TD1 — Read/Write (programmation système C)

Objectif: lire/écrire **en binaire** correctement sur des *flux de fichiers* (fd), comprendre les **syscalls** `read(2)` / `write(2)` et les pièges (EOF, erreurs, lecture/écriture partielles, padding des structures).

---

1) `man` et sections

- `man -k mot` (aka *apropos*) cherche dans toutes les sections.
- Sections utiles:
  - `1` : commandes utilisateur (ex: `od(1)`, `objdump(1)`)
  - `2` : **appels systèmes** (ex: `read(2)`, `write(2)`)
  - `3` : fonctions de bibliothèque (libc) (ex: `printf(3)`, `perror(3)`)
  - `3p` / `3posix` : interfaces **POSIX** (spécif), parfois différentes/plus normatives que les pages GNU.

Q: `printf` est dans quelle section ?

- `printf(3)` (fonction libc, pas un syscall).

---

2) Syscall vs libc

- **Syscall**: bascule noyau via instruction CPU (ex: `syscall` en x86_64). Interface documentée en section `2`.
- **Libc**: fonctions en espace utilisateur (section `3`) qui peuvent:
  - appeler des syscalls (souvent via wrappers)
  - ou être purement “userland”.

Q: `perror()` est-elle un appel système ? Pourquoi ?

- Non: `perror(3)` est une fonction libc.
- Elle lit `errno` (thread-local) et écrit un message sur `stderr` (souvent via `write(2)` en interne).

---

3) File descriptors (fd) et redirections

- `STDIN_FILENO = 0`, `STDOUT_FILENO = 1`, `STDERR_FILENO = 2`.
- Redirections shell (zsh/bash):
  - `./prog < in` (redirige stdin)
  - `./prog > out` (redirige stdout)
  - `2> err` (redirige stderr)

Test d'erreur demandé (écriture sur un répertoire):

```
./03-echo blabla < . 1>&0
```

Idée: la redirection rend `stdout` invalide (ou non-écrivable) selon le montage, et `write` doit échouer.

---

4) `write(2)` — points essentiels

Signature:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- Retour:
  - `>= 0`: nb d'octets réellement écrits
  - `-1`: erreur, `errno` est positionné
- **Écriture partielle** possible (pipes, sockets, terminaux, fd non-bloquants, signaux…).

Template: `write_all` (robuste)

```c
#include <errno.h>
#include <unistd.h>

static int write_all(int fd, const void *buf, size_t n) {
    const char *p = (const char *)buf;
    while (n > 0) {
        ssize_t w = write(fd, p, n);
        if (w < 0) {
            if (errno == EINTR) continue; // interrompu par signal
            return -1;
        }
        p += (size_t)w;
        n -= (size_t)w;
    }
    return 0;
}
```

Exo: `echo` (syscall-only)

Dans votre `03-echo.c`, vous écrivez chaque argument vers `stdout`.

- Variante “propre”: utiliser `write_all(STDOUT_FILENO, av[i], strlen(av[i]))` + `"\n"`.

---

5) `read(2)` — points essentiels

Signature:

```c
ssize_t read(int fd, void *buf, size_t count);
```

- Retour:
  - `> 0` : nb d’octets lus
  - `0` : **EOF** (fin de fichier / pipe fermé)
  - `-1` : erreur, `errno` positionné
- Lecture partielle possible.
- `Ctrl-D` dans un terminal: envoie EOF si le buffer de ligne est vide.

Template: boucle de copie (stdin → stdout)

Votre `06-read.c` fait la boucle classique:

```c
#define BUFFER_SIZE 512
char buf[BUFFER_SIZE];

for (;;) {
    ssize_t r = read(STDIN_FILENO, buf, BUFFER_SIZE);
    if (r < 0) {
        if (errno == EINTR) continue;
        /* erreur */
        break;
    }
    if (r == 0) break;            // EOF
    if (write_all(STDOUT_FILENO, buf, (size_t)r) < 0) {
        /* erreur */
        break;
    }
}
```

Tests typiques

```
./06-read
./06-read < input_file
./06-read < input_file > output_file
./06-read < .
```

---

6) Lire/écrire un entier en binaire

Écrire un `long` “tel quel” en mémoire:

```c
long x = 5;
write(STDOUT_FILENO, &x, sizeof x);
```

Lire symétriquement:

```c
long x;
ssize_t r = read(STDIN_FILENO, &x, sizeof x);
/* vérifier r == sizeof x */
```

Pourquoi `./04-write_number` affiche “du bruit” ?

- Parce que vous envoyez des octets binaires sur un terminal.

Observer avec `od(1)`

```
./04-write_number | od -x
```

- Vous voyez l’endianess (souvent **little-endian**): pour `5`, les octets faibles arrivent d’abord.

Template: `read_exact` (robuste)

Utile pour protocoles binaires:

```c
#include <errno.h>
#include <unistd.h>

static int read_exact(int fd, void *buf, size_t n) {
    char *p = (char *)buf;
    size_t off = 0;
    while (off < n) {
        ssize_t r = read(fd, p + off, n - off);
        if (r < 0) {
            if (errno == EINTR) continue;
            return -1;
        }
        if (r == 0) return 1; // EOF avant d’avoir tout lu
        off += (size_t)r;
    }
    return 0;
}
```

---

7) Lire/écrire une structure: padding & alignement

Structure du sujet:

```
struct nopad {
    char c1;
    long l;
    char c2;
};
```

Taille: pourquoi ce n'est pas `1 + 8 + 1 = 10` ?

- Le compilateur insère du **padding** pour respecter l'**alignement**.
- Sur x86_64 typique:
  - `long` fait 8 octets et est aligné sur 8
  - donc après `char c1`, il y a 7 octets de padding avant `long`
  - et souvent du padding final pour que `sizeof(struct)` soit multiple de l'alignement max.

Dans votre correction, la struct est réordonnée dans `utils.h`:

```
struct nopad {
    long l;
    char c1;
    char c2;
};
```

- Ce simple réordonnancement réduit en général la taille (ex: 16 au lieu de 24).

Est-ce que `-O3` change `sizeof(struct)` ?

- Non: l'optimisation ne change pas l'ABI/layout mémoire "observable" d'une struct standard.
- Ce qui change la taille:
  - l'ordre des champs
  - l'ABI/architecture (`long` peut varier)
  - les attributs de packing (`__attribute__((packed))`) **(à éviter** sauf besoin de format binaire imposé: performances/alignement)

Attention: portabilité d'un format binaire

Écrire "la mémoire brute" d'une struct:

- dépend de l'endianess
- dépend de la taille des types (`long`)
- dépend du padding

  => OK pour un TP local, mais pas un format d'échange stable sans normalisation.

---

8) Gestion d’erreur (pattern du TD)

Votre utilitaire `exit_if()` (dans `utils.c`) illustre un pattern simple:

- si `errno != 0` → `perror(prefix)`
- sinon → message simple sur `stderr`

Rappels:

- La plupart des syscalls renvoient `-1` en cas d’erreur et posent `errno`.
- Après succès, `errno` n’est pas garanti “remis à zéro”: on ne le lit qu’après un retour d’erreur.

---

9) Liens rapides (man)

- `read(2)`, `write(2)`
- `errno(3)`, `perror(3)`, `strerror(3)`
- `od(1)`, `objdump(1)`

---

10) Commandes utiles (TP)

Compiler rapidement un fichier (à adapter à votre Makefile):

```
gcc -Wall -Wextra -Werror -O2 06-read.c utils.c -o 06-read
```

Chaîner les programmes:

```
./04-write_number | ./07-read_number
./08-write_struct | ./08-read_struct
```