TD6 — Threads & concurrence (pthreads)

Objectif: créer/attendre des threads (`pthread_create/join`), comprendre ce qui est **partagé** vs **privé** (pile vs data), identifier une **section critique** et corriger une condition de course avec **mutex** ou **sémaphore**.

---

1) Rappels: processus vs threads

- Un **processus** a (en gros) son espace d'adressage.
- Des **threads** d'un même processus partagent:
  - le code, les variables globales / heap, les fd…
- Chaque thread a sa **pile** (stack) propre.

Conséquence directe:

- variable **locale** (dans une fonction de thread) → adresse différente par thread
- variable **globale** → adresse identique (partagée) → risque de data race

Démo: 01-thread.c

- `local_counter` a une adresse différente pour chaque thread.
- `global_counter` a la même adresse pour tous.

---

2) Créer 8 threads + passer un "id"

APIs

- `pthread_create(3)`
- `pthread_join(3)`

Signature:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Passer un entier en argument (portable 64-bit):

```c
pthread_create(&t, NULL, start, (void*)(intptr_t)i);

void *start(void *arg) {
    int id = (int)(intptr_t)arg;
    ...
}
```

Attendre tous les threads

```c
for (int i=0; i<N; i++) pthread_join(threads[i], NULL);
```

Compilation

Toujours avec pthreads:

```
gcc ... -pthread
# ou Makefile: CFLAGS+=-pthread
```

---

3) Variables locales vs globales: ce qui se passe

Dans le sujet on te demande:

- 1 variable globale + 1 variable locale à `start`, toutes deux initialisées à 0
- les incrémenter et afficher adresse + valeur

Ce que tu observes:

- `local_counter`:
  - chaque thread a sa copie → valeurs indépendantes
  - adresses différentes (piles différentes)
- `global_counter`:
  - partagé → incréments concurrents → résultat non déterministe possible
  - adresse identique

Mot-clé examen: data race = 2 threads accèdent à la même donnée, au moins 1 écrit, sans synchronisation.

---

4) Ticket sellers: condition de course (data race)

Dans 02-tickets.c:

- variable globale `ticket_to_sell`
- chaque thread fait:
  - teste `ticket_to_sell > 0`
  - `sched_yield()`
  - décrémente `ticket_to_sell`

Où est la section critique ?

La séquence "tester puis décrémenter" doit être atomique:

- lecture de `ticket_to_sell`
- décision
- écriture `ticket_to_sell--`

Sinon:

- deux threads peuvent voir `ticket_to_sell == 1`, passer le test, puis décrémenter chacun → `-1` et total vendu trop grand.

Pourquoi `sched_yield()` aggrave le problème ?

`yield` force des changements de contexte → augmente la probabilité d'interleavings "mauvais".

---

5) Correction 1: mutex (exclusion mutuelle)

Implémentation: 02-mutex.c

APIs

- `pthread_mutex_t`
- `pthread_mutex_lock(3)` / `pthread_mutex_unlock(3)`

Pattern:

```c
pthread_mutex_lock(&m);
if (ticket_to_sell > 0) {
    ticket_to_sell--;
    sold++;
}
pthread_mutex_unlock(&m);
```

Propriété:

- exactement 1 thread à la fois dans la section critique.

---

6) Correction 2: sémaphore (binaire)

Implémentation: 02-semaphore.c

APIs

- `sem_init(3)`, `sem_wait(3)`, `sem_post(3)`, `sem_destroy(3)`

Pour un sémaphore **binaire** (équivalent mutex simple):

- `sem_init(&sem, 0, 1)`
- `sem_wait` = prendre le “token”
- `sem_post` = relâcher

---

7) Retour de valeur d’un thread

Deux options:

- `pthread_exit(void*)`
- ou `return (void*)...` depuis la routine

Récupération via `pthread_join(thread, &ret)`.

Dans vos exos, vous passez un entier via `(void*)(intptr_t)ticket_sold`.

---

8) Check-list examen (threads)

- Toujours compiler avec `-pthread`.
- Toujours `pthread_join` (sinon fin prématurée / threads orphelins côté logique).
- Tout accès écriture à une donnée partagée → penser “mutex/sem/atomics”.
- Une section critique = test+modif d’un état partagé (ex: compteur de tickets).
- `sched_yield()` n’est pas une synchro, c’est juste un amplificateur de race.

---

9) Pages de man à connaître

- `pthreads(7)` / `pthread_create(3)` / `pthread_join(3)`
- `pthread_mutex_lock(3)` / `pthread_mutex_unlock(3)`
- `sem_init(3)` / `sem_wait(3)` / `sem_post(3)`
- `sched_yield(2)`