

Systèmes d'Exploitation - Examen

12X009 - TP06

Noah Munz (19-815-489)

Département d'Informatique
Université de Genève

Mardi 31 Janvier 2023

Code: [Lien GitHub du code](#)

Rapport: [Lien GitHub du rapport](#)

Plan: [Lien GitHub du plan de l'implémentation](#)



Table of Contents

- 1 Rappel : But du TP
- 2 TADs & leurs relations
 - Décomposition modulaire
 - Structures de données utilisées
 - Décomposition fonctionnelle
- 3 Tests réalisés pour valider le fonctionnement du TP
- 4 Réponses aux questions (générales)



Table of Contents

- 1 Rappel : But du TP
- 2 TADs & leurs relations
 - Décomposition modulaire
 - Structures de données utilisées
 - Décomposition fonctionnelle
- 3 Tests réalisés pour valider le fonctionnement du TP
- 4 Réponses aux questions (générales)



Rappel : But du TP

Créer son propre Shell. C-à-d :

- Gérer 2 commandes “built-in” `cd` et `exit()` + exécuter des jobs
- Créer des processus avec `fork` pour ces jobs
- Gérer ces processus, notamment éviter les zombies et les orphelins
- Gérer les signaux envoyés au shell (en partie pour gérer zombies & orphelins)

Les principaux défis de ce TP sont :

- La taille du projet, les petites erreurs qui avant étaient “bénignes” voient leur impact grossir avec la taille du projet et le temps “d’utilisation” / test de ce dernier.
- La gestion de signaux qui peut rappeler une sorte de `try-catch` version C i.e. où le `catch` doit pouvoir être exécuté n’importe où et ne peut faire que certaines actions limitées (pas de `printf`, pas de passage de variable en argument, ne doit pas accéder aux variables globales pour être réentrant...)



Table of Contents

- 1 Rappel : But du TP
- 2 TADs & leurs relations
 - Décomposition modulaire
 - Structures de données utilisées
 - Décomposition fonctionnelle
- 3 Tests réalisés pour valider le fonctionnement du TP
- 4 Réponses aux questions (générales)



TADs & leurs relations: Décomposition modulaire

4 modules (+ `main.c`) ont été créés pour la réalisation du TP :

- ❶ Le fichier `main.c` qui contient la fonction `main` qui va être la première instruction à exécuter. (Contient juste une boucle infinie `for (;;) { ... }` qui va appeler les fonctions du module `shell` (notamment `sh_getAndResolveCmd()`) pour interpréter l'entrée utilisateur et exécuter les actions correspondantes.
- ❷ Le module `input` qui interprète l'entrée utilisateur, la parse, détermine si le job est à exécuter en foreground ou background... puis la passe au module `shell`
- ❸ Le module `shell`, le module principale de ce TP qui s'occupe de à peu près tout. Gestion de signaux, créations de processus avec `fork()`, gestion de ces processus, terminaisons "clean" en attendant ses enfants et où en les forçant à se terminer...



④ Le module `files` (repris des TP03-5)

S'occupe de la gestion fichiers, (existence, type, taille etc.), des path des fichiers (absolute path, concatenation...) et de la gestion des erreurs liées à ces opérations.

Ici seul `absPath()` et `concatPath()` ont été utilisées dans ce TP.

⑤ Le module `util` (repris des TP03-5)

Fonctions / macros servant divers usages allant de la gestion d'erreurs à la gestion de chaînes de caractères, en passant par des wrappers qui incluent lesdites fonctions de gestions d'erreurs.



Une structure de donnée (opaque) `Shell` a été implémenté pour garder plus facilement trace (*pid*) des tâches de fonds et de 1er plan en encapsulant le tout dans une structure.

La structure sert aussi à sauvegarder la dernière commande avec les `argv` / `argc` correspondants afin de pouvoir la relancer si jamais cela est nécessaire. (e.g. en combinaison de l'utilisation du flag `SA_RESTART` si une tâche de fond a été interrompu par un signal.)



TADs & leurs relations

Voici à quoi la structure ressemble :

`shell.c` :

```
struct Shell {  
    /** Contains the cwd (of Size PATH_MAX).  
     * Copy of current working directory as a field, to not having to refetch it everytime since  
     * 'getcwd()' copies the actual each time it is called */  
    char* crt_path;  
    // Pid of current process launched as foreground job  
    pid_t foreground_job;  
    // Pid of current process launched as background job  
    pid_t background_job;  
    // Number of current non-waited/terminated child  
    int child_number;  
  
    int old_fj_argc;  
    char*** old_fj; //pointer to argv of last foreground job  
    int old_bj_argc;  
    char*** old_bj; //pointer to argv of last background job  
};
```

Figure – Module `shell`, structure `Shell` (`shell.c`)



TADs & leurs relations: Décomposition fonctionnelle

Une fois “validée”, l’entrée utilisateur suit le “chemin” suivant :

- ❶ Comme dit précédemment, le module `input` se fait “utiliser” par `shell` (fonction `sh_getAndResolveCmd(Shell* sh)`) en passant l’entrée etc avec la fonction `readParseIn(int* argc, int* isForeground)`
- ❷ dans `sh_getAndResolveCmd` : on determine si la commande est built-in ou doit être exécuté en tant que job, puis appelle `cd()` , `exit_shell()` ou `execute_job(Shell*, char* cmd_name, int isForeground)`
- ❸ `execute_job` va gérer les forks, update les attributs de la struct (tenir compte du nombre d’enfants de leurs pids...), refuser le background job si on en a déjà un en cours, puis, si tout va bien, appelle la fonction `exec(Shell*, char* filename, char* argv[], int isForeground)` dans l’enfant créé. Si le job est à exécuter en foreground, le parent attend la mort de l’enfant avec `wait` (shell bloque).



TADs & leurs relations: Décomposition fonctionnelle

- ④ `exec` appelle `execvp(filename, argv)` avec les arguments qu'on lui a donné, puis désattribue les pids des foreground/background jobs de la struct shell (les mets à -2)
- ⑤ Quand un enfant meurt, il envoie un `SIGCHLD` à son processus parent, les signaux sont gérés avec les handlers définis avec `manage_signals()` et `hdl_sigint()`, `hdl_sighup()`, `hdl_sigchld()`.
`hdl_sigchld()` va `waitpid()` sur le child enregistré dans la struct qui a le pid correspondant puis va mettre ses attributs à jour. Elle s'occupe aussi de relancer les appels interrompu par signaux.
(i.e. `errno == EINTR`)



- ⑥ Pour quitter proprement le shell à plusieurs fonctions.

`clean_exit(Shell*, int exitCode, int forceExit)` et
`terminate_all_children(Shell*)`.

- ▶ `terminate_all_children()` est assez simple, elle va simplement `wait()` tant qu'il reste des enfants enregistré dans la struct.
- ▶ `clean_exit()` quant à elle est un peu plus complexe, elle va, en fonction de `forceExit`, attendre le back/foreground job, `SIGTERM` le background job, `SIGTERM` le foreground job ou les 2.

Ensuite, elle va appeler `terminate_all_children()`, puis `sh_free(Shell*)` puis finalement `exit(exitCode)` avec le code qu'on lui a passé en argument.



Table of Contents

- 1 Rappel : But du TP
- 2 TADs & leurs relations
 - Décomposition modulaire
 - Structures de données utilisées
 - Décomposition fonctionnelle
- 3 Tests réalisés pour valider le fonctionnement du TP
- 4 Réponses aux questions (générales)



Tests réalisés pour valider le fonctionnement du TP

Les tests réalisés ont été les suivants :

```
Pid: 10257
( /home/noahl/BA3/12X009-OS-TPs/TP06 )
|_ $ sleep 200 &
[1]      [10376] - sleep
```

```
( /home/noahl/BA3/12X009-OS-TPs/TP06 )
|_ $ sleep 200
```

```
Nb of child waiting to be terminated: 2
- job exited with exit code 1
```

```
Nb of child waiting to be terminated: 1
- job exited with exit code 1
```

Exiting with exit code 0.

```
(noahl@NoahMnz-Leg5Pro in ~/BA3/12X009-OS-TPs/TP06 on master x (origin/master)
```

```
$ (15:19:22) $ ps -f --forest
```

| UID | PID | PPID | C | STIME | TTY | TIME | CMD |
|-------|-------|------|---|-------|-------|----------|-------------------|
| noahl | 9816 | 2504 | 0 | 15:11 | pts/1 | 00:00:00 | sleep 501 |
| noahl | 2505 | 2504 | 0 | 14:36 | pts/1 | 00:00:00 | -zsh |
| noahl | 9414 | 2505 | 0 | 15:07 | pts/1 | 00:00:00 | _ nvim shell.c |
| noahl | 10410 | 2505 | 0 | 15:19 | pts/1 | 00:00:00 | _ ps -f --forest |

```
(noahl@NoahMnz-Leg5Pro in ~/BA3/12X009-OS-TPs/TP06 on master x (origin/master)
```

```
$ (15:19:32) $ █
```

```
( /home/noahl/BA3/12X009-OS-TPs/TP06 )
|_ $ kill -SIGHUP 10257
-Foreground job exited with exit code 0
```

```
( /home/noahl/BA3/12X009-OS-TPs/TP06 )
|_ $
```

Figure – Gestion de Zombies/orphelin, background jobs + Gestion de signaux.



Tests réalisés pour valider le fonctionnement du TP

```
( /home/noahl/BA3/12X009-OS-TPs/TP06 ) [15:24:55]
└─ $ sleep 5000 &
[1] [10868] - sleep

( /home/noahl/BA3/12X009-OS-TPs/TP06 ) [15:24:57]
└─ $ ^C
└─ $ -Background job exited with exit code 2

interrupted by signal, restarting background job...
[1] [10869] - sleep

└─ $ sleep 2 &
executeJob: Device or resource busy, background job (10869) is still running. Please wait for its completion o
r launch it as foreground job.
An error happened, while processing the command. Please try again.

( /home/noahl/BA3/12X009-OS-TPs/TP06 ) [15:25:09]
└─ $ ps --forest -f
UID      PID  PPID  C  STIME TTY      TIME CMD
noahl    2505  2504  0  14:36 pts/1    00:00:01 -zsh
noahl    9414  2505  0  15:07 pts/1    00:00:00 \_ nvim shell.c
noahl   10859  2505  0  15:24 pts/1    00:00:00 \_ ./shell
noahl   10869 10859  0  15:24 pts/1    00:00:00 \_ sleep 5000
noahl   10870 10859  0  15:25 pts/1    00:00:00 \_ ps --forest -f
-Foreground job exited with exit code 0

( /home/noahl/BA3/12X009-OS-TPs/TP06 ) [15:25:30]
└─ $ sleep 2
^C
└─ $ -Foreground job exited with exit code 2
```

Restart de processus en tâche de fond quand interrompu par signal & 1 seul BG job à la fois.



(/home/noahl/BA3/12X009-0S-TPs/TP06) [17:26:01]

```
l_ $ screenfetch

      ./.+o+-
      -yyyyyy+
      -yyyyyyo
      .+sss/`

      .++
      .:++o:
      o:+++:++.
      .:+++:o/.
      +oo+o:
      "oo+o
      "o++o      +++++.
      ++oo+:`    /odddhhh.
      .+..o+o:..`odddhhh+
      \.,+o+o:..`..:ohdhhhhh+
      ;o+++`ohhhhhhhhhyo++os:
      .o:`.syhhhhhhh/.oo++o`
      /osyyyyyyyo++ooo++/
      +oo++o\`
      oo++.
```

noahl@NoahMnz-Leg5Pro
OS: Ubuntu (on the Windows Subsystem for Linux)
Kernel: x86_64 Linux 5.10.102.1-microsoft-standard-WSL2
Uptime: 3h 20m
Packages: 1051
Shell: ThisDeserves-At-Least-a-6-;D
Resolution: 2560x1600
WM: Weston WM
GTK Theme: Adwaita [GTK3]
Disk: 1.3T / 3.4T (39%)
CPU: 12th Gen Intel Core i9-12900H @ 20x 2.918GHz
GPU: NVIDIA GeForce RTX 3070 Ti Laptop GPU
RAM: 796MiB / 15861MiB

-Foreground job exited with exit code 0

Pid: 28529

(/home/noahl/BA3/12X009-0S-TPs/TP06) [17:30:47]

```
l_ $ rm out -rf
-Foreground job exited with exit code 0
```

(/home/noahl/BA3/12X009-0S-TPs/TP06) [17:30:50]

```
l_ $ make
mkdir out
gcc -c main.c -lm -Wall -Wuninitialized -Wfloat-equal -Wconversion -Wunreachable-code -Wformat=2 -Winit-self
incompatible-pointer-types -g -DDEBUG -o out/main.o
gcc -c shell.c -lm -Wall -Wuninitialized -Wfloat-equal -Wconversion -Wunreachable-code -Wformat=2 -Winit-self
incompatible-pointer-types -g -DDEBUG -o out/shell.o
gcc -c input.c -lm -Wall -Wuninitialized -Wfloat-equal -Wconversion -Wunreachable-code -Wformat=2 -Winit-self
incompatible-pointer-types -g -DDEBUG -o out/input.o
gcc -c files.c -lm -Wall -Wuninitialized -Wfloat-equal -Wconversion -Wunreachable-code -Wformat=2 -Winit-self
incompatible-pointer-types -g -DDEBUG -o out/files.o
gcc -c util.c -lm -Wall -Wuninitialized -Wfloat-equal -Wconversion -Wunreachable-code -Wformat=2 -Winit-self
incompatible-pointer-types -g -DDEBUG -o out/util.o
gcc out/main.o out/shell.o out/input.o out/files.o out/util.o -o shell -lreadline
-Foreground job exited with exit code 0
```

(/home/noahl/BA3/12X009-0S-TPs/TP06) [17:30:52]

```
l_ $ ./shell
```

Pid: 28551

(/home/noahl/BA3/12X009-0S-TPs/TP06) [17:30:55]

```
l_ $ ps -f --forest

UID      PID  PPID  C  TIME TTY          TIME CMD
noahl    27964 27963  0 17:27 pts/2    00:00:00 -zsh
noahl    28529 27964  0 17:30 pts/2    00:00:00 \_ ./parent-shell
noahl    28551 28529  0 17:30 pts/2    00:00:00 \_ ./shell
noahl    28552 28551  0 17:30 pts/2    00:00:00 \_ ps -f --forest
-Foreground job exited with exit code 0
```



Tests réalisés pour valider le fonctionnement du TP

Où le but du dernier était de compiler le Shell dans le Shell puis le lancer toujours à partir du shell.

Aussi, pour tester SIGINT on a aussi lancé `tree /` (liste tous les fichiers depuis la racine formatté selon un arbre \Rightarrow job très long qui nous le laisse le temps de le stopper) puis appuyé sur `ctrl+c` pour voir si ça arrêtrait bien comme il le fallait.

file descriptors : Pour vérifier les redirections de `stdin` pour les background job, on a aussi utilisé les commandes `lsof -a -p <pid>` et `ls /proc/<pid>/fd -la` (où `<pid>` est bien le pid du background job.) La 2e est particulièrement pratique vu qu'elle nous montre tout simplement quoi est ouvert et pointe vers quoi.



Table of Contents

- 1 Rappel : But du TP
- 2 TADs & leurs relations
 - Décomposition modulaire
 - Structures de données utilisées
 - Décomposition fonctionnelle
- 3 Tests réalisés pour valider le fonctionnement du TP
- 4 Réponses aux questions (générales)



Réponses aux questions (générales)

Code : [Lien GitHub du code](#)

Rapport : [Lien GitHub du rapport](#)

Plan : [Lien GitHub du plan de l'implémentation](#)

