

LAB 5 :

Interprocess Synchronization



I - Concurrent Access To Shared Memory : Race Problems

1. Code with race problems

```
Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part1.c > ...
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8  #include <semaphore.h>
9  #include <pthread.h>
10
11  int i = 65;
12
13  void *child(void *arg){
14      printf("Child before : %d\n", i);
15      int reg = i;
16      sleep(2);
17      ++reg;
18      i = reg;
19      printf("Child after (++i) : %d\n", i);
20      return NULL;
21  }
22
23  void *parent(void *arg){
24      printf("Parent before : %d\n", i);
25      int reg = i;
26      sleep(2);
27      --reg;
28      i = reg;
29      printf("Parent after (--i) : %d\n", i);
30      return NULL;
31  }
32
33  int main (int argc, char *argv[]) {
34
35      int iret1, iret2;
36      pthread_t thread1, thread2;
37
38      printf("initial value i = %d\n", i);
39
40      //printf("child: begin\n");
41      iret1 = pthread_create(&thread1, NULL, child, NULL);
42      if(iret1) {
43          fprintf(stderr, "Error - pthread_create() return code: %d\n", iret1);
44          exit(EXIT_FAILURE);
45      }
46
47      //printf("parent: begin\n");
48      iret2 = pthread_create(&thread2, NULL, parent, NULL);
49      if(iret2) {
50          fprintf(stderr, "Error - pthread_create() return code: %d\n", iret2);
51          exit(EXIT_FAILURE);
52      }
53
54      pthread_join(thread1, NULL);
55      pthread_join(thread2, NULL);
56
57      printf("final value : i = %d\n", i);
58
59      return 0;
60  }
```

2. Results

```
[→ lab5 gcc -g -o ex1bis part1bis.c
[→ lab5 ./ex1bis
initial value i = 65
Child before : 65
Parent before : 65
Child after (++i) : 66
Parent after (--i) : 64
final value : i = 64
```

In this exemple, we can observe **critical race conditions** because the threads depend on the same shared value of "i". The two operations we carry out on variable "i" are **not mutually exclusive**. It means that there is **no restriction to access to the shared data** and **operations can be interrupted** while accessing the segment of shared memory. As a result, the **execution is invalid** and we get a wrong final result.

Here is what happens in the code with the shared data (inspired by Wikipedia, see III - Sources) :

Thread 1	Thread 2		Integer value
			65
read value		←	65
	read value	←	65
increase value			66
	decrease value		64
write back		→	66
	write back	→	64
final value		→	64

II - Solving the Problem : Synchronizing access using semaphores

1. Solving race problem

```
Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part2-1.c > ...
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8  #include <semaphore.h>
9  #include <pthread.h>
10
11  sem_t * s;
12  int i = 65;
13
14  void *mythread1(void *arg){
15      printf("thread1 before : %d\n", i);
16      ++i;
17      printf("thread1 after (++i) : %d\n", i);
18      sem_post(s);
19      return NULL;
20  }
21
22  void *mythread2(void *arg){
23      sem_wait(s);
24      printf("thread2 before : %d\n", i);
25      --i;
26      printf("thread2 after (--i) : %d\n", i);
27      return NULL;
28  }
29
30  int main (int argc, char *argv[]) {
31
32      int iret1, iret2;
33      pthread_t thread1, thread2;
34
35      s = sem_open("sem", 0_CREAT, S_IRUSR | S_IWUSR, 1);
36
37      printf("initial value i = %d\n", i);
38
39      iret1 = pthread_create(&thread1, NULL, mythread1, NULL);
40      if(iret1) {
41          fprintf(stderr, "Error - pthread_create() return code: %d\n", iret1);
42          exit(EXIT_FAILURE);
43      }
44
45      iret2 = pthread_create(&thread2, NULL, mythread2, NULL);
46      if(iret2) {
47          fprintf(stderr, "Error - pthread_create() return code: %d\n", iret2);
48          exit(EXIT_FAILURE);
49      }
50
51      pthread_join(thread1, NULL);
52      pthread_join(thread2, NULL);
53
54      printf("final value : i = %d\n", i);
55
56      return 0;
57  }
```

Results

```
[→ lab5 gcc -g -o ex21 part2-1.c
[→ lab5 ./ex21
initial value i = 65
thread1 before : 65
thread1 after (++i) : 66
thread2 before : 66
thread2 after (--i) : 65
final value : i = 65
```

The race condition problem has been solved using semaphores which allows the first thread to be fully executed before the second one is launched using `sem_wait()` and `sem_post()` :

- 1st : `sem_post()` unlocks the semaphore pointed by `sem`. The value of the semaphore becomes greater than zero (process1)
- 2nd : `sem_wait()` will be woken up and proceed to lock the semaphore (process2)

If we had more than 2 processes, we should determine an order of execution for each of them in order to be executed one after another. If not, we **could have a deadlock situation** in which a process is waiting for some resource held by another process waiting for it to release another resource.

That is why we can **use lock to enforce a mutual exclusion concurrency control policy**.

2. Force a deadlock situation using semaphores (and 3 threads)

Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part2-2.c > ...

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8  #include <semaphore.h>
9  #include <pthread.h>
10
11  sem_t * mutex1;
12  sem_t * mutex2;
13  sem_t * mutex3;
14  int i = 65;
15
16  void *mythread1(void *arg){
17      sem_wait(mutex1);
18      sleep(2);
19      sem_wait(mutex2);
20      printf("T1 before : %d\n", i);
21      ++i;
22      printf("T1 after (++i) : %d\n", i);
23      sem_post(mutex2);
24      sem_post(mutex1);
25      return NULL;
26  }
27
28  void *mythread2(void *arg){
29      sem_wait(mutex2);
30      sleep(2);
31      sem_wait(mutex3);
32      printf("T2 before : %d\n", i);
33      --i;
34      printf("T2 after (--i) : %d\n", i);
35      sem_post(mutex3);
36      sem_post(mutex2);
37      return NULL;
38  }
39
40  void *mythread3(void *arg){
41      sem_wait(mutex3);
42      sleep(2);
43      sem_wait(mutex1);
44      printf("T3 before : %d\n", i);
45      i+=10;
46      printf("T3 after (i+=10) : %d\n", i);
47      sem_post(mutex1);
48      sem_post(mutex3);
49      return NULL;
50  }
```

```

54 int main (int argc, char *argv[]) {
55
56     int iret1, iret2, iret3;
57     pthread_t thread1, thread2, thread3;
58
59     mutex1 = sem_open("mutex1", 0_CREAT, S_IRUSR | S_IWUSR, 1);
60     mutex2 = sem_open("mutex2", 0_CREAT, S_IRUSR | S_IWUSR, 1);
61     mutex3 = sem_open("mutex3", 0_CREAT, S_IRUSR | S_IWUSR, 1);
62
63     printf("initial value i = %d\n", i);
64
65     iret1 = pthread_create(&thread1, NULL, mythread1, NULL);
66     if(iret1) {
67         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret1);
68         exit(EXIT_FAILURE);
69     }
70
71     iret2 = pthread_create(&thread2, NULL, mythread2, NULL);
72     if(iret2) {
73         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret2);
74         exit(EXIT_FAILURE);
75     }
76
77     iret3 = pthread_create(&thread3, NULL, mythread3, NULL);
78     if(iret3) {
79         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret3);
80         exit(EXIT_FAILURE);
81     }
82
83     pthread_join(thread1, NULL);
84     pthread_join(thread2, NULL);
85     pthread_join(thread3, NULL);
86
87     printf("final value : i = %d\n", i);
88
89     return 0;
90 }

```

Results

```

[→ lab5 gcc -g -o ex22 part2-2.c
[→ lab5 ./ex22
initial value i = 65

```

In this case **we created a deadlock situation using semaphores**, because each thread is waiting for another to be unlocked (thread 2 for thread 1 / thread 3 for thread 2 / thread 1 for thread 3).

It is forming a **loop of blocked processes**. Therefore the compilation is endless and won't stop until we manually do it:

```

[→ lab5 gcc -g -o ex22 part2-2.c
[→ lab5 ./ex22
initial value i = 65
^C
→ lab5

```


3. Run 3 different applications

```
sem_t * s2, *s3;

void *firefox(){
    if (fork() == 0){
        execlp("firefox", "firefox", NULL);
    }
    sem_post(s2);
}

void *vi(){
    sem_wait(s2);
    if (fork() == 0){
        execlp("vi", "vi", NULL);
    }
    sem_post(s3);
}

void *emacs(){
    sem_wait(s3);
    if (fork() == 0){
        execlp("emacs", "emacs", NULL);
    }
}

int main (int argc, char *argv[]) {

    pthread_t p1, p2, p3;

    s2 = sem_open("s2", O_CREAT, S_IRUSR | S_IWUSR, 0);
    s3 = sem_open("s3", O_CREAT, S_IRUSR | S_IWUSR, 0);

    pthread_create(&p1, NULL, firefox, NULL);
    pthread_create(&p2, NULL, vi, NULL);
    pthread_create(&p3, NULL, emacs, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);
}
```


4. Implement a parallelized calculation with semaphores

```
Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part2-4.c > main(int, char * [])
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8  #include <semaphore.h>
9  #include <pthread.h>
10
11  //(a+b)*(c-d)*(e+f)
12
13  sem_t * mutex1;
14  sem_t * mutex2;
15  sem_t * mutex3;
16  int a=10, b=5, c=7, d=2, e=1, f=1;
17  int count = -1; // counter for the number of results in : results[]
18  int results[3]; // results table
19
20  void *mythread1(void *arg){
21
22      sem_wait(mutex3);
23      count++;
24      results[count] = (a+b);
25      printf("T1: (a+b) = %d\n", results[count]);
26
27      if (count == 1) // get 2 results in the table
28      {
29          sem_post(mutex1); // unlock mutex1 in mythreads4
30      }
31      else if (count == 2) // get 3 results in the table
32      {
33          sem_post(mutex2); // unlock mutex2 in mythreads4
34      }
35      else // get 0 or 1 result : call 1 of the 2 remaining threads
36      {
37          sem_post(mutex3);
38      }
39      return NULL;
40  }
```

```
42  void *mythread2(void *arg){
43
44      sem_wait(mutex3);
45      count++;
46      results[count] = (c-d);
47      printf("T2: (c-d) = %d\n", results[count]);
48
49      if (count == 1)
50      {
51          sem_post(mutex1);
52      }
53      else if (count == 2)
54      {
55          sem_post(mutex2);
56      }
57      else
58      {
59          sem_post(mutex3);
60      }
61      return NULL;
62  }
63
64  void *mythread3(void *arg){
65
66      sem_wait(mutex3);
67      count++;
68      results[count] = (e+f);
69      printf("T3: (e+f) = %d\n", results[count]);
70
71      if (count == 1)
72      {
73          sem_post(mutex1);
74      }
75      else if (count == 2)
76      {
77          sem_post(mutex2);
78      }
79      else
80      {
81          sem_post(mutex3);
82      }
83      return NULL;
84  }
85
86  void *mythread4(void *arg){
87
88      sem_wait(mutex1);
89      results[0] = results[0]*results[1]; // stock the 1st result (btw the 2 first threads) in the table
90      printf("T4: First Result (T1*T2) = %d\n", results[0]);
91
92      sem_post(mutex3);
93
94      sem_wait(mutex2);
95      results[0] = results[0]*results[2]; // stock the final result
96      printf("T4: Final Result (T1*T2*T3) = %d\n", results[0]);
97
98      return NULL;
99  }
```

```

101 int main (int argc, char *argv[]) {
102
103     int iret1, iret2, iret3, iret4;
104     pthread_t thread1, thread2, thread3, thread4;
105
106     // Initialization to 0 : waiting for the 3 other threads to call sem_post()
107     // and increment the value of semaphores from 0 to 1
108     mutex1 = sem_open("mutexx1", 0_CREAT, S_IRUSR | S_IWUSR, 0);
109     mutex2 = sem_open("mutexx2", 0_CREAT, S_IRUSR | S_IWUSR, 0);
110     // Initialization to 1 : waiting for 1 of the 3 threads to call sem_wait()
111     // and decrement the value of semaphores from 1 to 0
112     mutex3 = sem_open("mutexx3", 0_CREAT, S_IRUSR | S_IWUSR, 1);
113
114     iret1 = pthread_create(&thread1, NULL, mythread1, NULL);
115     if(iret1) {
116         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret1);
117         exit(EXIT_FAILURE);
118     }
119
120     iret2 = pthread_create(&thread2, NULL, mythread2, NULL);
121     if(iret2) {
122         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret2);
123         exit(EXIT_FAILURE);
124     }
125
126     iret3 = pthread_create(&thread3, NULL, mythread3, NULL);
127     if(iret3) {
128         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret3);
129         exit(EXIT_FAILURE);
130     }
131
132     iret4 = pthread_create(&thread4, NULL, mythread4, NULL);
133     if(iret4) {
134         fprintf(stderr, "Error - pthread_create() return code: %d\n", iret4);
135         exit(EXIT_FAILURE);
136     }
137
138     pthread_join(thread1, NULL);
139     pthread_join(thread2, NULL);
140     pthread_join(thread3, NULL);
141     pthread_join(thread4, NULL);
142
143     return 0;
144 }

```

Results

```

[→ lab5 gcc -g -o ex24 part2-4.c -lpthread
[→ lab5 ./ex24
T1: (a+b) = 15
T2: (c-d) = 5
T4: First Result (T1*T2) = 75
T3: (e+f) = 2
T4: Final Result (T1*T2*T3) = 150
→ lab5 █

```

In this example, we used **4 different threads with 3 semaphores** in order to synchronize them as we want and compute the operation:

- The first 3 threads separately compute a part of the operation and use 1 semaphore (mutex3) in order to prevent a race problem. Moreover, the conditions on the counter determine how many operations the threads have already done and unlock mutex1 or mutex2 in consequence.
- The last thread realises the 2 main operations from the results of T1, T2 and T3.

The main() initialize mutex1 & mutex2 at 0, and mutex3 at 1 so that we have **the desired order of execution** of the threads (see “31.3 Semaphores For Ordering” in the course).

III - Sources

https://en.wikipedia.org/wiki/Race_condition

<https://medium.com/swlh/race-conditions-locks-semaphores-and-deadlocks-a4f783876529>

Course : Semaphores (on Campus)