

TD4 — Fork & tubes non nommés (pipe)

Objectif: maîtriser `pipe(2)` + `fork(2)` + `dup2(2)` + `exec*(2/3)` pour faire (1) une communication parent/enfant, (2) éviter les blocages/EOF, (3) implémenter un pipeline type shell (`cmd1 | cmd2 | ...`).

1) Rappel: `pipe(2)`

```
int pipefd[2];
pipe(pipefd);
// pipefd[0] = READ end
// pipefd[1] = WRITE end
```

Propriétés:

- Un pipe est un buffer noyau.
- `read` bloque tant qu'il n'y a pas de données **et** qu'au moins un writer est ouvert.
- `read` retourne `0` (EOF) quand **tous** les writers ont fermé la fin écriture.
- `write` peut bloquer si le buffer du pipe est plein.

Dans vos fichiers, vous utilisez l'enum:

```
enum { READ, WRITE };
```

2) Exo 1/2: parent écrit, enfant lit (et pourquoi ça bloque parfois)

Syscalls/fonctions utilisées

- `pipe(2)`, `fork(2)`, `read(2)`, `write(2)`, `close(2)`
- `fprintf(3)` sur `stderr` pour les logs

Règle d'or (anti-blocage)

Après le `fork()` :

- le **parent** doit fermer `pipefd[READ]`
- l'**enfant** doit fermer `pipefd[WRITE]`

Sinon:

- si l'enfant garde un writer ouvert (même sans écrire), le `read()` peut ne **jamais** voir EOF → boucle infinie.
- si le parent garde un reader ouvert, ça ne casse pas EOF mais c'est sale (et peut compliquer le debug).

Dans vos solutions

- [01-pipe.c](#) : parent écrit 10x, enfant lit jusqu'à EOF.
- [02-pipe.c](#) : le nombre d'écritures est paramétrable (`count`).

Point TP: l'enfant lit “au plus 500 octets” à la fois → si le parent écrit plus, l'enfant doit faire plusieurs `read()`.

3) Statut de compléTION en tâche de fond (`./prog &`)

Quand tu lances `./mypipe xx &`, le shell affiche ensuite un *status*.

- Si le parent et l'enfant terminent normalement et que tu ne fais pas d'erreur, le job “réussit”.
 - Attention: si tu ne `wait()` pas tes enfants et que tu n'ignores pas `SIGCHLD`, tu peux créer des **zombies** (dans un TP court, ça se voit via `ps`).
-

4) Exo 3: transformer en majuscules (pipe + exec)

Idée

- Le parent ne produit plus les données à la main.
- Il redirige sa `stdout` vers le pipe (`dup2`) puis lance une commande (ex: `ps`) via `execvp`.
- L'enfant lit le pipe en continu et transforme (ici caractère par caractère).

Dans votre [03-upper.c](#):

- Child: `read(..., 1)` → convertit en majuscules → `write(1, ...)`
- Parent: `dup2(tube[WRITE], STDOUT_FILENO)` puis `execvp("ps", ...)`

Points d'attention

- `execvp` signature: `execvp(file, arg0, arg1, ..., NULL)`
 - `arg0` doit exister (souvent = nom de commande)
 - Après un `exec*` réussi: le code après n'est pas exécuté.
 - En cas d'échec de `exec*`: afficher `perror` puis terminer avec `_exit(127)`.
-

5) Pipeline type shell: cmd1 | cmd2 | ... | cmdN

Plan général (N commandes)

Pour N commandes, on crée N-1 pipes.

Pour la commande i (0-indexée):

- si `i > 0`: son `stdin` vient du pipe `i-1` (`dup2 read-end → STDIN_FILENO`)
- si `i < N-1`: son `stdout` va vers le pipe `i` (`dup2 write-end → STDOUT_FILENO`)

Dans vos implémentations [04-pipeline.c](#) et [05-final.c](#):

- création de tous les pipes avant les `fork()`
- dans chaque child: `dup2` selon sa position, puis **fermeture de tous les fd de pipes**
- parent: ferme aussi tous les fd de pipes, puis `waitpid` sur tous

La règle qui évite 90% des bugs

Après avoir fait les `dup2`, **fermer tous les fd de pipes** (read+write) dans le child.

Sinon:

- EOF jamais atteint (un writer "reste ouvert" quelque part)
 - blocages difficiles à diagnostiquer
-

6) Propager le status du dernier processus

But du sujet: le programme pipeline doit se terminer juste après le dernier processus du pipeline, et:

- si le dernier sort normalement: retourner son `exit code`
- sinon: afficher la raison (signal) et retourner `EXIT_FAILURE`

Macros `wait` à connaître

- `WIFEXITED(status)` / `WEXITSTATUS(status)`
- `WIFSIGNALED(status)` / `WTERMSIG(status)`

Vos codes font bien:

- `waitpid(pids[last], &last_status, 0)`
 - puis `WIFEXITED` / `WIFSIGNALED` avec message via `signame()`
-

7) Parsing `cmd -- cmd -- cmd` (travail perso)

Dans [05-final.c](#):

- la ligne de commande est découpée avec `--` comme séparateur
- la fonction utilitaire `split_args` (dans `utils.c`) aide à séparer `args1` et `args2`

Exemple attendu:

```
./pipeline ls -l -- grep "\\.c$" -- wc -l
```

Equivalent shell:

```
ls -l | grep "\\.c$" | wc -l
```

Points d'attention:

- toujours terminer chaque commande par `NULL` (format `execvp`)
 - en cas de parse invalide (commande vide / `--` mal placé): afficher usage + `EXIT_FAILURE`
-

8) Pages de man à connaître

- `pipe(2)`, `fork(2)`, `dup2(2)`, `close(2)`
- `read(2)`, `write(2)`
- `execve(2)`, `execvp(3)`, `execlp(3)`

- `wait(2)` , `waitpid(2)`