



# Interprocess Synchronization

*Operating Systems*

*Submitted To:*

Mr. Khoury

*Submitted By:*

Romain Brisse

Hyunjae Lee

11/11/2018

L<sup>A</sup>T<sub>E</sub>X

# Contents

0.1	Remarks . . . . .	2
0.2	Concurrent access to shared memory: Race problems . . . . .	3
0.2.1	Q1: A Race condition . . . . .	3
0.2.2	Q2: . . . . .	4
0.3	Solving the Problem : Synchronizing access using semaphores . . . . .	5
0.3.1	Q1: . . . . .	5
0.3.2	Q2: . . . . .	7
0.3.3	Q3: . . . . .	9
0.3.4	Q4: . . . . .	11

## 0.1 Remarks

We submit to reports regarding this lab because we did most of it separately and only finished it together. We both wanted to show our work, and it gives you a c version and a java version! Thank you.

## 0.2 Concurrent access to shared memory: Race problems

### 0.2.1 Q1: A Race condition

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/wait.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <unistd.h>
7
8  #define KEY 1234
9  #define PERMS 0660
10
11 int main(int argc, char **argv)
12 {
13     int id, a;
14     id = shmget(KEY, sizeof(int), IPC_CREAT | PERMS);
15
16     a = (int) shmat(id, NULL, 0);
17     a = 65;
18
19     if(fork()==0)
20     {
21         sleep(2);
22         a--;
23         printf("-: %d\n", a);
24     }
25     else
26     {
27         sleep(2);
28         a++;
29         printf("+: %d\n", a);
30         sleep(2);
31         printf("=: %d\n", a);
32     }
33
34
35
36     return 0;
37 }
```

### 0.2.2 Q2:

```
romain@tuxtop:~/Bureau/s7/05/labs/lab5$ gcc -w -o conc conc.c
romain@tuxtop:~/Bureau/s7/05/labs/lab5$ ./conc
+: 66
-: 64
=: 66
```

Figure 1: the result of a race condition

The code presented above has a major flaw. Indeed, the variable "i" is shared and there is no restriction to access it. meaning that one of the two threads operating on it could interrupt the other one while trying to operate on i.

This means you could get a result like this when executing the program:

## 0.3 Solving the Problem : Synchronizing access using semaphores

### 0.3.1 Q1:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/wait.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <sys/mman.h>
7  #include <unistd.h>
8  #include <semaphore.h>
9  #include <pthread.h>
10
11 sem_t mutex;
12 int a;
13
14 void *fThread1(void *arg)
15 {
16     a--;
17     printf("-: %d\n", a);
18     sem_post(&mutex);
19 }
20
21 void *fThread2(void *arg)
22 {
23     sem_wait(&mutex);
24     a++;
25     printf("+: %d\n", a);
26 }
27
28 int main(int argc, char **argv)
29 {
30     sem_init(&mutex,0,0);
31     a=65;
32
33     pthread_t t1, t2;
34     pthread_create(&t1, NULL, fThread1, NULL);
35     pthread_create(&t2, NULL, fThread2, NULL);
36
37     pthread_join(t1, NULL);
38     pthread_join(t2, NULL);
39
40     printf("=: %d\n",a);
41 }
```

In this program you can see I used a semaphore to solve the race condition problem we had in part 1.

we speak of synchronized threads because as you can see, after initializing it in the main, the second thread will wait for a signal saying that the semaphore can be used. That signal will be sent by the first thread, but only once it has done its part of the work!

That way, the race condition is solved.

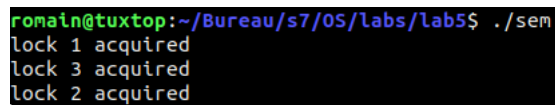
Now, if I were to use more processes, as you will be able to see in question 3, I would need to synchronize them as well by defining for example an order of execution, or only allowing a thread or a process to proceed with its execution if some condition is met.

### 0.3.2 Q2:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/wait.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <unistd.h>
7  #include <semaphore.h>
8  #include <pthread.h>
9
10 sem_t mut1, mut2, mut3;
11
12 void * fThread1(void *arg)
13 {
14     sem_wait(&mut1);
15     printf("lock 1 acquired\n");
16     sleep(2);
17     sem_wait(&mut2);
18     printf("lock 2 acquired\n");
19     sleep(2);
20     sem_post(&mut2);
21     printf("lock 2 released\n");
22     sem_post(&mut1);
23     printf("lock 1 released\n");
24 }
25
26 void * fThread2(void *arg)
27 {
28     sem_wait(&mut2);
29     printf("lock 2 acquired\n");
30     sleep(2);
31     sem_wait(&mut3);
32     printf("lock 3 acquired\n");
33     sleep(2);
34     sem_post(&mut3);
35     printf("lock 3 released\n");
36     sem_post(&mut2);
37     printf("lock 2 released\n");
38 }
39
40 void * fThread3(void *arg)
41 {
42     sem_wait(&mut3);
43     printf("lock 3 acquired\n");
44     sleep(2);
45     sem_wait(&mut1);
46     printf("lock 1 acquired\n");
47     sleep(2);
48     sem_post(&mut1);
49     printf("lock 1 released\n");
```



```
50     sem_post(&mut3);
51     printf("lock 3 released\n");
52 }
53
54 int main(int argc, char **argv)
55 {
56     sem_init(&mut1,0,1);
57     sem_init(&mut2,0,1);
58     sem_init(&mut3,0,1);
59     pthread_t t1, t2, t3;
60     pthread_create(&t1, NULL, fThread1, NULL);
61     pthread_create(&t2, NULL, fThread2, NULL);
62     pthread_create(&t3, NULL, fThread3, NULL);
63     pthread_join(t1,NULL);
64     pthread_join(t2,NULL);
65     pthread_join(t3,NULL);
66
67     sem_destroy(&mut1);
68     sem_destroy(&mut2);
69     sem_destroy(&mut3);
70 }
```



```
romain@tuxtop:~/Bureau/s7/OS/labs/lab5$ ./sem
lock 1 acquired
lock 3 acquired
lock 2 acquired
```

Figure 2: deadlocks are fun

The execution of this code results in a deadlock situation. Because each thread will wait for one another to unblock itself and free some resource, however it will never happen since I created a cycle of waiting here. A deadlock is a common example of what can happen when you don't synchronize processes correctly!

As you can see on the screenshot the execution indeed results in each thread acquiring a lock and then everything is blocked, they are mutually waiting for each other, forming a cycle, an eternal loop.

### 0.3.3 Q3:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/wait.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <unistd.h>
7  #include <semaphore.h>
8  #include <pthread.h>
9
10
11  sem_t mutex1, mutex2;
12
13  void *fThread1(void *arg)
14  {
15      system("opera");
16      sem_post(&mutex1);
17  }
18  void *fThread2(void *arg)
19  {
20      sem_wait(&mutex1);
21      system("vi");
22      sem_post(&mutex2);
23  }
24  void *fThread3(void *arg)
25  {
26      sem_wait(&mutex2);
27      system("gnnumeric");
28  }
29
30  int main(int argc, char **argv)
31  {
32      pthread_t t1, t2, t3;
33
34      sem_init(&mutex1,0,0);
35      sem_init(&mutex2,0,0);
36
37      pthread_create(&t1, NULL, fThread1, NULL);
38      pthread_create(&t2, NULL, fThread2, NULL);
39      pthread_create(&t3, NULL, fThread3, NULL);
40
41      pthread_join(t1, NULL);
42      pthread_join(t2, NULL);
43      pthread_join(t3, NULL);
44
45
46      sem_destroy(&mutex1);
47      sem_destroy(&mutex2);
48  }
```

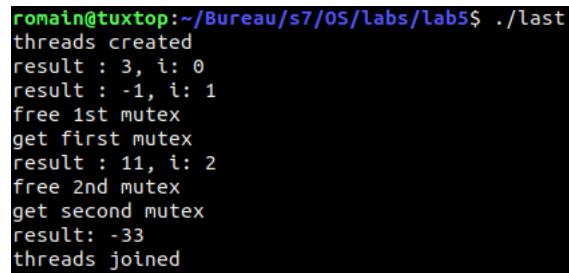
As promised in question 1, this is a way of synchronizing three processes using two semaphores. I did it in a way that they will always execute in the same order, it is a predefined sequence.

### 0.3.4 Q4:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/wait.h>
4  #include <sys/types.h>
5  #include <sys/shm.h>
6  #include <sys/mman.h>
7  #include <unistd.h>
8  #include <semaphore.h>
9  #include <pthread.h>
10
11 sem_t mutex1, mutex2, muti;
12 int a=1, b=2, c=3, d=4, e=5, f=6, i=-1;
13 int results[3];
14
15 void *fThread1(void *arg)
16 {
17     sem_wait(&muti);
18     i++;
19     results[i]=a+b;
20     printf("result : %d, i: %d\n",results[i],i);
21     sem_post(&muti);
22     if(i==1)
23     {
24         sem_post(&mutex1);
25         printf("free 1st mutex\n");
26     }
27     else if(i==2)
28     {
29         sem_post(&mutex2);
30         printf("free 2nd mutex\n");
31     }
32 }
33
34 void *fThread2(void *arg)
35 {
36     sem_wait(&muti);
37     i++;
38     results[i]=c-d;
39     printf("result : %d, i: %d\n",results[i],i);
40     if(i==1)
41     {
42         sem_post(&mutex1);
43         printf("free 1st mutex\n");
44     }
45     else if(i==2)
46     {
47         sem_post(&mutex2);
48         printf("free 2nd mutex\n");
49     }
```

```
50     sem_post(&muti);
51 }
52
53 void *fThread3(void *arg)
54 {
55     sem_wait(&muti);
56     i++;
57     results[i]=e+f;
58     printf("result : %d, i: %d\n",results[i],i);
59     if(i==1)
60     {
61         sem_post(&mutex1);
62         printf("free 1st mutex\n");
63     }
64     else if(i==2)
65     {
66         sem_post(&mutex2);
67         printf("free 2nd mutex\n");
68     }
69     sem_post(&muti);
70 }
71
72 void *fThread4(void *arg)
73 {
74     sem_wait(&mutex1);
75     printf("get first mutex\n");
76
77     results[0]=results[0]*results[1];
78
79     sem_wait(&mutex2);
80     printf("get second mutex\n");
81
82     printf("1: %d, 2: %d, 3: %d\n",results[0],results[1],results[2]);
83
84     results[0]=results[0]*results[2];
85     printf("result: %d\n",results[0]);
86 }
87
88 int main(int argc, char **argv)
89 {
90     sem_init(&mutex1,0,0);
91     sem_init(&mutex2,0,0);
92     sem_init(&muti,0,1);
93     pthread_t t1, t2, t3, t4;
94
95     pthread_create(&t1, NULL, fThread1, NULL);
96     pthread_create(&t2, NULL, fThread2, NULL);
97     pthread_create(&t3, NULL, fThread3, NULL);
98     pthread_create(&t4, NULL, fThread4, NULL);
99     printf("threads created\n");
```

```
100
101     pthread_join(t1, NULL);
102     pthread_join(t2, NULL);
103     pthread_join(t3, NULL);
104     pthread_join(t4, NULL);
105     printf("threads joined\n");
106 }
```



```
romain@tuxtop:~/Bureau/s7/OS/labs/lab5$ ./last
threads created
result : 3, i: 0
result : -1, i: 1
free 1st mutex
get first mutex
result : 11, i: 2
free 2nd mutex
get second mutex
result: -33
threads joined
```

Figure 3: Synchronization of three processes

In this code I used 3 semaphores, 2 of them used to control that the parts of the calculus were done in the right order, and the last one to check that no thread was updating the value of the results array concurrently, creating once again a race condition. That way I synchronized processes using semaphores in order to solve a math problem.