

# Interprocess Synchronization

## Concurrent Access To Shared Memory: Race Problems

1.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define KEY 4500
#define PERMS 8600

void *inc(void* i){
    printf("INCREMENTATION DEBUT | i: %d\n", *(int*)i);
    i++;
    printf("INCREMENTATION FIN | i: %d\n", *(int*)i);
    return NULL;
}

void *dec(void* i){
    printf("DECREMENTATION DEBUT | i: %d\n", *(int*)i);
    i--;
    printf("DECREMENTATION FIN | i: %d\n", *(int*)i);
    return NULL;
}

int main(){
    pthread_t thread1, thread2;
    int i = 10;

    pthread_create(&thread1, NULL, inc, &i);
    pthread_create(&thread2, NULL, dec, &i);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

```
adrito@ubuntu:~/Documents/ECE/Lab5$ ./exec.exe
DECREMENTATION DEBUT | i: 10
DECREMENTATION FIN | i: 2716
INCREMENTATION DEBUT | i: 10
INCREMENTATION FIN | i: 0
adrito@ubuntu:~/Documents/ECE/Lab5$
```

2.

Au début, i a la même valeur. Dans les deux cas, i = 65.

Le problème vient du fait que les deux processus sont exécutés en parallèle et c'est donc le dernier qui va terminer qui va allouer sa valeur à i.

Dans un cas i = 65

L'autre i = 66.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define KEY 4500
#define PERMS 8600

void *inc(void* i){

    printf("INCREMENTATION DEBUT | i: %d\n", *(int*)i);
    int reg;
    reg = *(int*)i;
    sleep(1);
    reg++;
    *(int*)i= reg;
    printf("INCREMENTATION FIN | i: %d\n", *(int*)i);
    return NULL;
}

void *dec(void* i){

    printf("DECREMENTATION DEBUT | i: %d\n", *(int*)i);
    int reg;
    reg = *(int*)i;
    sleep(1);
    reg--;
    *(int*)i= reg;
    printf("DECREMENTATION FIN | i: %d\n", *(int*)i);
    return NULL;
}

int main(){

    pthread_t thread1, thread2;
    int i = 10;

    pthread_create(&thread1, NULL, inc, &i);
    pthread_create(&thread2, NULL, dec, &i);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

Avec la méthode reg, on obtient un résultat cohérent :

```
adrito@ubuntu:~/Documents/ECE/Lab5$ ./exec.exe
DECREMENTATION DEBUT | i: 10
INCREMENTATION DEBUT | i: 10
DECREMENTATION FIN | i: 9
INCREMENTATION FIN | i: 11
adrito@ubuntu:~/Documents/ECE/Lab5$
```

# Solving the Problem : Synchronizing access using semaphores

1.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define KEY 4500
#define PERMS 8600
sem_t mutex;

void *inc(void* i){

    sem_wait(&mutex); // Prologue
    printf("INCREMENTATION DEBUT | i: %d\n", *(int*)i);
    int reg;
    reg = *(int*)i;
    sleep(1);
    reg++;
    *(int*)i= reg;
    printf("INCREMENTATION FIN | i: %d\n", *(int*)i);
    sem_post(&mutex); // Epilogue
    return NULL;
}

void *dec(void* i){

    sem_wait(&mutex);
    printf("DECREMENTATION DEBUT | i: %d\n", *(int*)i);
    int reg;
    reg = *(int*)i;
    sleep(1);
    reg--;
    *(int*)i= reg;
    printf("DECREMENTATION FIN | i: %d\n", *(int*)i);
    sem_post(&mutex);
    return NULL;
}

int main(){

    sem_init(&mutex, 0,1);
    pthread_t thread1, thread2;
    int i = 10;

    pthread_create(&thread2, NULL, dec, &i);
    pthread_create(&thread1, NULL, inc, &i);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

Si on a plus de deux processus, il faut créer un nouveau sémaphore pour renforcer l'exclusion mutuelle.

Si dessous, on essaie avec trois processus :

Ici on a un thread qui additionne, un qui soustrait et un qui multiplie.

```
void *inc(void* i){
    sem_wait(&mutex); // Prologue
    printf("INCREMENTATION DEBUT | i: %d\n", *(int*)i);
    int reg;
    reg = *(int*)i;
    sleep(1);
    reg++;
    *(int*)i= reg;
    printf("INCREMENTATION FIN | i: %d\n", *(int*)i);
    sem_post(&mutex); // Epilogue
    return NULL;
}

void *dec(void* i){
    sem_wait(&mutex);
    printf("DECREMENTATION DEBUT | i: %d\n", *(int*)i);
    int reg;
    reg = *(int*)i;
    sleep(1);
    reg--;
    *(int*)i= reg;
    printf("DECREMENTATION FIN | i: %d\n", *(int*)i);
    sem_post(&mutex);
    return NULL;
}

void *mult(void* i){
    sem_wait(&mutex);
    printf("MULTIPLICATION DEBUT | i: %d\n", *(int*)i);
    int reg;
    reg = *(int*)i;
    sleep(1);
    reg*=2;
    *(int*)i= reg;
    printf("MULTIPLICATION FIN | i: %d\n", *(int*)i);
    sem_post(&mutex);
    return NULL;
}

int main(){
    sem_init(&mutex, 0,1);
    pthread_t thread1, thread2, thread3;
    int i = 10;

    pthread_create(&thread2, NULL, dec, &i);
    pthread_create(&thread1, NULL, inc, &i);
    pthread_create(&thread3, NULL, mult, &i);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);

    sem_destroy(&mutex);
}
```

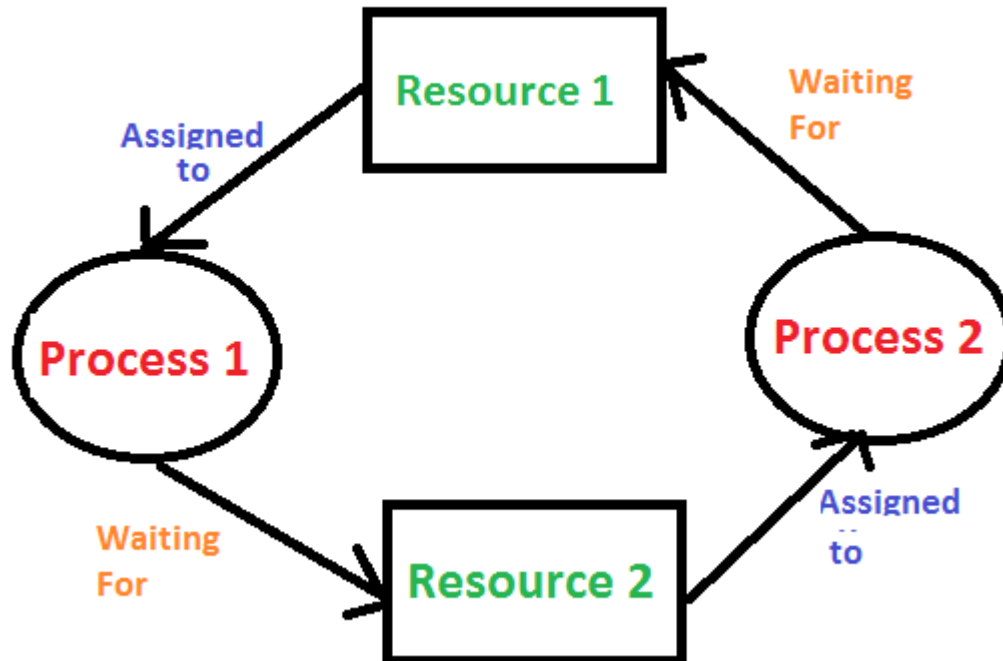
On obtient la sortie suivante :

```
adrito@ubuntu:~/Documents/ECE/OS/Lab5$ ./exec.exe
MULTIPLICATION DEBUT | i: 10
MULTIPLICATION FIN | i: 20
INCREMENTATION DEBUT | i: 20
INCREMENTATION FIN | i: 21
DECREMENTATION DEBUT | i: 21
DECREMENTATION FIN | i: 20
adrito@ubuntu:~/Documents/ECE/OS/Lab5$
```

(sem\_init, sem\_wait, sem\_post)

## 2. Deadlock :

Ici le but est de créer un deadlock :



On crée trois thread de la sorte dans lesquels on interroge de la manière présentée si dessus.

```
void * proc2(){
    printf("Process 2 started\n");
    pthread_mutex_lock(&lock2);    //On verrouille le lock2 (pour le thread2)
    sleep(1);
    printf("Trying to get ressource 3 \n"); // On essaie de collecter une ressource du thread3.
    pthread_mutex_lock(&lock3); //On verrouille le lock3 mais already locked par thread3.
    printf("Aquired ressource 3 \n");
    pthread_mutex_unlock(&lock3);    //On déverrouille le thread3.
    printf("Trying to get ressource 1 \n"); // On essaie de collecter une ressource du thread1.
    pthread_mutex_lock(&lock1); //On verrouille le lock1 mais already locked par thread1.
    printf("Aquired ressource 1 \n");
    pthread_mutex_unlock(&lock1); //On déverrouille le thread1.
    printf("Job done in process 2 \n"); //Si le programme est ok, on ne voit jamais cette ligne
    pthread_mutex_unlock(&lock2); //On déverrouille le thread2.
    pthread_exit(NULL);
}
```

```
adrito@ubuntu:~/Documents/ECE/OS/Lab5$ ./exec.exe
Process 3 started
Process 2 started
Process 1 started
Trying to get ressource 1
Trying to get ressource 3
Trying to get ressource 2
```

3.

Le code suivant nous permet d'ouvrir 3 applications dans un ordre précis, à savoir Firefox, un terminal puis Thunderbird.

Grâce aux sémaphores, les trois applications s'ouvrent correctement dans le bon ordre.

```
void *prg1(void* i){

    sem_wait(&mutex1); // Prologue
    printf("%d\n", *(int*)i);
    system("firefox");
    sem_post(&mutex2); // Epilogue
}

void *prg2(void* j){

    sem_wait(&mutex2);
    printf("%d\n", *(int*)j);
    system("gnome-terminal");
    sem_post(&mutex3);
}

void *prg3(void* k){

    sem_wait(&mutex3);
    printf("%d\n", *(int*)k);
    system("thunderbird");
}

int main(){

    sem_init(&mutex1, 0,1);
    sem_init(&mutex2, 0,0);
    sem_init(&mutex3, 0,0);

    pthread_t thread1, thread2, thread3;
    int i = 1;
    int j = 2;
    int k = 3;

    pthread_create(&thread1, NULL, prg1, &i);
    pthread_create(&thread2, NULL, prg2, &j);
    pthread_create(&thread3, NULL, prg3, &k);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);

    sem_destroy(&mutex1);
    sem_destroy(&mutex2);
    sem_destroy(&mutex3);

    return 0;
}
```

---

4.

La fonction `pthread_create` ne pouvant recevoir que 4 arguments au maximum, nous sommes dans l'obligation de créer un groupement de variable. Pour ça, deux solutions, à savoir l'utilisation de tableaux ou bien de structure. Ici nous allons utiliser un tableau.

```
int* table[3] = {0,0,0};
```

On crée ensuite un sous programme qui va nous permettre de gérer l'avancée des calculs effectués dans les différents threads. En effet, dès qu'un résultat arrive, la valeur du compteur (count) est incrémentée. Une fois arrivée à trois, le tableau cesse de se remplir.

Le sous-programme reçoit en paramètre un pointeur sur la table partagée (int) ainsi qu'un (int) représentant le résultat de l'opération effectuée par le thread.

```
16 int storeInTable(int* sharedTable, int x){
17     int cpt=0;
18     for(int i=0; i<3; i++){
19         if((sharedTable[i]==0)&&(cpt==0)){
20             sharedTable[i] = x;
21             cpt=cpt+1;
22             return i;
23         }
24     }
25     return 0;
26 }
```

On crée ensuite 3 threads dédiés aux calculs suivant : (a+b), (c-d) et (e+f). Le quatrième thread servira pour afficher le résultat final : (a+b) \* (c-d) \* (e+f).

```
28 void *calc1(void *table){
29     int* sharedTable = (int*) table;
30     int a=1;
31     int b=2;
32     storeInTable(sharedTable, a + b);
33     printf("sous-élément 1 calculé\n");
34     sem_post(&pmutex);
35     pthread_exit(NULL);
36 }
--
```

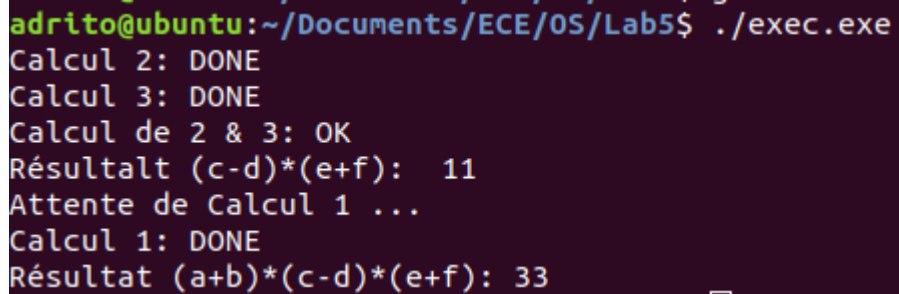
Ici, on n'utilise que `sem_post()` qui sert à déverrouiller le sémaphore pointé par `sem`. Le but est de signaler (en incrémentant la valeur du sémaphore) que le processus est terminé.

On réitère l'opération trois fois puis on passe au calcul final.

Calcul final :

```
59 void *res4(void *table){
60     int* sharedTable = (int*) table;
61
62     while(sharedTable[1]==0){
63         sem_wait(&pmutex);
64     }
65     printf("Calcul de 2 & 3: OK\n");
66
67
68     int result;
69     result = sharedTable[0] * sharedTable[1];
70     printf("Résultat (c-d)*(e+f): %d\n", result);
71
72
73     printf("Attente de Calcul 1 ...\n");
74     sem_wait(&pmutex);
75
76
77     result = result * sharedTable[2]; // Résultat de la partie 2 & 3 * P1
78     printf("Résultat (a+b)*(c-d)*(e+f): %d\n", result);
79     pthread_exit(NULL);
80 }
```

Le résultat dans le terminal :



```
adrito@ubuntu:~/Documents/ECE/OS/Lab5$ ./exec.exe
Calcul 2: DONE
Calcul 3: DONE
Calcul de 2 & 3: OK
Résultat (c-d)*(e+f): 11
Attente de Calcul 1 ...
Calcul 1: DONE
Résultat (a+b)*(c-d)*(e+f): 33
```