TD3 — Bufferisation (libc) & Fork/Exec/Redirection

Objectif: comprendre la différence **syscalls vs libc**, les **tampons stdio** (`printf/fprintf/...`), et les patrons **fork/exec + redirection + wait**.

---

1) Syscall ou libc ? (indice: prototypes + `man`)

Règle pratique:

- **Syscall**: manipule des **fd** (`int fd`), docs en `man 2 ...`
- **Libc (stdio)**: manipule des **flux** `FILE *` , docs en `man 3 ...`

Table (cohérent avec `03-buffer_fork/01-syscall.txt`):

Fonction	Nature	Indice prototype	Notes
`printf()`	libc	`int printf(const char*, ...)`	écrit via stdio (tampon)
`fopen()`	libc	`FILE *fopen(const char*, const char*)`	utilise `open(2)` en interne
`fclose()`	libc	`int fclose(FILE*)`	flush + close
`fread()`	libc	`size_t fread(void*,size_t,size_t,FILE*)`	utilise `read(2)` en interne
`open()`	syscall	`int open(const char*, int, ...)`	crée un fd
`close()`	syscall	`int close(int fd)`	ferme un fd
`lseek()`	syscall	`off_t lseek(int fd, off_t, int)`	déplace l’offset noyau
`rewind()`	libc	`void rewind(FILE*)`	équiv. `fseek(stream,0,SEEK_SET)`
`write()`	syscall	`ssize_t write(int,const void*,size_t)`	pas de tampon “visible” côté user
`exit()`	libc	`void exit(int)`	flush stdio + handlers `atexit()`
`_exit()`	syscall	`void _exit(int)`	termine **sans** flush stdio

---

2) Buffers stdio (libc) : les 3 modes

Sur `stdout` / `stderr`, la libc peut bufferiser.

Les 3 modes (classiques)

- **Unbuffered** (`_IONBF`): écriture immédiate (souvent `stderr`)
- **Line buffered** (`_IOLBF`): flush à chaque `\n` (souvent `stdout` vers un terminal)
- **Fully buffered** (`_IOFBF`): flush quand tampon plein / `fflush` / `exit` (souvent `stdout` vers fichier/pipe)

Différence cruciale

- `write(2)` écrit côté noyau **tout de suite** (pas de tampon stdio).
- `fprintf/printf` écrivent dans un tampon **en userland**, puis plus tard.

Tester (sujet)

Deux exécutions:

- Terminal direct:

```
./02-buffer
```

- `stdout` / `stderr` passent par un pipe:

```
./02-buffer 2>&1 | cat -u
```

Compilations (macros):

```
rm -f 02-buffer && make 02-buffer CFLAGS+="-DUTILISER_FPRINTF1"
rm -f 02-buffer && make 02-buffer CFLAGS+="-DUTILISER_FPRINTF2 -DUTILISER_SORTIE_ERREUR2"
rm -f 02-buffer && make 02-buffer CFLAGS+="-DUTILISER_FPRINTF1 -DUTILISER_FPRINTF2 -DUTILISER
```

Pourquoi ajouter `\n` change tout ?

En mode *line buffered* (`stdout` sur terminal), un `\n` déclenche un flush → la première chaîne apparaît avant le `sleep(1)` .

`exit()` vs `_exit()` (point clé)

- `exit()` **flush** les flux stdio (`stdout` , `stderr` , fichiers ouverts via `fopen`) puis termine.
- `_exit()` termine **immédiatement**, donc un `fprintf(stdout, ...)` non flush peut **ne jamais apparaître**.

---

3) `fork(2)` + stdio : piège du buffer dupliqué

`fork()` duplique la mémoire du processus (copy-on-write), donc aussi les **buffers stdio**.

Cas typique demandé au TP:

1. `printf("%d", getpid())` **sans** `\n` (reste dans le buffer)
2. `fork()`
3. si les deux processus finissent par `exit()`, ils peuvent **flusher deux fois le même tampon** → PID affiché en double.

3 solutions possibles (à connaître)

1. `fflush(stdout);` juste avant `fork()` (solution simple)
2. Écrire le premier affichage avec `write(2)` (pas de tampon stdio)
3. Désactiver/adapter le buffering: `setvbuf(stdout, NULL, _IONBF, 0)` ou écrire sur `stderr` (souvent unbuffered)

---

4) Patron `fork()` : qui affiche quoi ?

Rappels:

- Retour de `fork()`:
  - `-1` erreur
  - `0` dans l'enfant
  - `>0` (PID enfant) dans le parent

Dans votre `03-fork.c`, vous:

- ignorez `SIGCHLD` (`signal(SIGCHLD, SIG_IGN)`) pour éviter les zombies **(attention: pas toujours recommandé en prod, mais OK pour TP)**
- faites `fflush(stdout)` avant `_exit()` côté enfant

Tip examen: si vous utilisez `exec*` dans l'enfant, pensez à appeler `_exit(127)` en cas d'échec de `exec` (sinon le child continue).

---

5) Redirection + `execvp` (mini-shell `./run sortie cmd args...`)

Objectif équivalent shell:

```
./commande arg1 arg2 ... > sortie
```

Étapes (pattern standard)

1. (optionnel) Afficher la commande (vous le faites avec `write`)
2. `pid = fork()`
3. Dans l'enfant:
   - ouvrir le fichier de sortie `open(sortie, O_WRONLY|O_CREAT|O_TRUNC, 0644)`
   - `dup2(fd, STDOUT_FILENO)`
   - `close(fd)`
   - `execvp(cmd, argv_cmd)`
4. Dans le parent:
   - `waitpid(pid, &status, 0)`
   - interpréter `status`

`wait` / statut de fin

Macros à connaître:

- `WIFEXITED(status)` + `WEXITSTATUS(status)`
- `WIFSIGNALED(status)` + `WTERMSIG(status)`
- (optionnel) `WCOREDUMP(status)`

Tester les erreurs (segfault)

Dans l'enfant, avant `execvp`, appeler:

```
static void gen_segfault(void) { *((int*)0) = 42; }
```

Le parent doit détecter que le process a fini par signal (pas "exit code normal").

---

6) Pages de man à connaître

- `fork(2)`, `execve(2)` / `execvp(3)`, `wait(2)` / `waitpid(2)`
- `dup2(2)`, `open(2)`, `close(2)`
- `stdio(3)`, `setvbuf(3)`, `fflush(3)`
- `exit(3)`, `_exit(2)`