Systèmes et réseau - TP Shell

ANGERETTI Quentin BRIQUET Chloé

10 février 2026

Table des matières

1 Analyse du sujet

1.1 Préambule

L'objectif de ce TP est de réaliser un **mini-shell Unix** capable d'interpréter des commandes utilisateur. Avant de nous lancer dans le code, nous avons pris le temps de lire le sujet en entier pour bien cerner ce qu'on attendait de nous. En résumé, il s'agit de :

— Comprendre la **structure d'un shell** et son fonctionnement interne.
— Apprendre à manipuler les **appels système** essentiels : `fork`, `exec`, `pipe`, `dup2`, `wait`, etc.

Mais d'abord, qu'est-ce qu'un shell exactement ?

Shell

Un **shell** est un interprète de langage de commande. Il transforme une commande tapée sous forme textuelle (par exemple `ls -a`) en un ou plusieurs appels système. Dans le système Unix, des exemples de shells sont `bash`, `tcsh`, `ksh`, etc.

En d'autres termes, quand on tape `ls -a` dans un terminal, c'est le shell qui se charge de comprendre cette ligne, de trouver le programme `ls`, et de le lancer avec l'argument `-a`. Notre objectif est de reproduire ce mécanisme.

1.2 Fichiers fournis

Nous avons commencé par explorer le dossier `src/` pour comprendre ce qui nous était donné comme base de travail.

Fichiers du projet

`readcmd.h` / `readcmd.c` : Parser de commandes. Contient la fonction `readcmd()` qui analyse la ligne tapée par l'utilisateur.
`shell.c` : Programme de démonstration initial, à transformer en shell fonctionnel.
`shell.h` : Interface publique du shell (pas fourni, mais c'est vraiment pratique de la créer).
`csapp.h` / `csapp.c` : Wrappers pour les appels système (`Fork()`, `Dup2()`, `Wait()`, etc.).

Le fichier `shell.c` de départ ne faisait qu'afficher le contenu de la structure retournée par `readcmd()`. C'est à partir de cette base que nous avons construit notre shell, étape par étape.

1.3 Stratégie adoptée

Après avoir lu le sujet et examiné les fichiers, nous avons identifié un point clé : il est important de distinguer deux types de commandes. Les commandes **intégrées** (comme `cd` ou `quit`) qui doivent s'exécuter dans le shell lui-même, et les **commandes externes** (comme `ls`, `grep`) qui sont des programmes du système. Cette distinction a guidé toute notre architecture.

2 Concepts fondamentaux

Avant de plonger dans l'implémentation, nous avons dû nous approprier plusieurs concepts. Nous les présentons ici dans l'ordre où nous en avons eu besoin.

2.1 La structure `cmdline`

La première chose à comprendre est la structure de données que `readcmd()` nous renvoie. C'est elle qui contient toute l'information sur ce que l'utilisateur a tapé.

Structure `struct cmdline`

```
struct cmdline {
    char *err;   // Message d'erreur (NULL si OK)
    char *in;    // Fichier de redirection d'entree (< fichier)
    char *out;   // Fichier de redirection de sortie (> fichier)
    char ***seq; // Sequence de commandes separees par |
};
```

`err` : Si non-`NULL`, contient un message d'erreur de syntaxe.
`in` / `out` : Noms des fichiers pour les redirections `<`, `>` et `»`.
`seq` : Tableau de commandes. Chaque commande est un tableau de mots (`char **`) terminé par `NULL`. La séquence elle-même est terminée par `NULL`.

Le champ `seq` (un `char ***`) peut sembler intimidant au premier abord. Pour bien le comprendre, nous avons trouvé utile de regarder un exemple concret.

Comprendre `seq` par l'exemple

Pour la commande `cat fichier.txt | grep error | wc -l` :

```
seq[0] = {"cat", "fichier.txt", NULL} // 1re commande
seq[1] = {"grep", "error", NULL}      // 2e commande
seq[2] = {"wc", "-l", NULL}           // 3e commande
seq[3] = NULL                         // Fin de sequence
```

Cette structure est parfaitement adaptée pour `execvp()` car on peut directement écrire :

```
execvp(seq[i][0], seq[i]);
// seq[i][0] = nom de la commande
// seq[i]    = tableau {nom, arg1, arg2, ..., NULL}
```

Nous avons trouvé cette correspondance très élégante : le format de `seq` colle exactement à ce qu'attend `execvp`.

2.2 Les appels système essentiels

Passons maintenant aux appels système que nous avons utilisés tout au long du projet. Nous les présentons un par un, car chacun joue un rôle précis.
Le premier, et sans doute le plus fondamental, est `fork()`.

`fork()` - Création de processus

L'appel système `fork()` crée une copie du processus courant (le **fils**). Après le `fork()`, les deux processus (père et fils) s'exécutent en parallèle.

- Retourne `0` dans le processus fils.
- Retourne le **PID du fils** dans le processus père.
- Retourne `-1` en cas d'erreur.

Une fois le fils créé, il faut lui dire quel programme exécuter. C'est le rôle de `execvp()`.

`execvp()` - Exécution d'un programme

`execvp(const char *file, char *const argv[])` remplace le processus courant par le programme `file`. Le processus courant est **entièrement remplacé** : si `execvp` réussit, le code qui suit n'est jamais exécuté.
- `file` : nom de la commande (recherchée dans le `PATH`).
- `argv` : tableau d'arguments terminé par `NULL`.

Pour connecter plusieurs commandes entre elles (les fameux pipelines), nous avons eu besoin de `pipe()`.

`pipe()` - Tube de communication

`pipe(int pipefd[2])` crée un tube de communication unidirectionnel entre deux processus :
- `pipefd[0]` : extrémité de **lecture**.
- `pipefd[1]` : extrémité d'**écriture**.

Ce qui est écrit dans `pipefd[1]` peut être lu depuis `pipefd[0]`.

Mais créer un pipe ne suffit pas : il faut encore dire au processus fils d'utiliser ce pipe comme entrée ou sortie standard. C'est là qu'intervient `dup2()`.

`dup2()` - Duplication de descripteurs

`dup2(int oldfd, int newfd)` fait en sorte que le descripteur `newfd` devienne une copie de `oldfd`. Cela permet de **rediriger** les entrées/sorties standard :
- `dup2(fd, STDIN_FILENO)` : redirige l'entrée standard vers `fd`.
- `dup2(fd, STDOUT_FILENO)` : redirige la sortie standard vers `fd`.

Enfin, le père doit attendre que ses fils terminent, sinon le shell afficherait son prompt avant la fin de la commande.

`waitpid()` - Attente de processus

`waitpid(pid_t pid, int *status, int options)` attend la fin d'un processus fils.
- `pid` : PID du fils à attendre (`-1` pour n'importe quel fils).
- `status` : stocke le code de retour du fils.
- `options` : `0` pour attente bloquante, `WNOHANG` pour non-bloquante.

2.3 Commandes intégrées vs commandes externes

Comme nous l'avons mentionné dans notre stratégie, nous avons compris qu'il fallait traiter différemment les commandes intégrées et les commandes externes.

Commande intégrée / externe

Il existe visiblement deux types de commandes dans un shell :

Commandes intégrées : Elles s'exécutent **dans le processus du shell** lui-même, sans `fork()`. C'est indispensable pour les commandes qui modifient l'état du shell.
Exemple : `cd` doit changer le répertoire courant **du shell**. Si on l'exécutait dans un fils (avec `fork`), seul le fils changerait de répertoire, et le shell resterait au même endroit.

Commandes externes : Elles s'exécutent dans un **processus fils** via `fork()` + `execvp()`. Le shell n'a pas besoin de les implémenter : il délègue au système.
Exemple : `ls`, `cat`, `grep`, `wc` sont des programmes du système.

Ce point nous a semblé confus au début : est-ce que nous devions recoder chaque commande nous-mêmes ?
Heureusement, non, car le shell se contente de lancer les programmes existants du système via `execvp()`. Seules les commandes qui doivent modifier le shell lui-même (comme `cd` ou `quit`) sont implémentées comme commandes intégrées.

3 Architecture du shell

3.1 Choix d'architecture : approche modulaire

Quand nous avons implémenté la commande `quit` (étape 1 du TP), nous nous sommes vite rendu compte qu'ajouter un `if` dans le `main()` pour chaque nouvelle commande allait devenir ingérable. Nous avons donc décidé de mettre en place une **architecture modulaire** dès le départ, basée sur une table de fonctions.

Voici concrètement ce que ça change.

Comparaison des approches

Approche naïve (ce que nous avons voulu éviter) :

```
// main() pollue avec des cas particuliers
if (strcmp(cmd[0], "quit") == 0) {
    exit(0);
} else if (strcmp(cmd[0], "cd") == 0) {
    chdir(args[1]);
} else if (strcmp(cmd[0], "help") == 0) {
    // ...
} else {
    fork(); execvp(); wait();
}
```

Approche modulaire (ce que nous avons adopté) :

```
// main() reste simple et lisible
execute_cmdline(1); // Une seule ligne !
```

3.2 La table des commandes intégrées

Le cœur de cette architecture est une **table extensible** qui associe chaque nom de commande à sa fonction d'implémentation. Pour cela, nous avons défini un type structuré.

Type `builtin_cmd_t`

```
typedef struct {
    char *name;              // Nom de la commande
    int (*func)(char **args); // Pointeur de fonction
    char *description;       // Description pour l'aide
} builtin_cmd_t;
```

Le champ `func` est un **pointeur de fonction** : il permet d'appeler la bonne fonction sans `if/else`, en parcourant simplement la table.

Concrètement, la table ressemble à ceci dans notre code.

Table des commandes intégrées

```
builtin_cmd_t builtin_commands[] = {
    {"quit", builtin_quit, "Quitte␣le␣shell"},
    {"exit", builtin_exit, "Quitte␣le␣shell"},
    {"cd",  builtin_cd,  "Change␣de␣repertoire"},
    {"help", builtin_help, "Affiche␣l'aide"},
    {NULL, NULL, NULL} // Fin de table
};
```

Le jour où nous voulons ajouter une nouvelle commande intégrée, il nous suffit d'écrire **une fonction** et d'ajouter **une ligne** dans cette table. Nous n'avons pas à toucher au `main()`.

3.3 Flux d'exécution

Pour mieux visualiser comment tout s'articule, voici la chaîne d'appels que suit notre shell quand l'utilisateur tape une commande.

Chaîne d'appels

```
main()                        // Boucle principale
  |-> readcmd()               // Lire et parser la commande
  |-> execute_cmdline(l)      // Point d'entree execution
       |-> try_execute_builtin() // C'est une commande integree ?
       |     |-> builtin_quit/cd/help/...
       |-> execute_pipeline(l) // Sinon, pipeline
            |-> execute_simple_command() // Si 1 commande
            |-> [pipeline parallele]  // Si pipes
```

La fonction `try_execute_builtin()` retourne `-1` si la commande n'est pas une commande intégrée, et ≥ 0 (le code de retour) si elle a été exécutée. Cela permet de savoir s'il faut poursuivre avec l'exécution externe.

4 Implémentation étape par étape

Nous allons maintenant détailler chaque étape de notre implémentation, dans l'ordre chronologique où nous les avons réalisées.

4.1 Étape 1 - Commande `quit`

La toute première chose demandée par le sujet est de pouvoir quitter proprement le shell. C'est simple, mais c'est ce qui nous a fait réfléchir à l'architecture.

Implémentation de `quit`

```c
int builtin_quit(char **args) {
    printf("A␣bientot␣!\n");
    exit(0);
    return 0; // Jamais atteint, mais convention (pour les warnings notamment)
}
```

Grâce à la table des commandes intégrées, cette commande est automatiquement reconnue et exécutée par `try_execute_builtin()` sans toucher au `main()`.

4.2 Étape 2 - Exécution d'une commande simple

Une fois `quit` en place, nous avons voulu pouvoir lancer des commandes comme `ls -a`. C'est ici que le patron `fork/exec/wait` entre en jeu.

Schéma fork/exec/wait

Le patron d'exécution d'une commande externe suit toujours le même schéma :

1. `fork()` : le shell se dédouble.
2. Dans le **fils** : `execvp()` remplace le fils par le programme demandé.
3. Dans le **père** : `waitpid()` attend que le fils se termine.

Nous avons encapsulé cette logique dans une fonction dédiée pour ne pas la dupliquer.

Implémentation de `execute_simple_command()`

```c
void execute_simple_command(char **cmd,
                            char *input_file,
                            char *output_file,
                            int out_append) {


        pid_t pid = fork();

        if (pid == 0) {
            // FILS : configurer redirections puis executer
            setup_redirections(input_file, output_file, out_append);
            execvp(cmd[0], cmd);

            command_error(cmd[0]);
            exit(1);
        } else {
            // PERE : attendre la fin du fils
            int status;
            waitpid(pid, &status, 0);
        }
}
```

Un détail important : si `execvp()` réussit, le code qui suit n'est *jamais* exécuté (le processus est remplacé). Donc `command_error()` n'est atteint que si la commande n'existe pas.

4.3 Étape 3 - Redirections d'entrée/sortie

L'étape suivante consiste à gérer les redirections `< fichier`, `> fichier` et `» fichier`. Concrètement, quand l'utilisateur tape `cat < input.txt`, il faut que `cat` lise depuis `input.txt` au lieu du clavier. De même, `echo hello > out.txt` écrit dans un fichier, et `echo world » out.txt` ajoute à la fin d'un fichier existant.

Principe des redirections

La redirection repose sur `dup2()` : on ouvre le fichier cible avec `open()`, puis on redirige le descripteur standard (`STDIN` ou `STDOUT`) vers ce fichier. Ainsi, le programme lit/écrit dans le fichier **sans le savoir**.
Il existe trois types de redirection :

- `< fichier` : redirige l'entrée standard depuis un fichier.
- `> fichier` : redirige la sortie standard vers un fichier (**écrase** le contenu existant).
- `» fichier` : redirige la sortie standard vers un fichier (**ajoute** à la fin, sans écraser).

La différence entre `>` et `»` se traduit au niveau de l'appel à `open()` : on utilise le flag `O_TRUNC` pour écraser, et `O_APPEND` pour ajouter.

Nous avons isolé cette logique dans une fonction `setup_redirections()` pour pouvoir la réutiliser partout. Le paramètre `out_append` permet de distinguer `>` (troncature) de `»` (ajout).

Implémentation de `setup_redirections()`

```
void setup_redirections(char *input_file,
                        char *output_file,
                        int out_append) {

    if (input_file != NULL) {
        int fd_in = open(input_file, O_RDONLY);
        if (fd_in < 0) { perror(input_file); exit(1); }
        dup2(fd_in, STDIN_FILENO); // stdin <- fichier
        close(fd_in);
    }
    if (output_file != NULL) {
        int flags = O_WRONLY | O_CREAT;
        flags |= out_append ? O_APPEND : O_TRUNC;
        int fd_out = open(output_file, flags, 0644);
        if (fd_out < 0) { perror(output_file); exit(1); }
        dup2(fd_out, STDOUT_FILENO); // stdout -> fichier
        close(fd_out);
    }
}
```

La clé est le choix du flag : `O_TRUNC` efface le fichier avant d'écrire, tandis que `O_APPEND` positionne le curseur à la fin pour écrire à la suite.

Pour supporter `»`, nous avons aussi dû modifier le parser `readcmd.c` : le tokenizer produit désormais un token unique `"»"` (au lieu de deux `">"`), et la structure `cmdline` possède un nouveau champ `out_append` qui vaut `1` quand `»` est utilisé.

Les redirections sont configurées **dans le processus fils**, après le `fork()` et avant le `execvp()`. Cela garantit que seul le fils est affecté, pas le shell parent.

4.4 Étape 4 - Gestion des erreurs

Tout au long du développement, nous avons veillé à afficher des messages d'erreur clairs. Le sujet demande deux cas précis.

Gestion des erreurs

```c
// Commande inexistante : affichage sur stderr
void command_error(const char *cmd) {
    fprintf(stderr, "%s: command not found\n", cmd);
}

// Erreur systeme (fichier inaccessible, etc.)
// -> utiliser perror() qui affiche automatiquement
//    le message correspondant a errno
perror("nom_fichier");
// Affiche : nom_fichier: Permission denied
```

L'avantage de `perror()` est qu'il traduit automatiquement le code d'erreur système en message lisible. Nous l'utilisons pour les erreurs liées aux fichiers et aux appels système.

4.5 Étapes 5-7 - Pipes et exécution en pipeline

C'est la partie du TP qui nous a demandé le plus de réflexion. Quand l'utilisateur tape `cmd1 | cmd2 | cmd3`, toutes les commandes doivent s'exécuter **en parallèle**, pas l'une après l'autre.

4.5.1 Pourquoi une exécution parallèle ?

Notre première intuition était de lancer chaque commande séquentiellement : exécuter `cmd1`, récupérer sa sortie, puis la passer à `cmd2`. Mais en y réfléchissant, nous avons compris que c'était fondamentalement incorrect.

Exécution parallèle vs séquentielle

Séquentiel (incorrect) : On lance `cmd1`, on attend qu'elle finisse, puis on lance `cmd2` avec sa sortie.
Problèmes :
- Lent : chaque commande doit finir avant que la suivante démarre.
- Mémoire : il faut stocker toute la sortie intermédiaire.
- Impossible avec les flux infinis (`tail -f log | grep error`).

Parallèle (correct) : On lance **toutes** les commandes en même temps, connectées par des pipes. Les données transitent en flux continu.

Le cas de `tail -f` est parlant : ce programme ne termine jamais, donc si on attendait sa fin avant de lancer `grep`, le pipeline resterait bloqué indéfiniment.

4.5.2 Anatomie d'un pipeline

Voyons comment nous avons construit notre pipeline étape par étape. D'abord, il faut créer les tubes qui vont relier les processus entre eux.

Création des pipes

Pour n commandes, il faut $n - 1$ pipes :

```c
int num_cmds = count_commands(l->seq);
int pipes[num_cmds - 1][2];

for (int i = 0; i < num_cmds - 1; i++) {
    pipe(pipes[i]);
}
```

Pour `cat | grep | wc` (3 commandes) :
- `pipes[0]` : tube entre `cat` et `grep`
- `pipes[1]` : tube entre `grep` et `wc`

Ensuite vient le cœur du mécanisme :
Le fork de tous les processus dans une même boucle, pour qu'ils démarrent simultanément.

Fork de tous les processus en parallèle

```
pid_t pids[num_cmds];
for (int i = 0; i < num_cmds; i++) {
    pids[i] = fork();

    if (pids[i] == 0) {
        // FILS i : configurer ses redirections

        // Entree : pipe precedent (sauf 1re commande)
        if (i > 0)
            dup2(pipes[i-1][0], STDIN_FILENO);

        // Sortie : pipe suivant (sauf derniere commande)
        if (i < num_cmds - 1)
            dup2(pipes[i][1], STDOUT_FILENO);

        // CRITIQUE : fermer TOUS les pipes dans le fils
        for (int j = 0; j < num_cmds - 1; j++) {
            close(pipes[j][0]);
            close(pipes[j][1]);
        }

        execvp(l->seq[i][0], l->seq[i]);
        command_error(l->seq[i][0]);
        exit(1);
    }
}
```

Il est **critique** de fermer **tous** les pipes dans **chaque** processus fils, pas seulement ceux qu'il utilise.

Si un descripteur reste ouvert, le processus en bout de chaîne attend un EOF qui n'arrivera jamais et le pipeline reste **bloqué indéfiniment**. C'est un piège dans lequel nous sommes tombés avant de comprendre le mécanisme.

Pour finir, le processus parent doit lui aussi fermer ses pipes (il ne s'en sert pas) puis attendre la fin de tous ses fils.

Le parent : fermer les pipes et attendre

```
// Le parent ferme TOUS les pipes (il ne les utilise pas)
for (int i = 0; i < num_cmds - 1; i++) {
    close(pipes[i][0]);
    close(pipes[i][1]);
}

// Attendre que TOUS les fils se terminent
for (int i = 0; i < num_cmds; i++) {
    int status;
    waitpid(pids[i], &status, 0);
}
```

5 Le `main()` final

Grâce à l'architecture modulaire que nous avons mise en place, notre `main()` tient en une vingtaine de lignes et reste parfaitement lisible. Toute la complexité est encapsulée dans les fonctions appelées.

Boucle principale du shell

```
int main() {
    while (1) {
        struct cmdline *l;

        printf("Mini-shell >>> ");
        fflush(stdout);

        l = readcmd();

        if (!l) {                // EOF (Ctrl+D)
            printf("\nexit\n");
            exit(0);
        }

        if (l->err) {            // Erreur de syntaxe
            fprintf(stderr, "error: %s\n", l->err);
            continue;
        }

        execute_cmdline(l);      // Execution !
    }
    return 0;
}
```

Le `main()` ne contient aucune logique d'exécution : tout est délégué à `execute_cmdline()`, qui elle-même dispatche vers les commandes intégrées ou le pipeline.

6 Tests et validation

À chaque étape de notre développement, nous avons testé notre shell pour nous assurer que les nouvelles fonctionnalités ne cassaient pas les précédentes.

6.1 Script de test

Le TP fournit un script Perl `sdriver.pl` et des fichiers de test que nous avons utilisés pour valider chaque étape.

Utilisation du script de test

```
./sdriver.pl -t test01.txt -s ./shell
```

- `test01.txt` : teste la commande `quit`.
- `test02.txt` : teste une commande simple (`ls`).
- `test03.txt` : teste une succession de commandes.
- `test04.txt` : vérifie l'absence de processus zombies.

6.2 Création de tests automatiques

Le jeu de tests fourni (test01 à test04) ne couvrait que les bases : `quit`, une commande simple, une succession, et les zombies. Pour valider l'ensemble des fonctionnalités que nous avons implémentées, nous avons créé **14 fichiers de test supplémentaires** (test05 à test18), chacun ciblant une fonctionnalité précise.

6.2.1 Format des fichiers de test

Avant de détailler les tests, il est utile de comprendre le format attendu par `sdriver.pl`.

Format des traces `sdriver.pl`

Chaque fichier de test est une séquence de **directives** envoyées au shell :

- **Texte brut** : envoyé tel quel au shell comme une commande utilisateur.
- `SLEEP n` : pause de `n` secondes (laisse le temps au shell de traiter).
- `INT` : envoie `SIGINT` (simule Ctrl+C).
- `TSTP` : envoie `SIGTSTP` (simule Ctrl+Z).
- `QUIT` / `KILL` : envoie `SIGQUIT` / `SIGKILL`.
- `WAIT` : attend la fin du shell avant de terminer le test.
- `#` : commentaire (ignoré).

6.2.2 Tests des commandes de base (test05, test11, test12)

Tests des commandes simples et des erreurs

test05 – Commandes avec arguments :

```
echo Hello World
ls -la
quit
```

Vérifie que le shell transmet correctement les arguments aux commandes externes.
test11 – Gestion des erreurs :

```
commandebidon          # -> "command not found"
ls /repertoire_inexistant_xyz # -> erreur systeme
quit
```

Vérifie que le shell affiche un message d'erreur adapté **sans planter**.
test12 – Commandes intégrées (`cd`, `help`) :

```
help                   # Liste des builtins
cd /tmp                # Changement de repertoire
pwd                    # Doit afficher /tmp
cd                     # Retour au HOME
pwd                    # Doit afficher $HOME
quit
```

Vérifie que `cd` modifie bien le répertoire du shell (pas d'un fils) et que `cd` sans argument ramène au `HOME`.

6.2.3 Tests des redirections (test06, test07)

Tests des redirections d'entrée/sortie

test06 – Redirection `>` et `»` :

```
echo premiere ligne > /tmp/test_shell_redir.txt
echo deuxieme ligne >> /tmp/test_shell_redir.txt
cat /tmp/test_shell_redir.txt
quit
```

Le `cat` final doit afficher les **deux** lignes, prouvant que `>` crée le fichier et `»` ajoute à la suite sans écraser.
test07 – Redirection `<` :

```
echo contenu du fichier > /tmp/test_shell_in.txt
cat < /tmp/test_shell_in.txt
quit
```

Vérifie que `cat` lit bien depuis le fichier redirigé en entrée.

6.2.4 Tests des pipes (test08, test09, test10)

Tests des pipelines simples et complexes

test08 – Pipe simple (2 commandes) :

```
echo aaa bbb ccc | wc -w    # -> 3
ls -la | grep src
quit
```

test09 – Pipe multiple (4 commandes) :

```
echo un deux trois quatre cinq | xargs -n1 | sort | head -3
ls -la | grep -v total | wc -l
quit
```

Teste un pipeline à 4 étages : `echo` produit les mots, `xargs -n1` les met un par ligne, `sort` les trie, et `head -3` ne garde que les 3 premiers.

test10 – Pipes combinés avec redirections :

```
echo alpha bravo charlie > /tmp/test_shell_pipe.txt
cat < /tmp/test_shell_pipe.txt | xargs -n1 | sort > /tmp/test_shell_pipe_out.txt
cat /tmp/test_shell_pipe_out.txt
quit
```

C'est le test le plus complet pour les pipes : il combine redirection d'entrée (<), deux pipes, et redirection de sortie (>), le tout dans une seule commande.

Stratégie de validation des pipes

Pour tester un pipeline, on utilise des commandes dont la sortie est **déterministe** : `echo` produit un texte connu, `sort` trie alphabétiquement, `wc -w` compte les mots. Ainsi, on peut vérifier que le résultat est correct sans dépendre de l'état du système.

6.2.5 Tests de l'arrière-plan et des jobs (test13, test17, test18)

Tests de la gestion des jobs

test13 – Lancement en arrière-plan et `jobs` :

```
sleep 5 &       # Lance job 1
sleep 7 &       # Lance job 2
jobs             # Doit afficher 2 jobs Running
quit
```

test18 – Commande `stop` :

```
/bin/sleep 60 & # Lance en background
jobs            # -> [1] Running
stop %1         # Suspendre le job
jobs            # -> [1] Stopped
quit
```

Vérifie que `stop %N` envoie bien `SIGSTOP` au groupe du job.

test17 – Commande `bg` :

```
sleep 10        # Premier plan
TSTP            # Ctrl+Z -> stoppe
jobs            # -> [1] Stopped
bg %1           # Reprendre en background
jobs            # -> [1] Running
quit
```

Teste le cycle complet : foreground → Ctrl+Z → Stopped → `bg` → Running.

6.2.6 Tests des signaux (test14, test15)

Ces tests sont les plus importants car ils valident la **survie du shell** face aux signaux.

Tests de Ctrl+C et Ctrl+Z

test14 – SIGINT (Ctrl+C), le shell survit :

```
/bin/sleep 30    # Commande au premier plan
INT              # sdriver envoie SIGINT (= Ctrl+C)
echo shell toujours vivant apres Ctrl+C
quit
```

Le `echo` après l'`INT` **doit** s'exécuter. Si le shell est mort, le test échoue.
test15 – SIGTSTP (Ctrl+Z) → `jobs` → `fg` → SIGINT :

```
/bin/sleep 60    # Foreground
TSTP             # Ctrl+Z -> suspend
jobs             # Doit afficher [1] Stopped
fg %1            # Reprendre au premier plan
INT              # Ctrl+C -> termine
echo shell vivant apres Ctrl+Z puis fg puis Ctrl+C
quit
```

C'est le test le plus complet du projet : il enchaîne suspension, listage des jobs, reprise, et interruption. Le shell doit survivre à **chaque** étape.

Utilisation de chemins absolus dans les tests de signaux

On utilise `/bin/sleep` au lieu de `sleep` dans les tests de signaux pour éviter toute ambiguïté de `PATH`. C'est une bonne pratique dans les tests automatiques : on s'assure d'exécuter exactement le programme voulu.

6.2.7 Test des zombies (test16)

Test de la gestion des processus zombies

test16 – 3 jobs en arrière-plan puis `ps` :

```
/bin/echo job1 &    # Termine immediatement
/bin/echo job2 &
/bin/echo job3 &
SLEEP 2             # Attendre que tout se termine
/bin/ps -o pid,stat,comm
quit
```

Après 2 secondes, les trois `echo` sont terminés. La sortie de `ps` ne doit contenir **aucun** processus avec l'état `Z` (zombie) ou la mention `<defunct>`. Ce test valide que le handler `SIGCHLD` ramasse correctement les fils terminés en arrière-plan.

6.2.8 Récapitulatif des tests

Test	Fonctionnalité testée	Catégorie
test01–04	quit, commande simple, succession, zombies	*Fournis*
test05	Commandes avec arguments	Commandes
test11	Erreurs (commande/répertoire inexistants)	Commandes
test12	Builtins : `cd`, `help`, `pwd`	Commandes
test06	Redirections `>` et `>>`	Redirections
test07	Redirection `<`	Redirections
test08	Pipe simple (2 commandes)	Pipes
test09	Pipe multiple (4 commandes)	Pipes
test10	Pipe + redirections combinées	Pipes
test13	Arrière-plan (`&`) + `jobs`	Jobs
test17	`bg` : reprendre un job stoppé	Jobs
test18	`stop` : suspendre un job	Jobs
test14	Ctrl+C : shell survit à SIGINT	Signaux
test15	Ctrl+Z → `fg` → Ctrl+C	Signaux
test16	Pas de zombies après jobs bg	Signaux

6.2.9 Script de démonstration

Pour la soutenance orale, nous avons également créé un script `demonstration.sh` qui automatise le passage de tous les tests et guide la présentation.

Script `demonstration.sh`

Le script est organisé en **13 sections** qui suivent la progression du TP :

1. Compilation du projet (`make clean && make`)
2. Commande `quit`
3. Commandes simples
4. Commandes intégrées (`cd`, `help`)
5. Gestion des erreurs
6. Redirections (`< > >>`)
7. Pipes simples et multiples
8. Pipes + redirections combinées
9. Arrière-plan et `jobs`
10. Signal Ctrl+C (SIGINT)
11. Signal Ctrl+Z (SIGTSTP) + `fg`
12. Gestion des zombies
13. Commande `stop`

Chaque section affiche un bandeau coloré, lance le test correspondant via `sdriver.pl`, et affiche le résultat ([OK] ou [FAIL]). À la fin, un récapitulatif comptabilise les tests passés, échoués et en timeout. Le script se termine par un lancement interactif du shell pour une démonstration live.

6.3 Tests manuels

En complément des tests automatiques, nous avons effectué une série de tests manuels pour couvrir les cas d'usage courants. Voici la liste que nous avons suivie.

Checklist de validation

Commandes intégrées :

```
Mini-shell >>> help   # Liste des commandes integrees
Mini-shell >>> cd /tmp # Changement de repertoire
Mini-shell >>> cd     # Retour au HOME
Mini-shell >>> quit   # Sortie propre
```

Commandes externes :

```
Mini-shell >>> ls -la
Mini-shell >>> echo Hello World
Mini-shell >>> commandebidon # -> "command␣not␣found"
```

Redirections :

```
Mini-shell >>> echo "test" > output.txt
Mini-shell >>> cat < output.txt
Mini-shell >>> echo "ligne␣1" > append.txt
Mini-shell >>> echo "ligne␣2" >> append.txt
Mini-shell >>> cat append.txt # -> "ligne␣1" puis "ligne␣2"
```

Pipes :

```
Mini-shell >>> ls | wc -l
Mini-shell >>> ls -la | grep shell | wc -l
Mini-shell >>> cat /etc/passwd | grep root | wc -l
```

Pipes avec redirections :

```
Mini-shell >>> cat < input.txt | grep error > output.txt
```

Tous ces tests passent correctement sur notre implémentation.

7 Pièges courants et solutions

Au fil de notre développement, nous avons rencontré plusieurs pièges que nous documentons ici.

Le premier piège est le plus fourbe, car il ne produit pas d'erreur visible : le programme se bloque simplement sans rien dire.

Piège 1 : Ne pas fermer tous les pipes

Chaque processus fils hérite de **tous** les descripteurs de fichiers ouverts par le père. Si un fils ne ferme pas un pipe qu'il n'utilise pas, le processus en fin de chaîne ne recevra jamais l'EOF et restera bloqué.
Règle d'or : Dans chaque fils, fermer **tous** les pipes après les `dup2()`.

Le deuxième piège est plus subtil et concerne la distinction intégrée/externe que nous avons évoquée plus tôt.

Piège 2 : Exécuter une commande intégrée dans un fils

Si on exécute `cd` dans un processus fils (après un `fork()`), seul le fils change de répertoire. Le shell parent reste dans le même dossier.
Solution : Les commandes intégrées doivent être détectées **avant** le `fork()`, et exécutées directement dans le processus du shell.

Enfin, un piège classique lié à l'ordre des opérations dans le processus fils.

Piège 3 : Ordre des opérations

L'ordre correct dans un processus fils est :

1. `fork()` pour créer le fils.
2. Configurer les redirections (`dup2()`).
3. Fermer les descripteurs inutiles (`close()`).
4. `execvp()` pour lancer la commande.

Inverser les étapes 2 et 4 ferait que le programme s'exécute sans les bonnes redirections.

8 Exécution en arrière-plan et gestion des jobs

Jusqu'ici, notre shell ne pouvait exécuter qu'une commande à la fois : il lançait un processus fils et attendait sa fin avant de rendre la main. Mais un vrai shell permet aussi de lancer des commandes **en arrière-plan** (avec `&`), de les **suspendre** (Ctrl+Z), et de les **reprendre** (commandes `fg` et `bg`). C'est tout l'enjeu de cette section.

8.1 Premier plan vs arrière-plan

Foreground vs Background

Un shell Unix distingue deux modes d'exécution :
Premier plan (foreground) : le shell **attend** la fin de la commande avant d'afficher le prompt suivant. L'utilisateur ne peut rien taper pendant ce temps.
Arrière-plan (background) : le shell lance la commande mais **n'attend pas** sa fin. Il rend l'invite immédiatement, et l'utilisateur peut continuer à taper.
La syntaxe est simple : ajouter `&` à la fin d'une commande la lance en arrière-plan.

La détection du `&` se fait au niveau du parseur `readcmd()`. Si la ligne se termine par ce symbole, le champ `l->bg` vaut `1`. Notre moteur d'exécution teste ce champ pour savoir s'il doit attendre ou non.

8.2 La table des jobs

Pour pouvoir lister, suspendre et reprendre les commandes, le shell doit se souvenir de toutes les tâches en cours. Nous avons donc mis en place une **table des jobs**.

Structure `job_t`

```
typedef struct {
    int id;                 // Numero visible (1, 2, 3...)
    pid_t pgid;             // Process Group ID
    pid_t pids[MAX_PIPELINE]; // PIDs des processus du pipeline
    int num_procs;          // Nombre de processus
    int num_done;           // Nombre de processus termines
    job_state_t state;      // RUNNING, STOPPED ou DONE
    int bg;                 // 1 = arriere-plan, 0 = premier plan
    char cmdline[256];      // Texte de la commande
} job_t;
```

La table est un tableau fixe de taille `MAXJOBS` (10). Chaque slot libre a `pgid == 0`.

Les commandes lancées au premier plan reçoivent un identifiant interne `id = 0` (invisible dans la commande `jobs`). Seuls les jobs mis en arrière-plan ou stoppés par Ctrl+Z obtiennent un identifiant visible (`id > 0`).

Cela évite de consommer inutilement des numéros pour les commandes courantes (`ls`, `cat`, etc.) qui se terminent aussitôt.

8.3 Groupe de processus

Un concept clé pour la gestion des jobs est le **groupe de processus**. Sans lui, impossible de suspendre ou terminer proprement un pipeline entier.

Groupe de processus

Un **groupe de processus** est un ensemble de processus partageant un même identifiant de groupe (`pgid`). L'intérêt : on peut envoyer un signal à **tout le groupe** d'un seul coup avec `kill(-pgid, signal)`.
Dans notre shell, chaque pipeline forme son propre groupe. Le `pgid` est fixé au PID du **premier** processus du pipeline.

Nous proposons la solution suivante :

Mise en place des process groups

```
pid_t pgid = 0;

for (int i = 0; i < num_cmds; i++) {
    pids[i] = fork();

    if (pids[i] == 0) {
        // FILS : rejoindre le groupe du 1er fils
        setpgid(0, pgid);
        // ... redirections, execvp ...
    }

    // PERE : fixer le pgid (1er fils) et enregistrer
    if (i == 0) pgid = pids[0];
    setpgid(pids[i], pgid); // Aussi dans le pere !
}
```

Le `setpgid()` est appelé **à la fois** dans le fils **et** dans le père. C'est indispensable pour éviter une **situation de concurrence** / **conflit** : si le père essaie d'envoyer un signal au groupe avant que le fils ait eu le temps de s'y joindre, le signal serait perdu.

8.4 Commandes intégrées : `jobs`, `fg`, `bg`, `stop`

La gestion des jobs s'accompagne de quatre nouvelles commandes intégrées, que nous avons ajoutées à notre table de builtins.

Commande `jobs` : lister les travaux

```
int builtin_jobs(char **args) {
    for (int i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pgid != 0 && jobs[i].id > 0) {
            printf("[%d] %d %s\t%s\n",
                   jobs[i].id, jobs[i].pgid,
                   job_state_str(jobs[i].state),
                   jobs[i].cmdline);
        }
    }
    return 0;
}
```

Seuls les jobs avec `id` $>$ `0` sont affichés (les commandes foreground terminées ne sont pas visibles).

Commande `fg` : reprendre au premier plan

```
int builtin_fg(char **args) {
    job_t *j = parse_job_ref(args[1]); // %N ou PID
    if (j == NULL) {
        fprintf(stderr, "fg: aucun travail correspondant\n");
        return 1;
    }
    j->bg = 0;              // Passe en premier plan
    j->state = JOB_RUNNING;
    kill(-j->pgid, SIGCONT); // Reveiller le groupe
    wait_for_fg_job(j);     // Attendre sa fin
    return 0;
}
```

Le signal `SIGCONT` est envoyé au groupe entier (`-pgid`) pour réveiller tous les processus stoppés du pipeline.

Commande `bg` : reprendre en arrière-plan

```
int builtin_bg(char **args) {
    job_t *j = parse_job_ref(args[1]);
    j->bg = 1;
    j->state = JOB_RUNNING;
    kill(-j->pgid, SIGCONT); // Reveiller en background
    return 0;               // NE PAS attendre !
}
```

La différence avec `fg` : on **ne bloque pas**. Le shell rend l'invite immédiatement.

8.5 Attente du premier plan : la boucle `sleep(1)`

Quand un job est au premier plan, le shell doit attendre sa fin. Mais comment ? On ne peut pas utiliser `waitpid()` directement dans la boucle principale, car c'est le handler `SIGCHLD` qui s'en charge.

Attente active avec `sleep(1)`

La solution recommandée par le TP est une **boucle d'attente** :

```
void wait_for_fg_job(job_t *j) {
    while (j->state == JOB_RUNNING) {
        sleep(1); // Interrompu par SIGCHLD !
    }
    if (j->state == JOB_DONE) {
        remove_job(j->pgid);
    } else if (j->state == JOB_STOPPED) {
        printf("[%d] Stopped\t\t%s\n", j->id, j->cmdline);
    }
}
```

Pourquoi ça marche ? Quand un fils se termine ou est stoppé, le noyau envoie `SIGCHLD` au shell. Ce signal **interrompt** le `sleep(1)`, qui retourne immédiatement. Le handler met à jour `j->state`, et la boucle s'arrête.

L'intérêt de cette approche est que `waitpid()` n'est appelé qu'à **un seul endroit** dans tout le programme : le handler `SIGCHLD`. Cela évite les concurrences entre le shell et le handler qui essaieraient tous les deux de "ramasser" le même fils.

8.6 Exemple de session complète

Cycle de vie d'un job

```
Mini-shell >>> sleep 60 &
[1] 12345               # Lance en background, id=1

Mini-shell >>> jobs
[1] 12345 Running  sleep 60

Mini-shell >>> stop %1  # Suspendre le job
Mini-shell >>> jobs
[1] 12345 Stopped  sleep 60

Mini-shell >>> bg %1    # Reprendre en background
[1] sleep 60 &

Mini-shell >>> fg %1    # Ramener au premier plan
sleep 60
^C                      # Ctrl+C -> termine le job
Mini-shell >>>
```

9 Gestion des signaux

C'est la partie la plus subtile de tout le projet. Les signaux sont le mécanisme par lequel le noyau Unix communique des événements asynchrones aux processus. Dans un shell, il faut les maîtriser pour gérer Ctrl+C, Ctrl+Z et le nettoyage des processus fils.

9.1 Qu'est-ce qu'un signal ?

Signal Unix

Un **signal** est une notification asynchrone envoyée à un processus par le noyau (ou par un autre processus via `kill()`). Quand un processus reçoit un signal, il interrompt son exécution normale pour exécuter un **traitant** (*handler*).
Trois comportements possibles :

- **Action par défaut** : terminer le processus, le stopper, l'ignorer... (dépend du signal).
- **Ignorer** le signal : `SIG_IGN`.
- **Intercepter** le signal avec un handler personnalisé.

Voici les trois signaux qui nous intéressent dans ce projet :

Les trois signaux clés du shell

- `SIGINT` (Ctrl+C) : demande d'interruption. Action par défaut : **terminer** le processus.
- `SIGTSTP` (Ctrl+Z) : demande de suspension. Action par défaut : **stopper** le processus (il reste en mémoire mais ne s'exécute plus).
- `SIGCHLD` : envoyé au père quand un fils **se termine** ou est **stoppé**. Action par défaut : ignorer.

9.2 Handler Ctrl+C (`SIGINT`)

Quand l'utilisateur tape Ctrl+C, le terminal envoie `SIGINT` à **tous les processus** du groupe du terminal, y compris le shell lui-même. Or, on ne veut pas que le shell se termine ! On veut que seul le processus de premier plan soit interrompu.

Stratégie pour Ctrl+C

Le shell installe un handler personnalisé pour `SIGINT`. Ce handler :

1. Cherche le job de premier plan (`get_fg_job()`).
2. S'il existe, **transmet** le signal au **groupe de processus** de ce job via `kill(-pgid, SIGINT)`.
3. S'il n'existe pas (rien au premier plan), le signal est simplement ignoré.

Le shell lui-même n'est **jamais** terminé par Ctrl+C.

Implémentation de `sigint_handler()`

```
void sigint_handler(int sig) {
    job_t *fg = get_fg_job();
    if (fg != NULL) {
        // Transmettre SIGINT a TOUT le groupe du job fg
        kill(-fg->pgid, SIGINT);
    }
    // Si pas de job fg : ne rien faire (le shell survit)
}
```

Le signe - devant `fg->pgid` est crucial : il indique à `kill()` d'envoyer le signal à **tout le groupe** de processus, pas à un seul processus.

9.3 Handler Ctrl+Z (`SIGTSTP`)

Ctrl+Z fonctionne sur le même principe que Ctrl+C, mais au lieu de **terminer** le processus, il le **suspend**. Le processus reste en mémoire, prêt à reprendre quand on lui enverra `SIGCONT`.

Stratégie pour Ctrl+Z

Le handler de `SIGTSTP` :

1. Cherche le job de premier plan.
2. Transmet `SIGTSTP` au groupe de ce job via `kill(-pgid, SIGTSTP)`.
3. Le handler `SIGCHLD` (déclenché ensuite par le noyau) détectera que le fils est stoppé et mettra à jour la table des jobs.

Implémentation de `sigtstp_handler()`

```
void sigtstp_handler(int sig) {
    job_t *fg = get_fg_job();
    if (fg != NULL) {
        kill(-fg->pgid, SIGTSTP);
    }
}
```

Quand un processus est stoppé, le noyau envoie automatiquement `SIGCHLD` au père (le shell).
C'est le handler `SIGCHLD` qui se charge alors de :
— Détecter l'arrêt via `WIFSTOPPED(status)`.
— Marquer le job comme `JOB_STOPPED`.
— Lui attribuer un numéro visible s'il n'en avait pas (cas d'un job fg qui avait `id=0`).
Cela montre bien que `SIGCHLD` est la pièce centrale de toute la gestion des jobs.

9.4 Handler `SIGCHLD` : le ramasseur de zombies

C'est le handler le plus complexe et le plus important. C'est lui qui empêche la prolifération des processus zombies et qui maintient la table des jobs à jour.

Qu'est-ce qu'un processus zombie ?

Quand un processus fils se termine, il ne disparaît pas complètement : le noyau conserve ses informations (PID, code de retour) dans la table des processus jusqu'à ce que le père appelle `waitpid()`. Tant que le père n'a pas "ramassé" son fils, celui-ci reste dans l'état **zombie** (visible avec `ps` sous la forme `<defunct>`).
Si le shell lance beaucoup de commandes en arrière-plan sans jamais appeler `waitpid()`, les zombies s'accumulent et finissent par saturer la table des processus du système.

Implémentation de `sigchld_handler()`

```c
void sigchld_handler(int sig) {
    int saved_errno = errno; // (1)
    int status;
    pid_t pid;

    // (2)
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        // (3)
        for (int i = 0; i < MAXJOBS; i++) {
            if (jobs[i].pgid == 0) continue;
            int found = 0;
            for (int j = 0; j < jobs[i].num_procs; j++) {
                if (jobs[i].pids[j] == pid) {
                    if (WIFEXITED(status) || WIFSIGNALED(status)) {
                        // 4.1
                        jobs[i].pids[j] = 0;
                        jobs[i].num_done++;
                        if (jobs[i].num_done >= jobs[i].num_procs) {
                            jobs[i].state = JOB_DONE;
                        }
                    } else if (WIFSTOPPED(status)) {
                        // 4.2
                        jobs[i].state = JOB_STOPPED;
                        jobs[i].bg = 1;
                        if (jobs[i].id == 0) {
                            jobs[i].id = next_job_id++;
                        }
                    }
                    found = 1;
                    break;
                }
            }
            if (found) break;
        }
    }
    errno = saved_errno; // (5)
}
```

Ce handler est dense. Décortiquons-le point par point.

Analyse ligne par ligne du handler `SIGCHLD`

1. **Sauvegarde de `errno`** : Le handler est exécuté de manière asynchrone. Il pourrait modifier `errno` (variable globale utilisée par les appels système) et corrompre le code principal. On sauvegarde et restaure sa valeur.
2. **Boucle `while` avec `WNOHANG`** : Plusieurs fils peuvent se terminer "en même temps" (entre deux vérifications). Le `while` garantit qu'on les ramasse **tous**. Le flag `WNOHANG` fait que `waitpid` retourne `0` (au lieu de bloquer) quand il n'y a plus de fils à ramasser.
3. **Flag `WUNTRACED`** : Sans ce flag, `waitpid` ne signalerait que les fils **terminés**. Avec `WUNTRACED`, il signale aussi les fils **stoppés** (par Ctrl+Z), ce qui est indispensable pour détecter la suspension.
4. **Deux cas distincts** :
   — `WIFEXITED` / `WIFSIGNALED` : le processus s'est terminé (normalement ou tué par un signal). On incrémente `num_done`, et quand tous les processus du pipeline sont terminés, le job passe en `JOB_DONE`.
   — `WIFSTOPPED` : le processus a été stoppé. Le job entier passe en `JOB_STOPPED`. S'il était au premier plan (`id == 0`), on lui attribue un numéro visible.
5. **Restauration de `errno`** : Symétrique du point 1.

9.5 Masquage de SIGCHLD : éviter les concurrences

Il existe un problème de concurrence subtil lors de la création des processus. Le handler `SIGCHLD` peut être déclenché **avant** que le shell ait fini d'enregistrer le job dans la table.

Concurrence lors du `fork()`

Imaginons ce scénario :

1. Le shell fait `fork()` pour créer un fils.
2. Le fils s'exécute très vite et se termine **immédiatement**.
3. Le noyau envoie `SIGCHLD` au shell.
4. Le handler cherche le job correspondant... mais le shell n'a **pas encore** appelé `add_job()` !
5. Le PID du fils terminé n'est trouvé nulle part → il reste zombie.

La solution est de **bloquer** `SIGCHLD` pendant toute la phase de création du pipeline, et de ne le débloquer qu'après l'appel à `add_job()`.

Masquage de `SIGCHLD` pendant le fork

```
// BLOQUER SIGCHLD avant les forks
sigset_t mask_chld, prev_mask;
sigemptyset(&mask_chld);
sigaddset(&mask_chld, SIGCHLD);
sigprocmask(SIG_BLOCK, &mask_chld, &prev_mask);

// ... fork() de tous les fils du pipeline ...
// ... add_job() pour enregistrer le job ...

// DEBLOQUER SIGCHLD : les signaux en attente
// sont alors delivres d'un coup
sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

Entre le blocage et le déblocage, si un fils se termine, le signal `SIGCHLD` est mis en attente. Il sera délivré dès le déblocage, et à ce moment-là le job sera déjà dans la table : pas de race condition.

9.6 Restauration des handlers dans les processus fils

Un dernier point important : les processus fils héritent des handlers du père. Or, un programme lancé par le shell s'attend à ce que les signaux aient leur **comportement par défaut**. Il faut donc les restaurer.

Règle : restaurer `SIG_DFL` dans chaque fils

```
if (pid == 0) {
    // FILS : restaurer les handlers par defaut
    signal(SIGINT, SIG_DFL);
    signal(SIGTSTP, SIG_DFL);
    signal(SIGCHLD, SIG_DFL);

    // Debloquer SIGCHLD (herite du masquage du pere)
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);

    // ... puis execvp() ...
}
```

Sans cette restauration, un programme comme `cat` ne pourrait pas être interrompu par Ctrl+C (car il hériterait du handler du shell qui ne fait que transmettre).

On débloque aussi `SIGCHLD` dans le fils, car le père l'avait bloqué pendant la phase de création. Si on oublie ce déblocage, le fils (et donc le programme lancé) ne recevrait jamais `SIGCHLD` pour ses propres éventuels fils.

9.7 Installation des handlers dans le `main()`

Voici comment les trois handlers sont installés au démarrage du shell, avec `sigaction()`.

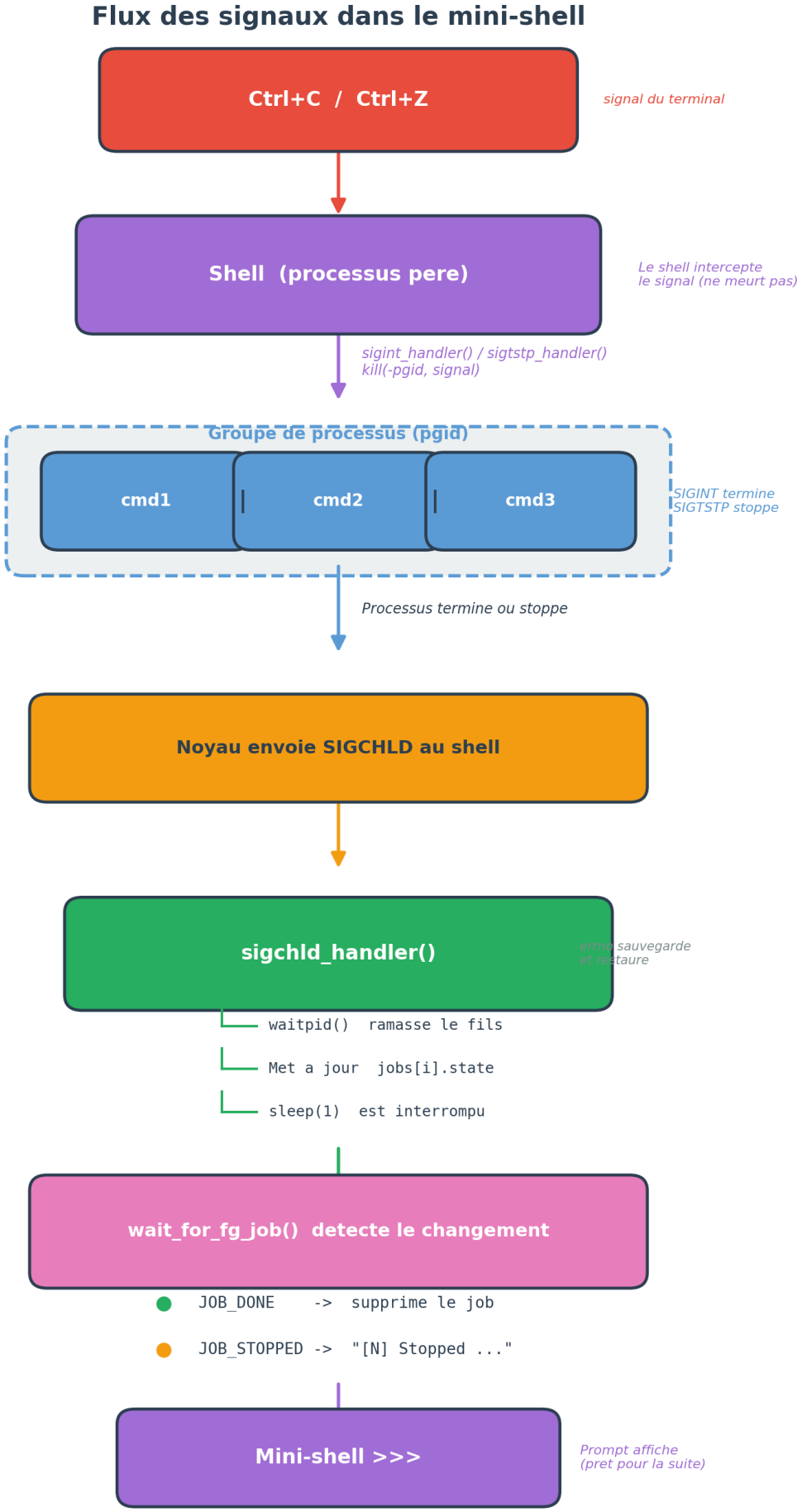
Installation des traitants de signaux

```
int main() {
    init_jobs();

    struct sigaction sa;

    // SIGCHLD : ramasser les fils termines/stoppes
    sa.sa_handler = sigchld_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sigaction(SIGCHLD, &sa, NULL);

    // SIGINT (Ctrl+C) : transmettre au job fg
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sigaction(SIGINT, &sa, NULL);

    // SIGTSTP (Ctrl+Z) : transmettre au job fg
    sa.sa_handler = sigtstp_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sigaction(SIGTSTP, &sa, NULL);

    // ... boucle principale ...
}
```

Chaque handler est installé avec `SA_RESTART` et un masque vide (`sigemptyset`), ce qui signifie qu'aucun signal supplémentaire n'est bloqué pendant l'exécution du handler.

9.8 Vue d'ensemble : flux des signaux

Pour résumer visuellement comment les signaux circulent dans le shell :


Flux des signaux dans le mini-shell
Ctrl+C / Ctrl+Z
signal du terminal
Shell (processus pere)
Le shell intercepte
le signal (ne meurt pas)
sigint_handler() / sigtstp_handler()
kill(-pgid, signal)
Groupe de processus (pgid)
cmd1
cmd2
cmd3
SIGINT termine
SIGTSTP stoppe
Processus termine ou stoppe
Noyau envoie SIGCHLD au shell
sigchld_handler()
errno sauvegarde
et restaure
waitpid() ramasse le fils
Met a jour jobs[i].state
sleep(1) est interrompu
wait_for_fg_job() detecte le changement
JOB_DONE -> supprime le job
JOB_STOPPED -> "[N] Stopped ..."
Mini-shell >>>
Prompt affiche
(pret pour la suite)

10 Bilan et extensions possibles

10.1 État de l'implémentation

Voici un résumé complet de ce que nous avons implémenté.

Étape	Fonctionnalité	État
1	Commande `quit` / `exit`	Implémenté
2	Commande simple (`ls -a`)	Implémenté
3	Redirections (`< > >>`)	Implémenté
4	Gestion des erreurs	Implémenté
5	Pipe simple (`cmd1 | cmd2`)	Implémenté
6-7	Pipes multiples	Implémenté
8	Arrière-plan (`&`)	Implémenté
9	Gestion des zombies (`SIGCHLD`)	Implémenté
Section 5 du sujet : "Pour aller plus loin"		
5.1	Ctrl+C (`SIGINT`) → job fg	Implémenté
5.2	Ctrl+Z (`SIGTSTP`) → job fg	Implémenté
5.3	Commande `jobs`	Implémenté
5.4	Commandes `fg`, `bg`, `stop`	Implémenté
5.5	Groupes de processus (`setpgid`)	Implémenté
5.6	Gestion de ∼ et *	Non-implémenté

10.2 Extensions possibles (pour aller plus loin)

Si nous devions poursuivre ce projet, voici les fonctionnalités que nous ajouterions en priorité.

Expansion du tilde et des variables d'environnement

Les primitives `wordexp()` et `wordfree()` permettent d'effectuer les expansions du shell avant l'exécution : le tilde (`~` → `/home/user`), le globbing (`*.c` → liste des fichiers), et les variables d'environnement (`$HOME`).

Historique des commandes

En intégrant la bibliothèque GNU Readline, on pourrait ajouter un historique des commandes (flèches haut/bas), l'auto-complétion par tabulation, et la coloration syntaxique de la ligne de commande en temps réel.

10.3 Conclusion

Ce projet nous a permis de comprendre concrètement le fonctionnement interne d'un shell Unix, depuis l'analyse syntaxique d'une commande jusqu'à l'orchestration complète des processus et des signaux. Les points clés que nous retenons :

- La **création de processus** avec `fork()` et `execvp()` : un mécanisme simple en apparence, mais dont les subtilités (code mort après `exec`, distinction père/fils) demandent de la rigueur.
- La **communication inter-processus** via les pipes : nous avons compris pourquoi l'exécution parallèle est indispensable et comment les données circulent en streaming.
- La **redirection** des entrées/sorties avec `dup2()` : un outil puissant qui permet de modifier le comportement d'un programme sans qu'il le sache.
- L'importance de la **fermeture des descripteurs** : le piège le plus vicieux du TP, car l'oubli provoque des blocages silencieux.
- La **gestion des signaux** avec `sigaction()` : nous avons compris le rôle central de `SIGCHLD` comme mécanisme unique de ramassage des processus fils, et pourquoi le masquage de signaux est indispensable pour éviter les race conditions pendant les `fork()`.
- La distinction entre **premier plan et arrière-plan** : la boucle `sleep(1)` interrompue par `SIGCHLD` est une solution élégante qui centralise toute la logique de `waitpid()` dans le handler.
- L'intérêt d'une **architecture modulaire** : un investissement initial qui nous a fait gagner beaucoup de temps par la suite, notamment lors de l'ajout des commandes `fg`/`bg`/`stop`/`jobs` qui n'ont nécessité qu'une ligne dans la table des builtins.