
                switch_current_process(pid);
            }

            continue;
        }
```

On prépare ici la création du fils. On stocke l'entrée du fils actuel, qui est au départ `stdin`, et la sortie de la pipeline entière, qui sera `stdout`, sauf si redirection.

```
        int current_child_stdin = STDIN_FILENO;
        int pipeline_stdout = STDOUT_FILENO;
        int group_pid = 0;
```

On gère la présence de redirections, en mettant à jour `current_child_stdin` et `pipeline_stdout`.

```
        if (commandline->in)
        {
            current_child_stdin = open(commandline->in, O_RDONLY);
            if (current_child_stdin < 0)
            {
                perror("open stdin redirector");
                continue;
            }
        }

        if (commandline->out)
        {
            pipeline_stdout = open(commandline->out, O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (pipeline_stdout < 0)
            {
                perror("open stdout redirector");
                continue;
            }
        }
```

On commence une boucle, qui va itérer sur chaque commande de la pipeline, en lançant un fils pour chaque commande. Seul le dernier fils aura `pipeline_stdout` comme sortie, les autres fils n'ont pas de sortie

```
        char ***commands_left = commandline->seq;
        while (*commands_left)
        {
            bool is_last_command = *(commands_left + 1) == NULL;
            current_child_stdin = start_child(&group_pid, current_child_stdin, is_last_command ? pipeline_stdout :
-1, *commands_left);
            commands_left++; // XXX c'est vraiment dégueulasse d'itérer comme ça.
        }
```

On ajoute le PID du groupe à la liste des jobs, et on passe le processus en premier plan si celui-ci n'est pas mis en arrière-plan.

```
        add_job(group_pid, commandline->seq);

        if (!commandline->backgrounded)
        {
            switch_current_process(group_pid);
        }
```

Enfin, quand on sort de la boucle principale (par une valeur de retour égale à `NULL` de la fonction `prompt`), on tue tout les jobs avec un `SIGKILL`, pour s'assurer de ne pas laisser des processus toujours en train de tourner, qu'on ne pourrait pas tuer avec un `SIGINT` une fois le shell fermé.

```
    }

    // kill everyone
    for (int i = 0; i <= last_job_index(); i++)
    {
        kill(-jobs[i].pid, SIGKILL);
    }

    return EXIT_SUCCESS;
}
```