آزمایشگاه سیستم عامل

گزارش آزمایش شماره ۶

نام استاد: مهندس حمیدرضا کیخا

تاریخ: ۲۲ / ۹ / ۱۴۰۰

نام و نام‌خانوادگی: نگار کرمی

شماره دانشجویی: ۹۷۲۲۰۳۹

نام و نام‌خانوادگی: مجید نامی

شماره دانشجویی: ۹۷۲۸۰۹۵

بخش اول: مساله خوانندگان-نویسندگان

کد:

```
12  typedef struct Shared Shared;
13
14
15  struct Shared
16  {
17      int count, rc;
18      pthread_mutex_t mutex;
19      pthread_mutex_t rw_mutex;
20  };
21
22  int shmid;
23
24  void reader()
25  {
26      Shared* shared = (Shared *)shmat(shmid, NULL, 0);
27      int pid = getpid();
28
29      bool done = 0;
30      while (!done)
31      {
32          pthread_mutex_lock(&(shared->mutex));
33          (shared->rc)++;
34          if (shared->rc == 1)
35              pthread_mutex_lock(&(shared->rw_mutex));
36          pthread_mutex_unlock(&(shared->mutex));
37
38          printf("Reader:\tPID: %d\tcount: %d\n", pid, shared->count);
39          if (shared->count >= 8)
40              done = true;
41
42          pthread_mutex_lock(&(shared->mutex));
43          (shared->rc)--;
44          if (shared->rc == 0)
45              pthread_mutex_unlock(&(shared->rw_mutex));
46          pthread_mutex_unlock(&(shared->mutex));
47          sleep(0.1);
48      }
49
50      shmdt(shared);
51  }
```

```
53  void writer()
54  {
55      Shared* shared = (Shared *)shmat(shmid, NULL, 0);
56      int pid = getpid();
57
58      bool done = 0;
59      while (!done)
60      {
61          pthread_mutex_lock(&(shared->rw_mutex));
62
63          shared->count++;
64          printf("Writer:\tPID: %d\tcount: %d\n", pid, shared->count);
65          if (shared->count >= 8)
66              done = true;
67          pthread_mutex_unlock(&(shared->rw_mutex));
68          sleep(0.5);
69      }
70
71      shmdt(shared);
72  }
```

```
74  int main(int argc, char const *argv[])
75  {
76      int pid;
77      int base_pid = getpid();
78
79      shmid = shmget(IPC_PRIVATE, sizeof(Shared), IPC_CREAT | 0666);
80
81      Shared* sh_at;
82      sh_at = (Shared *)shmat(shmid, NULL, 0);
83
84
85      pthread_mutexattr_t attr;
86      pthread_mutexattr_init(&attr);
87      pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
88
89      pthread_mutex_init(&(sh_at->mutex), &attr);
90      pthread_mutex_init(&(sh_at->rw_mutex), &attr);
91
92      shmdt(sh_at);
93
```

```
94
95      pid = fork();
96      if (pid == 0)
97      {
98          writer();
99          return 0;
100     }
101
102     for (int i = 0; i < 2; i++)
103         if (getpid() == base_pid)
104             pid = fork();
105         else
106             break;
107
108     if (pid == 0)
109     {
110         reader();
111         return 0;
112     }
113
114
115     if (getpid() == base_pid)
116     {
117         wait(NULL);
118         for (int i = 0; i < 2; i++)
119             wait(NULL);
120     }
121
122     shmctl(shmid, IPC_RMID, NULL);
123 }
```

خروجی:

```
negar@ubuntu:~/Desktop$ gcc Part1.c -lpthread -o output1.out
negar@ubuntu:~/Desktop$ ./output1.out
Writer: PID: 5869       count: 1
Reader: PID: 5871       count: 1
Reader: PID: 5870       count: 1
Reader: PID: 5871       count: 1
Writer: PID: 5869       count: 2
Reader: PID: 5871       count: 2
Writer: PID: 5869       count: 3
Reader: PID: 5870       count: 3
Reader: PID: 5871       count: 3
Reader: PID: 5870       count: 3
Writer: PID: 5869       count: 4
Reader: PID: 5870       count: 4
Reader: PID: 5871       count: 4
Writer: PID: 5869       count: 5
Reader: PID: 5870       count: 5
Writer: PID: 5869       count: 6
Reader: PID: 5871       count: 6
Reader: PID: 5870       count: 6
Reader: PID: 5871       count: 6
Reader: PID: 5870       count: 6
Writer: PID: 5869       count: 7
Reader: PID: 5870       count: 7
Reader: PID: 5871       count: 7
Reader: PID: 5870       count: 7
Reader: PID: 5871       count: 7
Writer: PID: 5869       count: 8
Reader: PID: 5870       count: 8
Reader: PID: 5871       count: 8
negar@ubuntu:~/Desktop$
```

دو عدد writer و یک عدد reader به مقدار count دسترسی دارند. در struct که ساخته شده است دو قفل mutex وجود دارد که یکی mutex و دیگری rw_mutex می‌باشد. که برای رفع مشکل race condition و برقراری شرط انحصار متقابل استفاده شده است. همچنین متغیر rc تعداد خواننده‌هایی هست که در حال خواندن داده هستند و چون همین متغیر rc هم یک ناحیه بحرانی محسوب می‌شود، پس از mutex استفاده شده است. همچنین بررسی شده است که اگر اولین فرآیند reader باشد، ما rw_mutex را lock کنیم تا writer حق نوشتن نداشته باشد و بعد از اتمام ناحیه بحرانی دوباره rw_mutexرا unlock می‌کنیم. همچنین سقف متغیر count را برابر ۸ در نظر گرفته‌ایم.

برنامه مربوطه را بصورت کامل نوشته و سپس اجرا کنید. آیا مشکلی وجود دارد؟

در لحظه‌ی مشابه امکان دارد خروجی (مقدار count) یکسانی نداشته باشیم.

در صورت وجود ناهماهنگی چه راهکاری ارائه می‌کنید؟

زمانی که یک writer در حال کار کردن با count است، هیچ فرآیند دیگری (writer or reader) نباید به count دسترسی داشته باشد. اما در حالتی که یک Reader در حال خواندن count است، یک یا چند reader دیگر می‌تواند به count دسترسی داشته باشد.

بخش دوم: مساله فیلسوف‌های غذاخور

کد:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>


pthread_mutex_t chopstick[6];

void *philosopher(int n)
{
    for (int i = 0; i < 2; i++)
    {
        printf("Philosopher %d is thinking!!!\n", n);

        pthread_mutex_lock(&chopstick[n - 1]);
        pthread_mutex_lock(&chopstick[n % 5]);

        printf("Philosopher %d is eating using chopstick(%d) and chopstick(%d)!!!\n", n, n - 1, n % 5);
        sleep(1);

        pthread_mutex_unlock(&chopstick[n - 1]);
        pthread_mutex_unlock(&chopstick[n % 5]);

        printf("Philosopher %d finished eating!!\n", n);
    }
}

int main()
{
    pthread_t threads[6];

    for (int i = 1; i <= 5; i++)
        pthread_mutex_init(&chopstick[i], NULL);

    for (int i = 1; i <= 5; i++)
        pthread_create(&threads[i], NULL, (void *) philosopher, (void *)(intptr_t) i);

    for (int i = 1; i <= 5; i++)
        pthread_join(threads[i], NULL);
}
```

در کدی که پیاده سازی شده است از ۵ عدد lock به صورت آرایه استفاده شده است که برای تضمین انحصار متقابل برای Thread هر یک از ۵ فیلسوف استفاده شده است. در خروجی بالا هر فیلسوف دو بار فرآیند خوردن را انجام می‌دهد.

در این روش، انحصار متقابل تضمین شده است اما امکان رخ دادن بن‌بست وجود دارد. به عنوان مثال هنگامی که همه‌ی فیلسوف‌ها در ابتدا قفل اول خود را قفل کنند، در شرایطی که کسی قفل دوم را قفل نکرده باشد. در این حالت امکان بن‌بست داریم.

خروجی:

```
negar@ubuntu:~/Desktop$ gcc Part2.c -lpthread -o output2.out
negar@ubuntu:~/Desktop$ ./output2.out
Philosopher 1 is thinking!!!
Philosopher 1 is eating using chopstick(0) and chopstick(1)!!!
Philosopher 3 is thinking!!!
Philosopher 3 is eating using chopstick(2) and chopstick(3)!!!
Philosopher 5 is thinking!!!
Philosopher 2 is thinking!!!
Philosopher 4 is thinking!!!
Philosopher 1 finished eating!!
Philosopher 1 is thinking!!!
Philosopher 1 is eating using chopstick(0) and chopstick(1)!!!
Philosopher 3 finished eating!!
Philosopher 3 is thinking!!!
Philosopher 3 is eating using chopstick(2) and chopstick(3)!!!
Philosopher 1 finished eating!!
Philosopher 3 finished eating!!
Philosopher 2 is eating using chopstick(1) and chopstick(2)!!!
Philosopher 5 is eating using chopstick(4) and chopstick(0)!!!
Philosopher 2 finished eating!!
Philosopher 2 is thinking!!!
Philosopher 2 is eating using chopstick(1) and chopstick(2)!!!
Philosopher 4 is eating using chopstick(3) and chopstick(4)!!!
Philosopher 5 finished eating!!
Philosopher 5 is thinking!!!
Philosopher 2 finished eating!!
Philosopher 4 finished eating!!
Philosopher 4 is thinking!!!
Philosopher 5 is eating using chopstick(4) and chopstick(0)!!!
Philosopher 5 finished eating!!
Philosopher 4 is eating using chopstick(3) and chopstick(4)!!!
Philosopher 4 finished eating!!
negar@ubuntu:~/Desktop$
```

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

بله، امکان دارد. برای مثال در لحظه‌ی ابتدایی اگر تمام فیلسوف‌ها چنگال‌های خود سمت چپ خود را در دست داشته باشند، آنگاه یک Dead lock ویا بن‌بست رخ می‌دهد و همچنین در هر زمانی از آینده نیز اگر تمام فیلسوف‌ها چنگال سمت چپ خود را بدست گرفته باشند آنگاه به بن بست می‌خوریم.

راه حل اول: با قرار دادن یک Scheduler یک ترتیب و برنامه به خوردن فیلسوف‌ها بدهیم.

راه حل دوم: به هر کدام از فیلسوف‌ها یک اولویت بدهیم که در زمان بن بست حق استفاده از چنگال را به فیلسوف با اولویت بالاتر بدهیم.