

CHRISTOFOROU ANTHONY

# RAPPORT

---

## Parsing

Ce module consiste a faire 3 choses. Il va s'occuper principalement de gerer l'input de l'utilisateur, et aussi afficher le **GUI** (interface) du shell. On va commencer avec l'interface,

```
void printGUI(void)
```

va juste print l'**utilisateur** du shell, le **host** (nom de la machine) et le **current directory**, tout ca avec des petite modification de couleurs.

On va voir ensuite,

```
char* getInput(void)
```

la premiere fonction qui s'occupe de l'input, elle va recuperer l'input taper sous forme **brute**.

Enfin pour la derniere fonction de ce module on a,

```
char** parseInput
```

qui va **parse** (decouper) l'input ET compter le nombre d'arguments. En effet, on va recuperer tout les arguments en utilisant le caractere **space** comme separation pour ce retrouver a la fin avec un **argv** et un **argc** a nous.

## Builtin

Dans ce module on va avoir 2 programmes de shell que l'on va implementer nous meme: **cd** et **exit**. Tout d'abord pour **cd**:

```
void cd(char* path)
```

Cette fonction est assez explicite, on va tout simplement utiliser un appel systeme pour changer le **current directory**. On test quand meme que si on ecrit rien apres cd, on change vers **home/user** (~).

La fonction **exit** et tout aussi simple,

```
void hexit(void)
```

on va envoyer un signal `SIGHUP` avec `kill` au processus du shell.

On a pour finir, une fonction qui va nous permettre **d'exécuter** ces deux programmes:

```
int execbin(int _argc, char* _argv[])
```

qui va comparer le premier argument et voir si c'est soit `cd` soit `exit` et lancer la fonction équivalente.

## Jobs

On va alors avoir le dernier module, qui va surtout être pour les signaux mais qui va tout d'abord tester si le job que l'on veut exécuter sera en `background` ou pas. Pour ça on va utiliser la fonction

```
int checkBackground(int _argc, char* _argv[])
```

qui va tout simplement tester si le dernier argument est **l'ampersand** (&). Dans ce cas la fonction va renvoyer 1. Dans le cas contraire elle renvoie 0.

Prochaine étape, on va gérer les signaux `SIGINT` et `SIGHUP` avec la fonction:

```
void handler(int sig)
```

Si on reçoit un `SIGINT` on va le rediriger sur le processus principal, le **foreground job** et si on reçoit un `SIGHUP` on va le rediriger vers tout ce qui tourne, **foreground** et **background**.

On s'occupe ensuite des **enfants**, les tâches en fond avec

```
void child_process_signal(int signum, siginfo_t *siginfo, void* unused)
```

qui va utiliser `waitpid` pour éviter tout zombie quand on quitte le background job.

Enfin, on a la fonction

```
void set_handlers(void)
```

qui sera là pour activer tous les masques et signaux, et pour rendre la fonction main plus propre.

On va donner à chaque signal leurs handlers précédents respectifs.

- `child_process_signal` pour `SIGCHLD`,
- `handler` pour `SIGINT` et `SIGHUP`

- Et on cree un mask pour ignorer `SIGTERM` et `SIGQUIT`

## Shell

Enfin, la fonction main, le coeur de notre shell. On va avoir la **loop** principale, un `while (1)` qui va garder le shell ouvert jusqu'a qu'on ecrive exit. Juste avant on va tester le nombre d'arguments pour qu'il y en ait aucun car on en a pas besoin. Premiere chose que l'on va faire dans la loop, on lance `set_handler` pour que tout les signaux et masks soient prêts. On va aussi utiliser notre fonction `printGUI` pour avoir une interface. On recupere notre input avec `getInput` et on la parse avec `parseInput`. Un fois que l'on a nos `argv` et `argc` on test tout d'abord si l'utilisateur a juste appuyer sur **Enter** (`argc == 0`).

Une fois tout ca tester et recuperer on lance notre fonction `execbin`. Si tout va bien on lance notre programme builtin et on recommence la loop. Dans le cas ou la fonction nous renvoie `-1`, on `fork` pour creer un processus enfant.

### Code parent

Dans le code parent on va tester si le programme que l'on veut lancer est en background. Si ce n'est pas le cas on `waitpid` et on affiche le status de celui-ci. Dans le cas ou c'est une tache en fond, on donne juste le pid a notre variable `background_job`.

### Code enfant

On check si c'est une tache en fond, si c'est le cas, on enleve le derner caractere (&) et on redirige la sortie dans `/dev/null` et on cree un masque pour que `Ctrl+C` soit ignorer par l'enfant.

Dans tout les cas on utilise alors

```
execvp(_argv[0], _argv)
```

La fonction qui va permettre de lancer n'importe qu'elle programme non builtin (par exemple `ls`). On a un controle d'erreur qui va nous afficher si quelque chose s'est mal passer.

On `exit(EXIT_SUCCESS)`.