IUT de Saint-Dié-des-Vosges — Département Informatique

ASR4 — Systèmes d’Exploitation

TP — Synchronisation de Threads avec les Sémaphores POSIX

Auteur : Timothé Belcour

Date : 12/10/2025

Système : macOS (Terminal, clang/gcc + pthreads)

1. Objectifs

• Mettre en évidence une condition de course entre deux threads incrémentant une variable globale.
• Corriger le problème à l'aide d'un sémaphore POSIX pour rendre l'accès exclusif à la section critique.

2. Pré-requis & Organisation

• Créer le dossier : ~/ASR4/semposix/
• Placer les fichiers sources compt.c (non protégé) et compt_mutex.c (version protégée, sémaphore POSIX nommé) dans ce dossier.
• Compiler avec -lpthread (sur macOS, -lrt n'est pas nécessaire).
Arborescence conseillée :
~/ASR4/semposix/
■■■ compt.c
■■■ compt_mutex.c
■■■ TPsemaphore.pdf (énoncé)

3. Exercice 1 — Mise en évidence de la condition de course

Compilation & exécution :
cd ~/ASR4/semposix
gcc compt.c -o compt -lpthread
./compt
./compt
./compt
Résultat attendu :
• Sortie non déterministe : parfois OK! compteur = [2000000], parfois BOOM! compteur < 2000000.
• Explication : les deux threads exécutent un schéma read–modify–write non atomique sur la variable globale compteur, ce qui provoque des interférences et des pertes d'incréments (condition de course).
Code source (compt.c) :

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
int compteur = 0;
void * incr(void * a)
{
   int i, tmp;
   for(i = 0; i < N; i++)
   {
   tmp = compteur; tmp = tmp+1; compteur = tmp; /* copie la variable globale compteur localement */
   /* incrémente la copie locale */
   /* stocke la valeur locale dans la variable globale compteur */
   }
}
int main(int argc, char * argv[]){

   pthread_t tid1, tid2;
   if(pthread_create(&tid1, NULL, incr, NULL))
   {
      printf("\n ERREUR création thread 1");
      exit(1);
   }
   if(pthread_create(&tid2, NULL, incr, NULL))
   {
      printf("\n ERROR création thread 2");
      exit(1);
```

```
    }
    if(pthread_join(tid1, NULL)) {
        printf("\n ERREUR thread 1 ");
        exit(1);
        /* Attente de fin de thread 1 */
    }
    if(pthread_join(tid2, NULL)) {
        printf("\n ERREUR thread 2");
        exit(1);
        /* Attente de fin de thread 2 */
    }
    if ( compteur < 2 * N)
        printf("\n BOOM! compteur = [%d], devrait être %d\n", compteur, 2*N);
    else
        printf("\n OK! compteur = [%d]\n", compteur);
        pthread_exit(NULL);
    return 0 ;
}
```

4. Exercice 2 — Correction via un sémaphore POSIX

Sur macOS, l'API des sémaphores POSIX non nommés (sem_init) est dépréciée/indisponible. On utilise donc un sémaphore POSIX nommé via sem_open/sem_wait/sem_post/sem_close/sem_unlink. On protège exactement la section critique (lecture/modification/écriture de compteur).

Compilation & exécution :

```
gcc compt_mutex.c -o compt_mutex -lpthread
./compt_mutex
./compt_mutex; ./compt_mutex; ./compt_mutex
```

Résultat attendu :

• Toujours OK! compteur = [2000000] (soit 2×N).

• La section critique est sérialisée : un seul thread à la fois peut faire read–modify–write.

Code source (compt_mutex.c) :

```
// compt_mutex_named.c  (macOS-friendly)
// Remplace sem_init/sem_destroy par sem_open/sem_close/sem_unlink (sémaphore POSIX nommé)
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>    // O_CREAT, O_EXCL
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   // getpid

#define N 1000000

int compteur = 0;
sem_t *mutex = NULL;

void* incr(void* a) {
    for (int i = 0; i < N; i++) {
        sem_wait(mutex);
        int tmp = compteur;   // read
        tmp = tmp + 1;        // modify
        compteur = tmp;       // write
        sem_post(mutex);
    }
    return NULL;
}

int main(void) {
    pthread_t tid1, tid2;
    // Nom unique pour éviter les collisions si tu relances souvent
```

```
    char name[64];
    snprintf(name, sizeof(name), "/compteur_mutex_%d", getpid());

    // Par précaution : supprimer un éventuel ancien sémaphore portant le même nom
    sem_unlink(name);

    // Création (valeur initiale = 1, comme un mutex binaire)
    mutex = sem_open(name, O_CREAT | O_EXCL, 0644, 1);
    if (mutex == SEM_FAILED) {
        perror("sem_open");
        return 1;
    }

    if (pthread_create(&tid1, NULL, incr, NULL) != 0) { perror("pthread_create tid1"); return 1; }
    if (pthread_create(&tid2, NULL, incr, NULL) != 0) { perror("pthread_create tid2"); return 1; }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (compteur < 2 * N)
        printf("\nBOOM! compteur = [%d], devrait être %d\n", compteur, 2 * N);
    else
        printf("\nOK! compteur = [%d]\n", compteur);

    // Fermeture + suppression du sémaphore nommé
    sem_close(mutex);
    sem_unlink(name);
    return 0;
}
```

5. Tests & Validation

• Comparer plusieurs exécutions de ./compt (résultat variable) et ./compt_mutex (résultat stable).
• Optionnel : mesurer le temps avec time ./compt et time ./compt_mutex ; la sérialisation peut introduire une petite pénalité mais garantit la correction.

6. Points d’attention spécifiques à macOS

• Les fonctions sem_init/sem_destroy sont dépréciées. Utiliser sem_open (nommé) et penser à sem_unlink pour éviter les collisions de noms.
• Pas de -lrt à l’édition de liens : sur macOS, -lpthread suffit.

7. Conclusion

Le premier programme illustre la condition de course : des opérations read–modify–write concurrentes sur une variable globale aboutissent à une perte d’incréments. En encadrant la section critique par un sémaphore POSIX (attente sem_wait, signal sem_post), on garantit l’exclusion mutuelle et un résultat déterministe égal à 2×N. La variante POSIX nommée (sem_open) est privilégiée sur macOS pour des raisons de compatibilité.