CHRISTOFOROU ANTHONY

RAPPORT

Dans ce TP, on cherchait a creer un shell fonctionnel comme celui que l'on peut trouver sur toute les distribution linux, par exemple: bash, zsh, ... L'idee etait de creer plusieurs modules qui s'occupent des commandes `builtin`, un autre pour le **parsing**, un pour les jobs en fond et enfin une fonction principale qui s'occupe de gerer le reste.

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 aver 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 tapper 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 charactere `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'executer** 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 equivalente.

Jobs

On va alors avoir le dernier module, qui va surtout etre pour les signaux mais qui va tout d'abord tester si le job que l'on veut executer sera en `background` ou pas. Pour ca 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 etape, on va gerer les signaux SIGINT et SIGHUP avec la fonction:

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

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

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

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

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

Enfin, on a la fonction

```
void set_handlers(void)
```

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

> On va donner a chaque signaux leurs handler precedents respectifs.
>
> - `child_process_signal` pour `SIGCHLD`,
> - `handler` pour `SIGINT` et `SIGUHP`
> - 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 prets. 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 charactere (&) 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)`.