TD5 — Signaux & mémoire partagée (`mmap` / `shm`)

Objectif: (1) installer et manipuler des gestionnaires de signaux (`sigaction`, masques, `sigsuspend`), (2) communiquer entre processus via mémoire partagée (`mmap` fichier + `MAP_SHARED`), (3) comprendre protections mémoire (`PROT_*`) et erreurs (SIGSEGV/SIGBUS), (4) partager de la mémoire via SysV shm (`ftok/shmget/shmat`).

---

1) Signaux: bases utiles examen

Syscalls / libc

- `sigaction(2)` (recommandé) vs `signal(2/3)` (à éviter en code sérieux)
- `kill(2)` / `kill(1)` (shell), `raise(3)`
- Masques: `sigemptyset`, `sigaddset`, `sigprocmask(2)`, `sigpending(2)`, `sigismember(3)`
- Attente atomique: `sigsuspend(2)`

Signaux “incapturables”

- `SIGKILL` et `SIGSTOP` ne peuvent être ni ignorés ni interceptés.
- `SIGCONT` relance un processus stoppé.

Pattern correct: attendre SIGUSR1

Idée (comme `01-mysignal.c`):

1. afficher PID
2. installer handler via `sigaction`
3. attendre
4. restaurer handler par défaut

Bon pattern (anti-race):

- bloquer SIGUSR1 (`sigprocmask`) avant de se mettre en attente
- puis `sigsuspend` avec un masque où SIGUSR1 est débloqué

Pseudo-code:

```c
struct sigaction sa = {0};
sa.sa_handler = handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART; // optionnel
sigaction(SIGUSR1, &sa, NULL);

sigset_t block, old;
sigemptyset(&block);
sigaddset(&block, SIGUSR1);
sigprocmask(SIG_BLOCK, &block, &old);

for (;;) {
    sigset_t tmp = old;            // masque pendant l'attente
    sigdelset(&tmp, SIGUSR1);      // s'assurer que SIGUSR1 n'est pas bloqué
    sigsuspend(&tmp);              // revient après réception d'un signal
}
```

Très important: handler “async-signal-safe”

Dans vos handlers vous faites `printf()` + `strsignal()` : c’est pratique pour le TD, mais en vrai c’est **non sûr** dans un handler.

- À connaître: dans un handler, utiliser uniquement des fonctions **async-signal-safe** (ex: `write(2)`), ou bien juste poser un flag `volatile sig_atomic_t`.

---

2) `mmap` sur fichier: communication receveur/expéditeur

Ce que fait `mmap(2)`

- projette un fichier (ou mémoire anonyme) dans l’espace virtuel du processus.
- avec `MAP_SHARED`, les écritures dans la zone sont visibles par les autres processus mappant la même zone.

Syscalls / flags

- `open(2)`, `lseek(2)` (taille), `mmap(2)`, `munmap(2)`, `close(2)`
- `PROT_READ | PROT_WRITE` (lecture/écriture)
- `MAP_SHARED` (partagé)

TD: protocole (idée attendue)

- Receveur:
  - mappe le fichier
  - écrit son PID au début de la zone
  - attend `SIGUSR1`
  - à réception, affiche les données écrites par l'expéditeur
- Expéditeur:
  - mappe le fichier
  - lit le PID au début
  - copie le texte tapé sur stdin dans la zone
  - envoie `SIGUSR1` au PID

Points d'attention (pièges fréquents)

- **Ne pas écraser le PID**: écrire les données après `sizeof(pid_t)`.
  - ex: `read(0, (char*)data + sizeof(pid_t), size - sizeof(pid_t));`
- **Taille du segment**:
  - c'est la longueur passée à `mmap`.
  - si tu veux au moins 128 caractères, il faut un fichier de taille ≥ 128 (souvent via `ftruncate`).
- **Dépasser la taille**:
  - écrire au-delà d'une zone mappée peut provoquer `SIGSEGV`.
  - pour un mapping fichier, accéder au-delà de la taille du fichier peut provoquer `SIGBUS`.

Astuce debug: `hexdump -C data.txt` pour visualiser le PID écrit en binaire (endianness).

---

3) `mmap` anonyme + protections (`PROT_*`) => pourquoi ça segfault

Exercice `03-map_anonymous.c`:

```
p = mmap(NULL, sizeof(int), PROT, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
```

Permissions

- `PROT_NONE`: aucun accès → lire/écrire déclenche faute (SIGSEGV)
- `PROT_READ`: lecture ok, écriture → faute

- `PROT_WRITE` : écriture ok, lecture… **dépend de l'OS/arch** (sur Linux, `PROT_WRITE` peut permettre la lecture en pratique)
- `PROT_EXEC` : exécution, mais pas lecture/écriture

À retenir: en pratique on met souvent `PROT_READ | PROT_WRITE` pour des données.

---

4) Partage de code: exécuter du code mappé (démo type "lib partagée")

Sujet: copier les octets d'une fonction `add()` dans un fichier mappé `MAP_SHARED`, puis dans un autre process mapper ce fichier en `PROT_EXEC` et appeler via un pointeur de fonction.

APIs

- `mmap(2)` / `munmap(2)`
- `memcpy(3)`
- `ftruncate(2)` pour fixer la taille du fichier

Points importants (à connaître)

- Beaucoup de systèmes appliquent des politiques **W^X** (mémoire pas écrivable et exécutable en même temps). `PROT_WRITE|PROT_EXEC` peut échouer.
  - alternative plus réaliste: écrire en `PROT_WRITE`, puis `mprotect(2)` vers `PROT_EXEC|PROT_READ`.
- Copier "2048 octets" d'une fonction n'est pas portable (taille inconnue, relocations, dépendances, PIC…). C'est une démo pédagogique.
- Lancer plusieurs `use_add` simultanément: tous exécutent le même code depuis le même fichier partagé; si `load_add` réécrit le fichier pendant l'exécution, tu peux observer des comportements non déterministes (course).

---

5) Mémoire partagée SysV: `shmget/shmat` (persistante)

APIs

- `ftok(3)` → `key_t` (nécessite un **fichier existant**)
- `shmget(2)` → obtient/crée un segment

- `shmat(2)` → attache dans l'espace virtuel
- `shmdt(2)` → détache
- (important) `shmctl(2)` avec `IPC_RMID` pour supprimer le segment

Observation attendue

Si tu fais:

- `int *data = shmat(...); data[0]++; printf("%d", data[0]);`
  Alors en relançant plusieurs fois:
- la valeur continue d'augmenter, car le segment **survit** aux processus.

Pourquoi en ssh ça change ?

- Les segments SysV sont gérés **par noyau**: changer de machine (ssh sur un autre host) => autre noyau => autre espace shm.
- Même clé `ftok` sur 2 machines différentes ne référence pas le "même" segment.

Extension (sujet): liste des PIDs

Format simple dans le segment:

- `data[0] = nb_pids`
- puis tableau des PIDs
- attention à la concurrence: sans synchronisation (sémaphores, atomiques, spinlock), ça peut se corrompre.

---

6) Pages de man à connaître

- `sigaction(2)`, `sigsuspend(2)`, `sigprocmask(2)`, `sigpending(2)`, `kill(2)`
- `mmap(2)`, `munmap(2)`, `mprotect(2)`
- `ftok(3)`, `shmget(2)`, `shmat(2)`, `shmdt(2)`, `shmctl(2)`