

به نام خدا  
دانشگاه صنعتی امیرکبیر (پلی تکنیک تهران)  
دانشکده مهندسی کامپیوتر

## آزمایشگاه سیستم‌های عامل

### گزارش کار آزمایش 6

فرشید نوشی - ۹۸۳۱۰۶۸

## بخش ۱:

برای اینکه در در نقاط بحرانی اجازه‌ی انجام عملیات به process های دیگر را ندهیم از mutex استفاده میکنیم.

ابتدا یک struct به نام Buffer تعریف می‌کنیم که در آن دو متغیر عدد صحیح برای شمارنده و شمارنده‌ی خواننده و دو متغیر سمافور نیز برای قفل کردن هر کدام وجود دارد.

```
15     typedef struct {
16         int count;
17         int reader_count;
18         sem_t count_mutex;
19         sem_t reader_count_mutex;
20     } Buffer;
21
```

سپس در main آبجکت از نوع Buffer، از حافظه‌ی اشتراکی attach می‌کنیم و مقدار اولیه‌ی count و reader\_count را صفر می‌گذاریم. و متغیرهای سمافور را نیز مقداردهی اولیه می‌کنیم و حافظه را detach می‌کنیم و 5 پردازشی فرزند با fork ایجاد می‌کنیم.

```
22     int main(int argc, char* argv[])
23     {
24         int status = 0;
25         int parent_pid = getpid();
26
27         Buffer *buffer;
28         int sh_writer = shmget(0x125, 32, S_IRUSR | S_IWUSR | IPC_CREAT);
29         int sh_reader = shmget(0x125, 32, S_IRUSR | IPC_CREAT);
30         buffer = (Buffer *)shmat(sh_writer, NULL, 0);
31         buffer->count = 0;
32         buffer->reader_count = 0;
33         sem_init(&(buffer->count_mutex), 1, 1);
34         sem_init(&(buffer->reader_count_mutex), 1, 1);
35         shmdt(buffer);
36
37         for (int i = 0; i < 4; i++)
38         {
39             if (parent_pid == getpid())
40             {
41                 fork();
42                 printf("forked\n");
43             }
44         }
```

هنگامی که یک reader در حال خواندن است جلوی writer را با استفاده از count\_mutex میگیریم در ادامه هنگامی که هیچ reader در حال خواندن نبود به writer خبر میدهم که آزاد است تا کارش را انجام دهد برای اینکار از یک متغیر شمارنده برای تعداد reader های در حال خواندن استفاده میکنم که با reader\_count\_mutex قفل میشود تا بقیه ی reader ها نتوانند به آن دسترسی داشته باشند هنگام نوشتن یک reader تا از race condition جلوگیری شود. متغیری که تعداد reader های در حال خواندن را نشان میدهد نامش reader\_count میباشد.

```

46
47 if (parent_pid == getpid())
48 {
49     buffer = (Buffer *) shmctl(sh_writer, NULL, 0);
50     for (int i = 1; i <= MAX_COUNT; i++)
51     {
52         sem_wait(&(buffer->count_mutex));
53         printf("Parent-Writing %d\n", parent_pid, i);
54         buffer->count = i;
55         sem_post(&(buffer->count_mutex));
56         sleep(0.1);
57     }
58     shmctl(buffer);
59 }
60 else
61 {
62     int child = getpid();
63     int flag = 1;
64     buffer = (Buffer *) shmctl(sh_reader, NULL, 0);
65
66     while(flag)
67     {
68         sem_wait(&(buffer->reader_count_mutex));
69         buffer->reader_count++;
70         if (buffer->reader_count == 1)
71         {
72             sem_wait(&(buffer->count_mutex));
73         }
74         sem_post(&(buffer->reader_count_mutex));
75
76         printf("Child-Writing %d\n", child, buffer->count);
77         if (buffer->count >= MAX_COUNT)
78         {
79             flag = 0;
80         }
81
82         sem_wait(&(buffer->reader_count_mutex));
83         buffer->reader_count--;
84         if (buffer->reader_count == 0)
85         {
86             sem_post(&(buffer->count_mutex));
87         }
88         sem_post(&(buffer->reader_count_mutex));
89         sleep(0.05);
90     }
91     shmctl(buffer);
92 }
93
94

```

حلقه ی while در آخر برای مطمئن شدن از اتمام کار پردازش های فرزند هست.

```

94
95     if (parent_pid == getpid())
96     {
97         while (wait(&status) > 0);
98         shmctl(sh_writer, IPC_RMID, NULL);
99     }
100
101     return 0;
102 }
103

```

دو نمونه خروجی را که read و write دچار شرایط مسابقه نشده اند.

Activities

Terminal

Dec 10 07:16

farshid@ubuntu: ~/Desktop/Lab6/sources

```
farshid@ubuntu:~/Desktop/Lab6/sources$ gcc -pthread -o ql ql.c
farshid@ubuntu:~/Desktop/Lab6/sources$ ./ql
forked
forked
Child-3917: 0
forked
forked
Child-3918: 0
forked
forked
Child-3919: 0
forked
Parent-3916: Writing 1
Child-3918: 1
forked
Child-3920: 1
Child-3917: 1
Child-3919: 1
Child-3918: 1
Child-3920: 1
Parent-3916: Writing 2
Parent-3916: Writing 3
Child-3917: 3
Child-3919: 3
Child-3918: 3
Child-3920: 3
Parent-3916: Writing 4
Child-3917: 4
Child-3920: 4
Child-3918: 4
Child-3919: 4
Child-3917: 4
Parent-3916: Writing 5
Child-3919: 5
Child-3920: 5
```

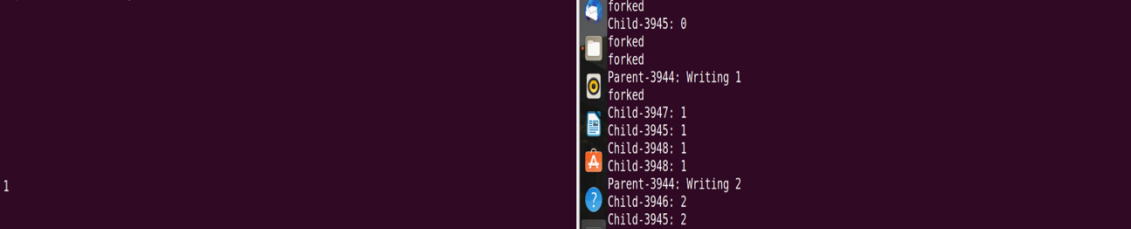
Activities

Terminal

Dec 10 07:16

farshid@ubuntu: ~/Desktop/Lab6/sources

```
forked
Child-3917: 0
forked
forked
Child-3918: 0
forked
forked
Child-3919: 0
forked
Parent-3916: Writing 1
Child-3918: 1
forked
Child-3920: 1
Child-3917: 1
Child-3919: 1
Child-3918: 1
Child-3920: 1
Parent-3916: Writing 2
Parent-3916: Writing 3
Child-3917: 3
Child-3919: 3
Child-3918: 3
Child-3920: 3
Parent-3916: Writing 4
Child-3917: 4
Child-3920: 4
Child-3918: 4
Child-3919: 4
Child-3917: 4
Parent-3916: Writing 5
Child-3919: 5
Child-3920: 5
Child-3917: 5
Child-3918: 5
farshid@ubuntu:~/Desktop/Lab6/sources$
```



```
farshid@ubuntu: ~/Desktop/Lab6/sources
farshid@ubuntu:~/Desktop/Lab6/sources$ ./q1
forked
forked
forked
Child-3945: 0
forked
Child-3946: 0
forked
Child-3945: 0
forked
Parent-3944: Writing 1
forked
Child-3947: 1
Child-3945: 1
Child-3948: 1
Child-3948: 1
Parent-3944: Writing 2
Child-3946: 2
Child-3945: 2
Child-3947: 2
Child-3948: 2
Child-3946: 2
Child-3947: 2
Parent-3944: Writing 3
Child-3945: 3
Child-3947: 3
Child-3948: 3
Child-3946: 3
Parent-3944: Writing 4
Child-3945: 4
Child-3948: 4
Child-3945: 4
Child-3947: 4
Child-3946: 4

farshid@ubuntu:~/Desktop/Lab6/sources$
```

```
farshid@ubuntu:~/Desktop/Lab6/sources$ ./q1
Child-3946: 0
forked
Child-3945: 0
forked
Parent-3944: Writing 1
forked
Child-3947: 1
Child-3945: 1
Child-3948: 1
Child-3948: 1
Parent-3944: Writing 2
Child-3946: 2
Child-3945: 2
Child-3947: 2
Child-3948: 2
Child-3946: 2
Child-3947: 2
Parent-3944: Writing 3
Child-3945: 3
Child-3947: 3
Child-3948: 3
Child-3946: 3
Parent-3944: Writing 4
Child-3945: 4
Child-3948: 4
Child-3945: 4
Child-3947: 4
Child-3946: 4
Parent-3944: Writing 5
Child-3946: 5
Child-3948: 5
Child-3945: 5
Child-3947: 5
farshid@ubuntu:~/Desktop/Lab6/sources$
```

## بخش ۲:

در این بخش می‌خواهیم از بروز شرایط مسابقه جلوگیری کنیم.

ابتدا تابع philosopher را توضیح می‌دهیم. نوع این تابع از نوع `void*` هست تا به `pthread_create()` پاس داده بشود. با توجه به تعداد دفعه‌های خوردن در هر مرحله تابع `eat` صدا زده میشود که چوب‌های مربوط به فیلسوف شماره `n` را قفل میکند و غذایش را میخورد و در ادامه چوب‌هایش را آزاد میکند. تابع برحسب شماره `n` ورودی خروجی مناسب را تولید میکند.

در `main` پنج نخ تعریف میکنیم (تعداد فیلسوف‌ها). در ادامه با استفاده از تابع `pthread_mutex_init` پنج لاک برای استفاده میسازیم. برای ساخت `thread` از تابع `pthread_create` استفاده میکنیم که تابع `philosopher` را بهش میدهم تا کار فیلسوف را بکند. برای اطمینان از اینکه همه `thread`‌ها کارشان تمام میشود از `pthread_join` روی همه `thread`‌ها استفاده میکنیم.

```
10 #define MAX_EAT_TIMES 1
11 #define NUMBER_OF_PHILOSOPHERS 5
12
13 pthread_mutex_t chopstick[5];
14
15
16 void eat(int n)
17 {
18     int next = (n + 1) % NUMBER_OF_PHILOSOPHERS;
19     pthread_mutex_lock(&chopstick[n]);
20     pthread_mutex_lock(&chopstick[next]);
21
22     printf("Philosopher %d is eating using chopstick[%d] and chopstick[%d].\n", n + 1, n, next);
23     sleep(1);
24
25     pthread_mutex_unlock(&chopstick[n]);
26     pthread_mutex_unlock(&chopstick[next]);
27 }
28
29 void *philosopher(int n)
30 {
31     for (int i = 0; i < MAX_EAT_TIMES; i++)
32     {
33         printf("Philosopher %d is thinking.\n", n + 1);
34         eat(n);
35         printf("Philosopher %d finished eating.\n", n + 1);
36     }
37 }
38
39
40 int main()
41 {
42     pthread_t threads[NUMBER_OF_PHILOSOPHERS];
43
44     for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++)//initializing mutex
45     {
46         pthread_mutex_init(&chopstick[i], NULL);
47     }
48
49     for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++)//initializing threads
50     {
51         pthread_create(&threads[i], NULL, (void *) philosopher, (void *) (intptr_t) i);
52     }
53
54     for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++)//wait to end threads
55     {
56         pthread_join(threads[i], NULL);
57     }
58
59     return 0;
60 }
```

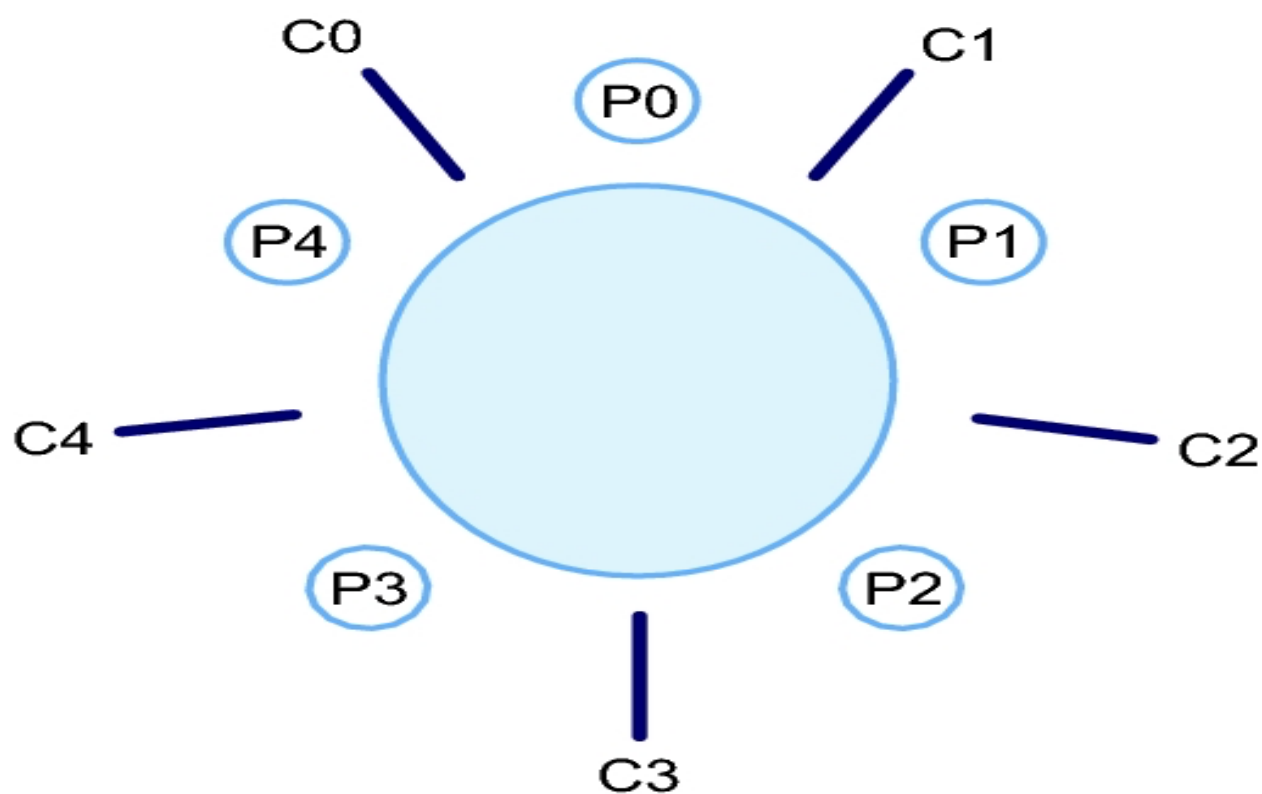
در ادامه دو نمونه خروجی را می‌بینیم:

با توجه به نمونه خروجی‌های زیر، برنامه به طور صحیح کار می‌کند و هرگز یک چوب همزمان در اختیار دو فیلسوف قرار نمی‌گیرد.

```
farshid@ubuntu:~/Desktop/Lab6/sources$ gcc -pthread -o q2 q2.c
farshid@ubuntu:~/Desktop/Lab6/sources$ ./q2
Philosopher 1 is thinking.
Philosopher 1 is eating using chopstick[0] and chopstick[1].
Philosopher 4 is thinking.
Philosopher 4 is eating using chopstick[3] and chopstick[4].
Philosopher 2 is thinking.
Philosopher 5 is thinking.
Philosopher 3 is thinking.
Philosopher 1 finished eating.
Philosopher 4 finished eating.
Philosopher 5 is eating using chopstick[4] and chopstick[0].
Philosopher 3 is eating using chopstick[2] and chopstick[3].
Philosopher 5 finished eating.
Philosopher 2 is eating using chopstick[1] and chopstick[2].
Philosopher 3 finished eating.
Philosopher 2 finished eating.
farshid@ubuntu:~/Desktop/Lab6/sources$ ./q2
Philosopher 1 is thinking.
Philosopher 1 is eating using chopstick[0] and chopstick[1].
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 3 is eating using chopstick[2] and chopstick[3].
Philosopher 4 is thinking.
Philosopher 5 is thinking.
Philosopher 1 finished eating.
Philosopher 2 is eating using chopstick[1] and chopstick[2].
Philosopher 5 is eating using chopstick[4] and chopstick[0].
Philosopher 3 finished eating.
Philosopher 2 finished eating.
Philosopher 4 is eating using chopstick[3] and chopstick[4].
Philosopher 5 finished eating.
Philosopher 4 finished eating.
farshid@ubuntu:~/Desktop/Lab6/sources$
```

آیا ممکن است بن بست رخ دهد؟ در صورت امکان چگونگی ایجاد آن را توضیح دهید.

بله ممکن است چون احتمال اینکه اگر هر فیلسوف روی یک چوب غذا قفل ایجاد کند بن بست به وجود می آید وجود دارد چون هیچ دو چوبی خالی نیستند هیچ غذایی خورده نمیشود به بن بست میرسیم.



P = Philosopher  
C = Chopstick