Compte-rendu de TP Shell

Implémentation d'un Mini-Shell Unix

Systèmes & Réseaux

Syrine BEN HASSINE Thibault GAILLARD

G1 — L3 Informatique Générale — S6

Février 2026

Table des matières

1 Principales réalisations **2**
- 1.1 Architecture générale . . . 2
- 1.2 Étapes réalisées . . . 2
  - 1.2.1 Exécution de commandes simples (étapes 3 et 4) . . . 2
  - 1.2.2 Gestion des erreurs (étape 5) . . . 2
  - 1.2.3 Pipelines (étapes 6 et 7) . . . 2
  - 1.2.4 Arrière-plan (étape 8) . . . 2
  - 1.2.5 Module de gestion des jobs (section 5.2) . . . 2
  - 1.2.6 Commandes intégrées . . . 3
- 1.3 Points techniques importants . . . 3

2 Description des tests effectués **4**
- 2.1 Tests manuels . . . 4
- 2.2 Tests automatisés avec `sdriver.pl` . . . 5
- 2.3 Résultats observés . . . 6

3 Difficultés rencontrées **6**

Conclusion **6**

1. Principales réalisations

1.1. Architecture générale

Le projet est structuré en deux modules principaux :

— `shell.c` — boucle principale du shell, exécution des commandes, gestion des signaux et des commandes intégrées (*built-ins*).

— `jobs.c` / `jobs.h` — module dédié à la représentation et à la manipulation des travaux (jobs).

— Les fichiers fournis `readcmd.c` et `csapp.c` ont été réutilisés sans modification.

1.2. Étapes réalisées

1.2.1. Exécution de commandes simples (étapes 3 et 4)

Le shell exécute les commandes saisies par l'utilisateur via un `fork()` suivi d'un `execvp()` dans le processus fils. Les redirections d'entrée (`<`) et de sortie (`>`) sont gérées par `open()` et `dup2()` avant l'appel à `execvp()`.

1.2.2. Gestion des erreurs (étape 5)

Les cas d'erreur suivants sont détectés et signalés à l'utilisateur :

— Commande introuvable (`ENOENT`) : message `command not found`.

— Permission refusée (`EACCES`).

— Erreur de syntaxe dans la ligne de commande.

— Fichier de redirection inexistant ou inaccessible : message `No such file or directory`.

— Chemin invalide (`ENOTDIR`) : message `Not a directory`.

1.2.3. Pipelines (étapes 6 et 7)

Les pipelines de N commandes (`cmd1 | cmd2 | ... | cmdN`) sont pris en charge. Le shell crée $N - 1$ pipes avec `pipe()`, puis fork N fils. Chaque fils ferme les descripteurs inutiles, redirige son entrée/sortie vers les bons bouts de pipe, puis appelle `execvp()`.

1.2.4. Arrière-plan (étape 8)

Une commande terminée par `&` est lancée en arrière-plan. Le père n'attend pas sa terminaison et affiche immédiatement `[jid] pid`.

1.2.5. Module de gestion des jobs (section 5.2)

Un tableau de structures `job_t` (de taille `MAXJOBS = 10`) représente l'ensemble des travaux actifs. Chaque entrée contient :

Champ	Type	Rôle
`jid`	`int`	Numéro de job (> 0), 0 = case libre
`pid`	`pid_t`	PID du processus leader
`pgid`	`pid_t`	PGID du groupe de processus
`state`	`job_state`	`FG`, `RUNNING`, `STOPPED`
`cmd`	`char[]`	Ligne de commande (pour affichage)

Les fonctions d'accès implémentées sont : `init_jobs()`, `add_job()`, `delete_job_by_pid()`, `delete_job_by_jid()`, `get_job_by_pid()`, `get_job_by_jid()`, `get_fg_job()`, `get_job_by_id_str()` et `list_jobs()`.

1.2.6. Commandes intégrées

Les built-ins suivants ont été implémentés :

- `quit` / `q` — quitte le shell.
- `jobs` — liste tous les jobs actifs avec leur état et PID.
- `fg %jid` ou `fg pid` — envoie `SIGCONT` au job et le place au premier plan.
- `bg %jid` ou `bg pid` — envoie `SIGCONT` au job et le fait s'exécuter en arrière-plan.
- `stop %jid` ou `stop pid` — envoie `SIGTSTP` au groupe de processus du job.

1.3. Points techniques importants

Point 1 — Attente du processus de premier plan — Au lieu d'appeler `waitpid()` dans la boucle principale, le shell attend la fin du job de premier plan via une boucle :

`wait_fg_job()`

```
static void wait_fg_job(void) {
    while (get_fg_job() != NULL) {
        sleep(1);
    }
}
```

Point 2 — `waitpid` uniquement dans le handler SIGCHLD — Tout ramassage de processus zombie est effectué exclusivement dans `sigchld_handler()`, qui utilise `waitpid(-1, &status, WNOHANG | WUNTRACED)` en boucle.

Point 3 — Exclusion mutuelle — Avant toute modification du tableau `jobs[]`, le signal `SIGCHLD` est masqué pour éviter qu'un handler concurrent ne corrompe les structures :

Masquage de SIGCHLD

```
sigset_t prev;
block_sigchld(&prev);
/* ... modification de jobs[] ... */
unblock_sigchld(&prev);
```

Point 4 — Réinitialisation des signaux dans les fils — Chaque fils réinitialise son masque de signaux et remet les handlers à `SIG_DFL` avant `execvp()` :

`reset_signals_in_child()`

```
static void reset_signals_in_child(void) {
    sigset_t empty;
    sigemptyset(&empty);
    sigprocmask(SIG_SETMASK, &empty, NULL);
    signal(SIGCHLD, SIG_DFL);
    signal(SIGINT,  SIG_DFL);
    signal(SIGTSTP, SIG_DFL);
}
```

Groupes de processus — Chaque commande crée son propre groupe via `setpgid(0, 0)` dans le fils et `setpgid(pid, pid)` dans le père. Les signaux sont envoyés au groupe entier via `kill(-pgid, sig)`.

2. Description des tests effectués

Les tests ont été réalisés de deux façons : manuellement en lançant `./shell` dans un terminal, et automatiquement via le script `sdriver.pl`.

2.1. Tests manuels

Commande simple
`echo hello world` → affichage correct.

Commande inconnue
`commandeinexistante` → message `command not found`.

Redirection entrée
`cat < /etc/hostname` → contenu du fichier affiché.

Redirection sortie
`echo test > /tmp/out.txt` → fichier créé et lu correctement.

Fichier inexistant
`cat < /tmp/inexistant` → message `No such file or directory`, shell continue.

Pipeline simple
`ls /bin | grep cat` → résultat filtré correct.

Pipeline chaîné
`cat /etc/passwd | grep root | wc -l` → compte correct.

Arrière-plan
`sleep 10 &` → affichage `[1] pid`, prompt immédiatement rendu.

Commande `jobs`
Après plusieurs `sleep &` → liste correcte avec jid, pid, état et commande.

Commande `stop`
`stop %1` → job passe à l'état `Stopped`.

Commande `bg`
Après `stop`, `bg %1` → job repasse à `Running`.

Commande `fg`
`fg %1` → shell bloque jusqu'à la fin ou suspension du job.

Ctrl+Z premier plan
`sleep 30` puis `Ctrl+Z` → job suspendu, affiché dans `jobs`.

Quit `quit` → message `exit prog shell` et terminaison propre.

EOF `Ctrl+D` → message `exit` et terminaison.

2.2. Tests automatisés avec `sdriver.pl`

Terminal

```
./sdriver.pl -t <fichier_trace> -s ./shell -v
```

Emplacement des fichiers de tests

Les fichiers de trace sont disponibles dans le dossier `./tests_ajoutes/`.

Fichier	Ce qui est vérifié
`trace01.txt`	Commande simple (`echo`)
`trace02.txt`	Commande inconnue → `command not found`
`trace03.txt`	Built-in `quit` → terminaison propre
`trace04.txt`	Redirection d'entrée (`<`)
`trace05.txt`	Redirection de sortie (`>`)
`trace06.txt`	Pipe simple (2 commandes)
`trace07.txt`	Pipe chaîné (3 commandes)
`trace08.txt`	Arrière-plan (`&`) : affichage `[n] pid`
`trace09.txt`	Signal `INT` : le shell survit
`trace10.txt`	Redirection depuis fichier inexistant + continuation
`trace11.txt`	`CLOSE` (EOF) → terminaison propre
`trace12.txt`	Pipeline avec redirection de sortie combinés
`trace_jobs01.txt`	`jobs` liste les jobs en arrière-plan
`trace_jobs02.txt`	`stop` puis `bg` : transitions d'état
`trace_jobs03.txt`	`TSTP` suspend le job de premier plan
`trace_jobs04.txt`	`stop` puis `fg` reprend le job
`trace_jobs05.txt`	Désignation d'un job par son PID

2.3. Résultats observés

Bilan des tests

L'ensemble des tests manuels et automatisés se déroule sans erreur.

En particulier :

— Les processus zombies sont correctement ramassés dans le handler `SIGCHLD`.

— L'exclusion mutuelle sur `jobs[]` empêche toute corruption lors de l'arrivée concurrente d'un signal.

— Les signaux `SIGINT` et `SIGTSTP` n'affectent pas le shell, uniquement les processus de premier plan.

— Le prompt `shell>` est réaffiché correctement après un message asynchrone du handler.

3. Difficultés rencontrées

— **Race condition fork/setpgid** : le père et le fils doivent tous deux appeler `setpgid()` pour éviter qu'un signal n'arrive avant que le groupe soit créé.

— **Affichage du prompt** : le handler `SIGCHLD` affichant un message asynchrone, il est nécessaire de réafficher `shell>` et d'appeler `fflush(stdout)`.

— **Concurrence waitpid** : supprimer tous les appels à `waitpid()` hors du handler a nécessité de repenser l'attente du premier plan (boucle `sleep(1)`).

Conclusion

Le mini-shell implémenté reproduit les fonctionnalités essentielles des shells Unix courants : exécution de commandes, redirections, pipelines, gestion de l'arrière-plan et des signaux, ainsi qu'une gestion complète des travaux avec les commandes `jobs`, `fg`, `bg` et `stop`. Les quatre contraintes techniques du sujet ont été respectées.