

Compte Rendu : Implémentation du Minishell

BENABOUD Mehdi

Résumé

Ce rapport documente les fonctionnalités implémentées dans le fichier `minishell.c`, en conformité avec les étapes des Travaux Pratiques. Il présente les choix de conception et la logique sous-jacente à chaque aspect développé, en intégrant des extraits de code pour illustrer ces implémentations.

Table des matières

1	Fonctionnalités Implémentées	1
1.1	Gestion des Processus Fils (TP1 : Processus)	1
1.1.1	Lancement de commandes externes (<code>fork()</code> et <code>execvp()</code>)	2
1.1.2	Gestion des processus en avant-plan et en arrière-plan	2
1.2	Gestion des Signaux (TP2 & TP3 : Signaux)	3
1.2.1	Gestionnaire de SIGCHLD (<code>traitement()</code> et <code>sigaction</code>)	3
1.2.2	Masquage de SIGINT et SIGTSTP dans le père (Étape 11.3)	4
1.2.3	Démasquage dans le fils	4
1.3	Redirection des Entrées/Sorties (TP4 : Fichiers)	5
1.3.1	Redirection d'entrée (<code>i</code>)	5
1.3.2	Redirection de sortie (<code>o</code>)	5
1.4	Commandes Internes <code>cd</code> et <code>dir</code> (TP4 : Fichiers)	6
1.4.1	Commande <code>cd</code> (Étape 14)	6
1.4.2	Commande <code>dir</code> (Étape 15)	7
1.5	Tubes (Pipelines) (TP5 : Tubes)	7
1.5.1	Étape 16 (Tubes simples) & Étape 17 (Pipelines)	7
2	Méthodologie de Test	8
3	Conclusion	9

1 Fonctionnalités Implémentées

Les sections suivantes décrivent les ajouts et modifications réalisés pour étendre les capacités du minishell au-delà de sa structure de base.

1.1 Gestion des Processus Fils (TP1 : Processus)

L'implémentation inclut la capacité du minishell à lancer des commandes externes en créant de nouveaux processus, ainsi qu'à gérer leur exécution en avant-plan ou en arrière-plan.

1.1.1 Lancement de commandes externes (fork() et execvp())

Pour exécuter une commande externe (toute commande qui n'est pas `exit`, `cd` ou `dir`), la création d'un nouveau processus fils est réalisée à l'aide de `fork()`. Ce choix est fondamental en environnement Unix, car il permet de dupliquer le processus du minishell, créant un environnement isolé pour la commande à exécuter.

Dans le processus fils (où `pid_commande == 0`), la primitive `execvp(cmd[0], cmd)` est utilisée. `execvp` est privilégiée car elle recherche l'exécutable dans le PATH système, simplifiant l'appel de commandes standard. Si `execvp` échoue (par exemple, si la commande n'existe pas), le fils affiche une erreur via `perror("erreur execvp")` et se termine avec `exit(EXIT_FAILURE)`, évitant ainsi que le fils ne continue l'exécution du code du minishell par erreur.

```
1 // Extrait de la boucle principale, dans le bloc 'else' après la lecture
  de commande
2 // ...
3 pid_t pid_commande = fork(); // Création d'un nouveau processus fils
4 if (pid_commande == -1) {
5     perror("erreur fork"); // Gestion de l'erreur du fork
6 } else if (pid_commande == 0) {
7     // Code exécuté par le processus fils
8     // ... (voir ci-dessous pour le masquage des signaux et setpgrp)
9     execvp(cmd[0], cmd); // Exécution de la commande
10    perror("erreur execvp"); // Si execvp retourne, c'est une erreur
11    exit(EXIT_FAILURE); // Le fils se termine en cas d'erreur
12 } else {
13     // Code exécuté par le processus père
14     // ... (voir ci-dessous pour la gestion avant/arrière-plan)
15 }
16 // ...
```

Listing 1 – Extrait de code: Lancement de commandes externes

1.1.2 Gestion des processus en avant-plan et en arrière-plan

Commandes en avant-plan : Si la commande ne se termine pas par `&` (indiqué par `commande->backgrounded == NULL`), le processus père utilise `pause()`. Ce choix permet au minishell de se mettre en attente d'un signal (typiquement `SIGCHLD` lorsque le fils en avant-plan se termine) sans consommer activement des ressources CPU, assurant que le prompt ne réapparaisse qu'après la fin de la commande.

```
1 // Extrait du bloc du processus père après le fork
2 // ...
3 if (commande->backgrounded == NULL) {
4     // Commande en avant-plan : le père attend un signal (SIGCHLD)
5     pause(); // Met le père en attente jusqu'à la réception d'un signal
6 }
7 // ...
```

Listing 2 – Extrait de code: Gestion des commandes en avant-plan

Commandes en arrière-plan : Si la commande se termine par `&` (`commande->backgrounded != NULL`), le processus père ne bloque pas. Il ne fait aucun appel `waitpid` ou `pause()` pour ce fils, lui permettant de revenir immédiatement au prompt et de lancer d'autres commandes pendant que le processus en arrière-plan continue son exécution de manière concurrente.

Détachement des processus fils en arrière-plan (Étape 12) : Pour les commandes lancées en arrière-plan (`commande->backgrounded != NULL`), la primitive `setpgrp()` est utilisée dans le processus fils, juste avant `execvp`. Ce choix permet de placer le processus fils dans un nouveau groupe de processus, l'isolant ainsi des signaux de terminal (`SIGINT`, `SIGTSTP`) envoyés au groupe de processus du minishell, assurant que seuls les processus en avant-plan sont affectés.

```

1 // Extrait du bloc du processus fils, avant execvp
2 // ...
3 if (commande->backgrounded != NULL) {
4     // Mettre les processus en arri re-plan dans un nouveau groupe de
    // processus
5     if (setpgrp() == -1) {
6         perror("erreur setpgrp");
7         exit(EXIT_FAILURE);
8     }
9 }
10 // ...

```

Listing 3 – Extrait de code: Détachement des processus en arrière-plan

1.2 Gestion des Signaux (TP2 & TP3 : Signaux)

Des mécanismes ont été mis en place pour que le minishell réagisse de manière appropriée aux signaux système, améliorant sa robustesse et son comportement.

1.2.1 Gestionnaire de SIGCHLD (`traitement()` et `sigaction`)

Une fonction `traitement(int sig)` a été définie et est associée au signal `SIGCHLD` via `sigaction`. Le choix de `sigaction` offre un contrôle plus précis que `signal()`, notamment l'utilisation du drapeau `SA_RESTART` pour redémarrer les appels système interrompus. Dans le gestionnaire `traitement`, une boucle `while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED | WCONTINUED)) > 0)` est utilisée. Cette boucle permet de récupérer l'état de tous les processus fils ayant changé d'état (terminés, suspendus, repris) sans bloquer le minishell. Les drapeaux `WNOHANG`, `WUNTRACED`, et `WCONTINUED` sont essentiels pour un comportement non bloquant et pour détecter les suspensions et reprises. Les macros `WIFEXITED`, `WIFSIGNALED`, `WIFSTOPPED`, et `WIFCONTINUED` sont utilisées pour afficher des messages informatifs sur l'état des processus fils.

```

1 // D finition de la fonction de traitement de signal
2 void traitement(int sig) {
3     pid_t pid;
4     int status;
5     // Boucle pour r cup rer l' tat de tous les fils ayant chang d'
    // tat
6     while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED | WCONTINUED))
    > 0) {
7         if (WIFEXITED(status)) {
8             printf("Processus %d termin avec code %d\n", pid,
    WEXITSTATUS(status));
9         } else if (WIFSIGNALED(status)) {
10            printf("Processus %d termin par le signal %d\n", pid,
    WTERMSIG(status));
11        } else if (WIFSTOPPED(status)) {
12            printf("Processus %d suspendu par le signal %d\n", pid,
    WSTOPSIG(status));

```

```

13     } else if (WIFCONTINUED(status)) {
14         printf("Processus %d repris\n", pid);
15     }
16 }
17 // printf("signal reçu : %d\n", sig); // D commenter pour debug
18 }
19
20 // Extrait de la fonction main, au d but
21 // ...
22 struct sigaction action;
23 action.sa_handler = traitement; // Associe la fonction traitement au
    signal
24 action.sa_flags = SA_RESTART; // Red marre les appels syst me
    interrompus
25 sigemptyset(&action.sa_mask); // Aucun signal n'est bloqu pendant l'
    ex cution du handler
26 if (sigaction(SIGCHLD, &action, NULL) == -1) {
27     perror("erreur sigaction SIGCHLD");
28     exit(EXIT_FAILURE);
29 }
30 // ...

```

Listing 4 – Extrait de code: Gestionnaire SIGCHLD

1.2.2 Masquage de SIGINT et SIGTSTP dans le père (Étape 11.3)

Pour empêcher le minishell d'être terminé par Ctrl+C (SIGINT) ou suspendu par Ctrl+Z (SIGTSTP), le masquage de ces signaux a été implémenté dans le processus père. Un ensemble de signaux (`sigset_t masque`) est créé, SIGINT et SIGTSTP y sont ajoutés, puis `sigprocmask(SIG_BLOCK, &masque, NULL)` est appelé. Ce choix garantit que le minishell reste actif et fonctionnel même si ces signaux sont envoyés depuis le terminal.

```

1 // Extrait de la fonction main, au d but
2 // ...
3 sigset_t masque;
4 sigemptyset(&masque); // Initialise l'ensemble de signaux vide
5 sigaddset(&masque, SIGINT); // Ajoute SIGINT (Ctrl+C) l'ensemble
6 sigaddset(&masque, SIGTSTP); // Ajoute SIGTSTP (Ctrl+Z) l'ensemble
7
8 if (sigprocmask(SIG_BLOCK, &masque, NULL) == -1) {
9     perror("erreur de sigprocmask");
10    exit(EXIT_FAILURE);
11 }
12 // ...

```

Listing 5 – Extrait de code: Masquage de signaux dans le père

1.2.3 Démasquage dans le fils

Dans le processus fils, juste avant `execvp`, le masque des signaux est vidé (`sigemptyset(&masque_fils);`) et appliqué avec `sigprocmask(SIG_SETMASK, &masque_fils, NULL)`. Ceci est crucial pour que les processus fils reçoivent normalement SIGINT et SIGTSTP et puissent être terminés ou suspendus comme attendu.

```

1 // Extrait du bloc du processus fils, avant execvp
2 // ...
3 sigset_t masque_fils;

```

```

4 sigemptyset(&masque_fils); // Initialise l'ensemble de signaux du fils
  vide
5 if (sigprocmask(SIG_SETMASK, &masque_fils, NULL) == -1) {
6     perror("erreur d masquage signaux dans le fils");
7     exit(EXIT_FAILURE);
8 }
9 // ...

```

Listing 6 – Extrait de code: Démasquage de signaux dans le fils

1.3 Redirection des Entrées/Sorties (TP4 : Fichiers)

La capacité de rediriger les flux d'entrée et de sortie des commandes vers des fichiers a été ajoutée. Ces opérations sont effectuées dans le processus fils avant l'appel à `execvp`.

1.3.1 Redirection d'entrée (i)

Un test est effectué sur `if (commande->in != NULL)`. Si un fichier d'entrée est spécifié, il est ouvert en lecture seule (`O_RDONLY`). La primitive `dup2(fd_source, 0)` est utilisée pour dupliquer le descripteur du fichier source sur le descripteur 0 (entrée standard). Le descripteur original du fichier est ensuite fermé. Ce mécanisme standard permet à la commande de lire ses données depuis le fichier au lieu du clavier.

```

1 // Extrait du bloc du processus fils, avant execvp
2 // ...
3 if (commande->in != NULL) {
4     int fd_source;
5     if ((fd_source = open(commande->in, O_RDONLY)) == -1) {
6         perror("erreur ouverture fichier source");
7         exit(EXIT_FAILURE);
8     }
9     if (dup2(fd_source, 0) == -1) { // 0 est STDIN_FILENO
10        perror("erreur redirection de l'entr e standard");
11        exit(EXIT_FAILURE);
12    }
13    if (close(fd_source) == -1) {
14        perror("erreur fermeture fichier source");
15        exit(EXIT_FAILURE);
16    }
17 }
18 // ...

```

Listing 7 – Extrait de code: Redirection d'entrée

1.3.2 Redirection de sortie (o)

Un test est effectué sur `if (commande->out != NULL)`. Si un fichier de sortie est spécifié, il est ouvert en écriture (`O_WRONLY`), en le créant s'il n'existe pas (`O_CREAT`) et en le tronquant s'il existe déjà (`O_TRUNC`). Les permissions `0644` sont appliquées. `dup2(fd_destination, 1)` est utilisée pour dupliquer le descripteur du fichier de destination sur le descripteur 1 (sortie standard). Le descripteur original est ensuite fermé. Ce choix assure que la sortie de la commande est écrite dans le fichier spécifié.

```

1 // Extrait du bloc du processus fils, avant execvp
2 // ...
3 if (commande->out != NULL) {

```

```

4     int fd_destination;
5     // O_WRONLY:  écriture seule, O_CREAT: créer si n'existe pas, O_TRUNC
: tronquer si existe
6     if ((fd_destination = open(commande->out, O_WRONLY | O_CREAT | O_TRUNC
, 0644)) == -1) {
7         perror("erreur ouverture fichier destination");
8         exit(EXIT_FAILURE);
9     }
10    if (dup2(fd_destination, 1) == -1) { // 1 est STDOUT_FILENO
11        perror("erreur redirection de la sortie standard");
12        exit(EXIT_FAILURE);
13    }
14    if (close(fd_destination) == -1) {
15        perror("erreur fermeture fichier destination");
16        exit(EXIT_FAILURE);
17    }
18 }
19 // ...

```

Listing 8 – Extrait de code: Redirection de sortie

1.4 Commandes Internes cd et dir (TP4 : Fichiers)

Deux commandes internes, exécutées directement par le minishell sans créer de processus fils, ont été implémentées.

1.4.1 Commande cd (Étape 14)

La commande est identifiée par `strcmp(cmd[0], "cd") == 0`. La primitive `chdir()` est utilisée pour changer le répertoire de travail du minishell lui-même. Ce choix est crucial car `chdir()` doit affecter le processus du shell pour que les commandes futures soient lancées depuis le nouveau répertoire. Si aucun argument n'est fourni (`cmd[1]` est NULL), `getenv("HOME")` est utilisé pour changer le répertoire vers le répertoire personnel de l'utilisateur.

```

1 // Extrait de la boucle principale, dans le bloc de traitement des
  commandes
2 // ...
3 if (strcmp(cmd[0], "cd") == 0) {
4     const char *path = cmd[1] ? cmd[1] : getenv("HOME"); // Utilise l'
argument ou HOME si absent
5     if (chdir(path) == -1) { // Tente de changer le r pertoire de travail
6         perror("erreur de chdir"); // Affiche une erreur si le changement
choue
7     }
8     indexseq++; // Passe la commande suivante dans un pipeline (si
applicable)
9     continue; // Retourne au d but de la boucle principale pour un
nouveau prompt
10 }
11 // ...

```

Listing 9 – Extrait de code: Commande cd

1.4.2 Commande dir (Étape 15)

La commande est identifiée par `strcmp(cmd[0], "dir") == 0`. Une fonction `executer_dir` a été créée, utilisant les primitives `opendir()`, `readdir()`, et `closedir()` pour lister le contenu d'un répertoire. Le répertoire cible est l'argument de la commande ou le répertoire courant (`.`) par défaut. L'implémentation en tant que commande interne permet une gestion directe des répertoires sans dépendre d'un exécutable externe.

```
1 // Définition de la fonction executer_dir
2 #include <dirent.h> // Nécessaire pour DIR, dirent, opendir, readdir,
   closedir
3
4 void executer_dir(const char *chemin) {
5     const char *target = chemin ? chemin : "."; // Définit le répertoire
   cible (courant par défaut)
6     DIR *d = opendir(target); // Ouvre le flux de répertoire
7     if (!d) { // Gère l'erreur d'ouverture (ex: répertoire inexistant)
8         perror("erreur ouverture répertoire");
9         return;
10    }
11    struct dirent *entree;
12    while ((entree = readdir(d)) != NULL) { // Lit chaque entrée du
   répertoire
13        printf("%s\n", entree->d_name); // Affiche le nom de l'entrée
14    }
15    if (closedir(d) == -1) { // Ferme le flux de répertoire et gère les
   erreurs
16        perror("erreur fermeture répertoire");
17    }
18 }
19
20 // Extrait de la boucle principale, dans le bloc de traitement des
   commandes
21 // ...
22 else if (strcmp(cmd[0], "dir") == 0) {
23     const char *chemin = cmd[1] ? cmd[1] : "."; // Utilise l'argument ou
   le répertoire courant
24     executer_dir(chemin); // Appelle la fonction d'exécution de dir
25     indexseq++;
26     continue;
27 }
28 // ...
```

Listing 10 – Extrait de code: Commande dir

1.5 Tubes (Pipelines) (TP5 : Tubes)

Le minishell supporte l'enchaînement de commandes via des tubes, permettant la communication inter-processus où la sortie d'une commande devient l'entrée de la suivante. Cette fonctionnalité est implémentée dans la fonction `executer_pipeline`.

1.5.1 Étape 16 (Tubes simples) & Étape 17 (Pipelines)

La fonction `executer_pipeline` gère les pipelines de n'importe quelle longueur. Le nombre de commandes est déterminé en parcourant `commande->seq`. Pour chaque commande (sauf la dernière), un nouveau tube est créé avec `pipe(tube_suiv)`. Un processus fils est créé pour chaque commande avec `fork()`. Dans le processus fils, les descripteurs

de fichiers sont redirigés avec `dup2()` pour connecter l'entrée/sortie du fils aux extrémités appropriées des tubes (`tube_prec[0]` pour l'entrée, `tube_suiv[1]` pour la sortie). Les descripteurs de tube inutilisés par le fils sont ensuite fermés. Enfin, `execvp` est appelé pour exécuter la commande. Dans le processus père, les descripteurs de tube inutilisés par le père sont fermés. Les descripteurs du tube actuel (`tube_suiv`) deviennent les "précédents" (`tube_prec`) pour la prochaine itération, préparant la connexion de la commande suivante. Après avoir créé tous les processus fils du pipeline, le processus père attend la terminaison de chacun d'eux avec une boucle `for` appelant `wait(NULL)`. Ce choix assure que le shell ne continue pas avant que l'ensemble du pipeline ne soit terminé.

2 Méthodologie de Test

La validation de ces implémentations a suivi une approche incrémentale. Chaque fonctionnalité a été testée pour s'assurer de son bon fonctionnement.

— Tests de Processus (TP1) :

- **Lancement de commandes externes** : Vérification que des commandes comme `ls`, `pwd`, `echo Hello` s'exécutent et affichent leur sortie.
- **Commandes en arrière-plan** : Lancement de `sleep X &` et vérification que le prompt revient immédiatement, et que le processus `sleep` est visible en arrière-plan (par exemple, via `ps -f`).
- **Commandes en avant-plan** : Lancement de `sleep X` et vérification que le minishell attend la fin de `sleep` avant de réafficher le prompt.

— Tests de Signaux (TP2 & TP3) :

- **SIGCHLD** : Lancement de processus en arrière-plan et vérification que le minishell affiche correctement les messages de terminaison, suspension ou reprise des processus fils.
- **SIGINT (Ctrl+C)** : Lancement d'une commande en avant-plan (`sleep X`), puis envoi de `Ctrl+C`. Vérification que la commande est terminée mais que le minishell reste actif.
- **SIGTSTP (Ctrl+Z)** : Lancement d'une commande en avant-plan (`sleep X`), puis envoi de `Ctrl+Z`. Vérification que la commande est suspendue mais que le minishell reste actif.
- **Détachement des processus en arrière-plan** : Vérification que les processus en arrière-plan ne sont pas affectés par `Ctrl+C` ou `Ctrl+Z` envoyés au terminal.

— Tests de Redirection et Commandes Internes (TP4) :

- **Redirection d'entrée** : `cat < fichier.txt` pour s'assurer que le contenu du fichier est lu.
- **Redirection de sortie** : `echo "test" > fichier.txt` pour vérifier que la sortie est écrite dans le fichier.
- **Commande cd** : `cd /tmp` pour vérifier le changement de répertoire du minishell, et `cd` (sans argument) pour vérifier le retour au répertoire `HOME`.
- **Commande dir** : `dir .` ou `dir /chemin/quelconque` pour vérifier l'affichage correct du contenu des répertoires.

— Tests de Tubes (TP5) :

- **Tube simple** : `ls | wc -l` pour vérifier que le nombre de lignes de la sortie de `ls` est affiché.

- **Pipelines complexes** : `cat fichier.c | grep "motif" | wc -l` pour vérifier l'enchaînement de plusieurs commandes via des tubes.

```

victus@Device:/mnt/c/Users/mehdi/Downloads/minishell-mbd1523(3)/minishell$ ./minishell
> ls
Makefile minishell minishell.o readcmd.c readcmd.h readcmd.o test test.txt test_readcmd.c
Processus 2038 terminé avec code 0
signal recu: 17
ls
> pwd
/mnt/c/Users/mehdi/Downloads/minishell-mbd1523(3)/minishell
Processus 2039 terminé avec code 0
signal recu: 17
pwd
> echo Hello
Hello
Processus 2056 terminé avec code 0
signal recu: 17
echo Hello
> sleep 3
Processus 2057 terminé avec code 0
signal recu: 17
sleep 3
> sleep 10
^CProcessus 2058 tué par signal 2
signal recu: 17
sleep 10
> sleep 10
^ZProcessus 2059 stoppé (CTRL+Z ?) signal 20
signal recu: 17
sleep 10
victus@Device:/mnt/c/Users/mehdi/Downloads/minishell-mbd1523(3)/minishell$ ./minishell
> cat < fichier.txt
test terminéProcessus 2090 terminé avec code 0
signal recu: 17
cat
> echo "test" > fichier.txt
Processus 2091 terminé avec code 0
signal recu: 17
echo "test"
> cd /tmp
> pwd
/tmp
Processus 2092 terminé avec code 0
signal recu: 17
pwd

victus@Device:/mnt/c/Users/mehdi/Downloads/minishell-mbd1523(3)/minishell$ ./minishell
> dir .
..
.vscode
fichier.txt
Makefile
minishell
minishell.c
minishell.o
readcmd.c
readcmd.h
readcmd.o
test
test.txt
test_readcmd.c
> dir /
bin.usr-is-merged
..
var
dev
libx32
wslcHomPp
init
lib64
sys
boot
media
tmp
opt
home
lib.usr-is-merged
lib32

victus@Device:/mnt/c/Users/mehdi/Downloads/minishell-mbd1523(3)/minishell$ ./minishell
> ls | wc -l
11
signal recu: 17
Processus 2123 terminé avec code 0
signal recu: 17
> cat minishell.c | grep commande | wc -l
31
Processus 2125 terminé avec code 0
signal recu: 17
signal recu: 17
signal recu: 17
>

```

FIGURE 1 – Illustrations du module Test

3 Conclusion

Les implémentations réalisées dans `minishell.c` ont permis d'étendre significativement les capacités du shell de base. L'intégration de la gestion des processus fils (avant-plan/arrière-plan), un traitement robuste des signaux (`SIGCHLD`, `SIGINT`, `SIGTSTP`), la redirection des flux d'E/S, l'ajout de commandes internes essentielles comme `cd` et `dir`, ainsi que la gestion des pipelines, sont des étapes cruciales pour la construction d'un interpréteur de commandes fonctionnel et démontrent une bonne compréhension des primitives système Unix. Les tests effectués confirment la validité des fonctionnalités implémentées, posant une base solide pour les développements ultérieurs.