TD2 — Read/Write (suite) : `fstat/open/close` + FIFOs + `dup` + répertoires

Objectif: manipuler de vrais fichiers et descripteurs (`open/close`), inspecter le type d'un fichier (`stat/fstat/lstat`), comprendre les **FIFOs nommées** (chat), et observer l'aliasing des descripteurs (`dup`).

---

1) Types de fichiers avec `stat(2)` / `fstat(2)` / `lstat(2)`

Syscalls / fonctions

- `stat(2)` : récupère les infos d'un **chemin**
- `lstat(2)` : comme `stat` , mais ne déréférence pas un **lien symbolique**
- `fstat(2)` : récupère les infos d'un **fd** (ex: `STDIN_FILENO`)

La structure `struct stat` contient `st_mode` (type + permissions).

Macros recommandées (POSIX)

Au lieu de tester `st_mode & S_IFMT` à la main, on utilise:

- `S_ISREG(m)` fichier régulier
- `S_ISDIR(m)` répertoire
- `S_ISLNK(m)` lien symbolique
- `S_ISCHR(m)` char device
- `S_ISBLK(m)` block device
- `S_ISFIFO(m)` FIFO/pipe
- `S_ISSOCK(m)` socket

Template: afficher le type d’un fd

```c
#include <sys/stat.h>
#include <unistd.h>

static char filetype_from_mode(mode_t m) {
    if (S_ISREG(m))  return '-';
    if (S_ISDIR(m))  return 'd';
    if (S_ISLNK(m))  return 'l';
    if (S_ISCHR(m))  return 'c';
    if (S_ISBLK(m))  return 'b';
    if (S_ISFIFO(m)) return 'p';
    if (S_ISSOCK(m)) return 's';
    return '?';
}

int main(void) {
    struct stat st;
    fstat(STDIN_FILENO, &st);
    char t = filetype_from_mode(st.st_mode);
    /* ... */
}
```

Ce que tu observes (ex `01-file_type.c`)

- `./01-file_type` → stdin = terminal → **character device**
- `./01-file_type < /etc` → stdin = répertoire (ouverture via redirection) → **directory**
- `./01-file_type < ./01-file_type` → stdin = fichier régulier → **regular file**

---

2) FIFOs nommées (pipes nommés) : chat mono-directionnel

Une FIFO est un objet du système de fichiers (un “nom”) qui référence un **tube noyau** : les données transitent via le noyau, pas comme un fichier sur disque.

Syscalls clés

- `mkfifo(3)` (wrapper libc) / `mkfifo(2)` selon systèmes → crée le fichier FIFO
- `open(2)` avec `O_RDONLY` ou `O_WRONLY`
- `read(2)`, `write(2)`, `close(2)`

Propriété importante: `open()` peut bloquer

- `open(fifo, O_WRONLY)` bloque tant qu'il n'y a **pas** de lecteur ouvert.
- `open(fifo, O_RDONLY)` peut bloquer tant qu'il n'y a **pas** d'écrivain (selon options).
- Avec `O_NONBLOCK`, le comportement change (à connaître, mais pas requis ici).

Pourquoi ça marche en ssh mais pas entre deux machines ?

- Une FIFO est **locale au système de fichiers/host**.
- Si tu lances le client sur ton PC et le serveur sur `ssh.enseirb-matmeca.fr`, tu ne partages pas le même fichier `versserveur`.
- En te connectant en ssh et en exécutant les deux programmes sur **la même machine**, ils voient la même FIFO.

---

3) Fonction de transfert robuste (gestion écriture partielle)

Le sujet insiste sur un point crucial: `write(2)` peut écrire **moins** que demandé.

Template (identique à votre code `transfer()`):

```
#define BUFFER_SIZE 256

static void transfer(int fd_in, int fd_out) {
    char buffer[BUFFER_SIZE];
    for (;;) {
        ssize_t len = read(fd_in, buffer, BUFFER_SIZE);
        exit_if(len == -1, "read");
        if (len == 0) return; // EOF

        ssize_t wrote = 0;
        while (wrote < len) {
            ssize_t rc = write(fd_out, buffer + wrote, (size_t)(len - wrote));
            exit_if(rc == -1, "write");
            wrote += rc;
        }
    }
}
```

À retenir:

- `read == 0` → EOF
- boucle de `write` obligatoire si protocole/pipe/fifo

---

4) `open(2)` / `close(2)` (et modes)

`open`

```
#include <fcntl.h>
int fd = open("donnees.txt", O_RDONLY);
```

Si on crée un fichier: `open(path, O_CREAT|O_WRONLY|O_TRUNC, 0644)`.

`close`

Toujours fermer les fd quand terminé.

---

5) Aliasing: `open` vs `dup`

Concept noyau:

- Un fd pointe vers une **open file description** (incluant l'offset courant).

`open()` deux fois

- Deux open file descriptions différentes
- Offsets **indépendants**

`dup()`

- Deux fd **partagent** la même open file description
- Donc offset **partagé**

Expérience attendue (à faire pour voir la différence)

Fichier `donnees.txt` = `abcdefgh`

Pseudo-code:

```c
int fd = open("donnees.txt", O_RDONLY);
char a[4], b[4];
read(fd, a, 4);

int fd2 = /* open(...) OU dup(fd) */;
read(fd2, b, 4);
```

- Si `fd2 = open(...)` → `a=abcd`, `b=abcd`
- Si `fd2 = dup(fd)` → `a=abcd`, `b=efgh`

Remarque: vos fichiers `03-alias_open.c` et `03-alias_dup.c` montrent l'idée, mais pour observer la différence "offset partagé vs pas partagé", il faut faire un premier `read()` sur `fd` avant le `read()` sur `fd2`.

---

6) Lire un répertoire (mini `ls`)

API utilisée (libc):

- `opendir(3)`, `readdir(3)`, `closedir(3)`

`struct dirent` et `d_type`

Dans `04-myls.c`, tu utilises `entry->d_type` pour deviner le type.

- Avantage: simple.
- Limite importante: `d_type` peut valoir `DT_UNKNOWN` sur certains systèmes/FS.

Approche robuste (souvent attendue en prog système):

- construire un chemin `dir + "/" + d_name`
- faire `lstat(2)` / `stat(2)` pour obtenir le type via `st_mode`.

Template: itérer sur 1..N répertoires

- Si aucun argument: traiter `.`
- Sinon: traiter chaque `av[i]`

---

7) Questions “Pour aller plus loin” (idées rapides)

Pourquoi 2 FIFOs pour un chat à deux ?

- Une FIFO est fondamentalement **unidirectionnelle** pour un échange propre.
- Pour du **bidirectionnel simultané**, on utilise 2 canaux:
  - `versserveur` (client → serveur)
  - `versclient` (serveur → client)

Chat à N (principe)

- Si plusieurs clients lisent la **même** FIFO, chaque message est consommé par le premier `read()` qui passe.
- Solution classique: serveur “hub” + **une FIFO de sortie par client** (ou sockets).

Éviter qu’un client voie son propre message

- Ajouter un octet “id client” (ou PID) au début du message.
- Le client ignore les messages dont l’id == son id.

---

8) Pages de man à connaître

- `open(2)`, `close(2)`, `read(2)`, `write(2)`
- `stat(2)`, `fstat(2)`, `lstat(2)`, `inode(7)`
- `mkfifo(3)`
- `opendir(3)`, `readdir(3)`, `closedir(3)`
- (options) `getopt(3)`