به نام خدا

محمد جواد زندیه 9831032

آزمایش شماره 6 سیستم عامل

بخش اول: مساله خوانندگان-نویسندگان را پیاده سازی کنید.

reader_writer.c

home > javad > Desktop > OSLab > project6 > reader_writer.c > main(int, char const * [])

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
# include <sys/shm.h>
# include <sys/ipc.h>

int max_count = 3;

/* without synchronization, processes may be…
void do_read_write(int proc_task, int* count){
    if(proc_task == 0){ // write into shared memory(count)
        *count = *count + 1;
        pid_t pid = getpid();
        printf("Task: Write, PID: %d, count: %d\n", pid, *count);
    }
    else{ //read from shared memory(count)
        pid_t pid = getpid();
        printf("Task: Read , PID: %d, count: %d\n", pid, *count);
    }
}
```

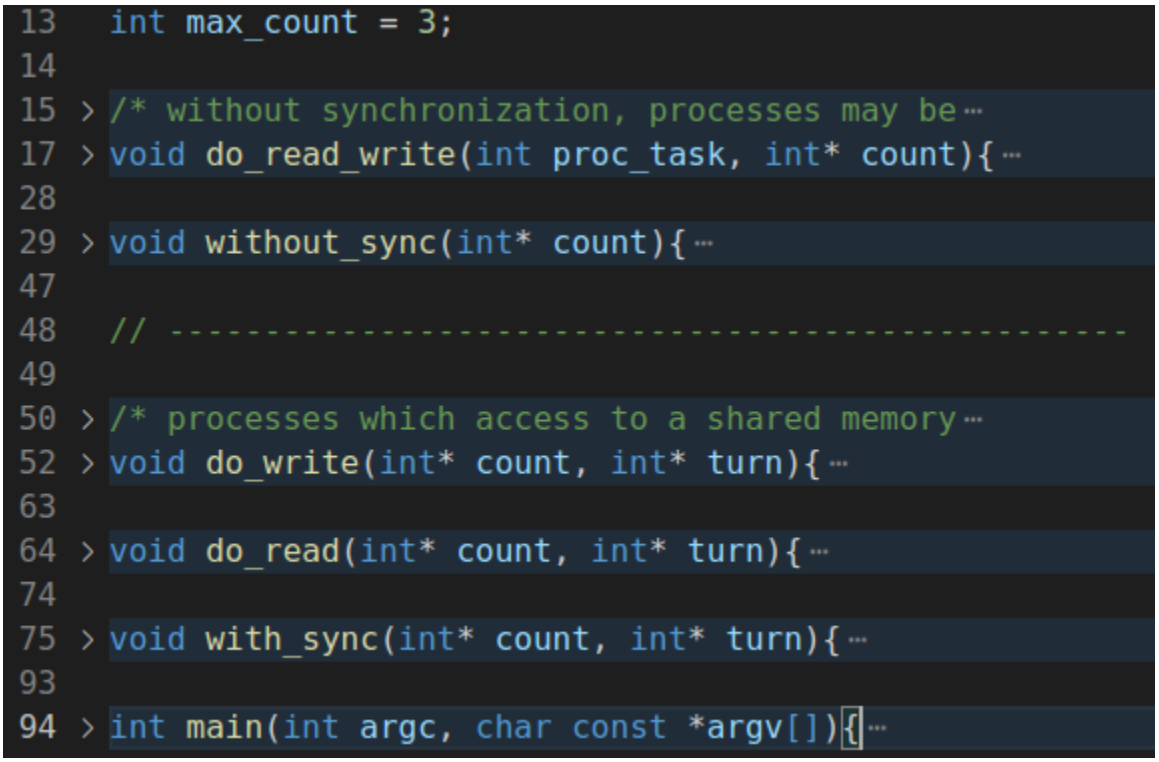
```
28
29  void without_sync(int* count){
30      // three processes
31      if(fork()){ //reader1
32          while(*count < max_count)
33              do_read_write(1, count);
34          exit(0);
35      }else{
36          if(fork()){ //reader2
37              while(*count < max_count)
38                  do_read_write(1, count);
39              exit(0);
40          }else{ //writer
41              while(*count < max_count)
42                  do_read_write(0, count);
43              wait(NULL);
44          }
45      }
46  }
```

```
48  // -------------------------------------------------
49
50  /* processes which access to a shared memory
51     must be sync to prevent race condition */
52  void do_write(int* count, int* turn){
53      *turn = 0;
54
55      while(*turn == 1);
56      // critical section
57      *count = *count + 1;
58      pid_t pid = getpid();
59      printf("Task: Write, PID: %d, count: %d\n", pid, *count);
60
61      *turn = 1;
62  }
63
64  void do_read(int* count, int* turn){
65      *turn = 1;
66
67      while(*turn == 0);
68      // critical section
69      pid_t pid = getpid();
70      printf("Task: Read , PID: %d, count: %d\n", pid, *count);
71
72      *turn = 0;
73  }
```

```
75  void with_sync(int* count, int* turn){
76      // three processes
77      if(fork()){ //reader1
78          while(*count < max_count)
79              do_read(count, turn);
80          exit(0);
81      }else{
82          if(fork()){ //writer
83              while(*count < max_count)
84                  do_write(count, turn);
85              exit(0);
86          }else{ //reader2
87              while(*count < max_count)
88                  do_read(count, turn);
89              wait(NULL);
90          }
91      }
92  }
```

```
94  int main(int argc, char const *argv[]){
95      // count = shm[0],
96      int shm_id = shmget(IPC_PRIVATE, 20, IPC_CREAT| SHM_R | SHM_W);
97      int* shm = (int *) shmat(shm_id, 0, 0);
98      shm[0] = 0; // count
99      shm[1] = 0; // turn 0 for writer and 1 for readers
100
101     printf("\nincreasing count without synchronization:\n");
102     without_sync(&shm[0]);
103
104     printf("\nincreasing count with synchronization:\n");
105     with_sync(&shm[0], &shm[1]);
106
107     shmctl(shm_id, IPC_RMID, NULL);
108     return 0;
109 }
```

یک شمای کلی از کد به این صورت می باشد



در این مسئله دو عدد reader و یک عدد writer داریم که reader ها همزمان اجازه دسترسی به ناحیه بحرانی یعنی خواندن مقدار count دارند اما reader و writer همزمان اجازه دسترسی به مقدار count را ندارند و باید با گذاشتن قفل روی ناحیه بحرانی از دسترسی همزمان خواننده و نویسنده به مقدار count جلوگیری کنیم. در این مسئله چون reader ها همزمان می توانند به مقدار count دسترسی داشته باشند در واقع میتوان مسئله را به مسئله Peterson تبدیل کرد و برای خواننده ها با turn = 1 و برای نویسنده با turn = 0 ناحیه بحرانی را کنترل کرد. این الگوریتم به این صورت عمل می کند که تا نوبت پردازه نرسیده باشد که وارد بشود حق ورود به ناحیه بحرانی را ندارد.

خروجی به این صورت می باشد:

```
javad@javad-HP-350-G1:~/Desktop/OSLab/project6$ ./reader_writer

increasing count with synchronization:
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 0
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Read , PID: 8427, count: 1
Task: Write, PID: 8428, count: 1
Task: Read , PID: 8427, count: 1
Task: Write, PID: 8428, count: 2
Task: Read , PID: 8427, count: 2
Task: Write, PID: 8428, count: 3
javad@javad-HP-350-G1:~/Desktop/OSLab/project6$
```

همانطور که مشاهده می شود پردازه ها طوری عمل نکرده اند که وقتی عمل خواندن انجام می شود به عنوان مثال بتواند همزمان عمل نوشتن انجام شود در واقع از روی اینکه مقدار count به صورت صعودی است می توان فهمید که عملیات خواندن به صورت اتومیک صورت گرفته و عملیات نوشتن هم به همین صورت.

اگر روی ناحیه بحرانی قفل گذاشته نمیشد به این صورت می بود که از به هم ریخته بودن و غیر صعودی بودن مقدار count که چاپ شده است میتوان متوجه شد که پردازه ها بدون توجه به اینکه دسترسی به count بحرانی است همزمان نوشتن و خواندن روی این داده را انجام داده اند.

```
Task: Read , PID: 10996, shm_count: 1
Task: Read , PID: 10996, shm_count: 1
Task: Read , PID: 10996, shm_count: 1
Task: Read , PID: 10996, shm_count: 1
Task: Read , PID: 10996, shm_count: 1
Task: Read , PID: 10996, shm_count: 1
Task: Read , PID: 10997, shm_count: 0
Task: Write, PID: 10998, shm_count: 1
Task: Read , PID: 10996, shm_count: 1
Task: Read , PID: 10997, shm_count: 1
Task: Write, PID: 10998, shm_count: 2
Task: Read , PID: 10996, shm_count: 2
Task: Read , PID: 10997, shm_count: 2
Task: Write, PID: 10998, shm_count: 3
javad@javad-HP-350-G1:~/Desktop/OSLab/project6$
```

بخش دوم: مساله فیلسوف های غذا خور

C reader_writer.c C Dining_Philosophers.c ×

home > javad > Desktop > OSLab > project6 > C Dining_Philosophers.c > ...

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define n 5 // number of philosophers
int phil_number[n] = { 0, 1, 2, 3, 4 };

// state of philosophers
enum state_phil{eating, hungry, thinking};
int phil_state[n];

// mutex is used such that no two philosophers
// may access the pickup or putdown at the same time
sem_t mutex;
sem_t phil_sem[n];
```

```c
// eat food by philosopher phil_num
void eat(int phil_num){

    int left_phil_num = (phil_num + 4) % n;
    int right_phil_num = (phil_num + 1) % n;

    if (
        phil_state[phil_num] == hungry
        &&
        phil_state[left_phil_num] != eating
        &&
        phil_state[right_phil_num] != eating
    ){
        // start eating by philosopher phil_num
        phil_state[phil_num] = eating;

        printf("philosopher %d is eating using chopstick[%d] and chopstick[%d]\n",
        phil_num + 1, left_phil_num + 1, phil_num + 1);

        // in put_chopsticks used for alarm hungry philosophers
        sem_post(&phil_sem[phil_num]);
    }
}
```

```
41  // take chopsticks by philosopher phil_num
42  void take_chopsticks(int phil_num){
43
44      // begining of critical section
45      sem_wait(&mutex);
46
47      // when he want to take chopsticks means that he is hungary
48      phil_state[phil_num] = hungry;
49      // he can eat if his neighbours are not eating
50      eat(phil_num);
51
52      // end of critical section
53      sem_post(&mutex);
54
55      // if unable to eat wait to be signalled
56      sem_wait(&phil_sem[phil_num]);
57  }
```

```
59  // put chopsticks by philosopher phil_num
60  void put_chopsticks(int phil_num){
61
62      // begining of critical section
63      sem_wait(&mutex);
64
65      // after putting chopsticks he is not
66      // hungary or eating but is thinking
67      phil_state[phil_num] = thinking;
68
69      printf("philosopher %d finished eating\n", phil_num + 1);
70      // printf("Philosopher %d is thinking\n", phil_num + 1);
71
72      int left_phil_num = (phil_num + 4) % n;
73      int right_phil_num = (phil_num + 1) % n;
74
75      eat(left_phil_num);
76      eat(right_phil_num);
77
78      // end of critical section
79      sem_post(&mutex);
80  }
```

```
82  void* phil_thread_handler(void* phil_num){
83      // while (1) {
84          int* i = phil_num;
85          take_chopsticks(*i);
86          put_chopsticks(*i);
87      // }
88  }
89
90  int main(){
91      int i;
92      pthread_t phil_thread[n];
93
94
95      // initialize the semaphores
96      sem_init(&mutex, 0, 1);
97
98      for(i = 0; i < n; i++)
99          sem_init(&phil_sem[i], 0, 0);
100
101     // create philosopher processes
102     for (i = 0; i < n; i++) {
103         pthread_create(&phil_thread[i], NULL, phil_thread_handler, &phil_number[i]);
104         printf("philosopher %d is thinking !!\n", i + 1);
105     }
```

```
107     for (i = 0; i < n; i++)
108         pthread_join(phil_thread[i], NULL);
109 }
```

در این مسئله 5 فیلسوف در دور یک میز هستند و قرار است که با 5 چوبی که روی میز است غذای وسط میز را بخورند. هر فیلسوف در کنار خود 2 فیلسوف چپ و راست را دارد و همچنین یک چوب چپ و یک چوب راست هم وجود دارد و برای آنکه بتواند بخورد باید هر دو چوب کنار خود را در اختیار داشته باشد. در این مسئله از ترد یا ریسمان برای هر یک از فیلسوف ها استفاده شده است و یک عدد سمافور برای غذای وسط میز و 5 عدد هم سمافور برای هر یک از فیلسوف ها در نظر گرفته شده است. هر فیلسوف می تواند در یکی از سه حالت hungry , thinking, eating باشد. در هندلر مربوط به هر یک از ترد ها 2 کار انجام میشود. ابتدا چوب ها را بر میدارد (که این عملیات به دلیل اینکه چوب ها با همسایگان مشترک است نیاز به سمافور دارد که بتواند هر دو چوب کناری خود را در صورتی که آزاد هستند در اختیار بگیرد) و سپس شروع به خوردن می کند (که نیاز است که دو همسایه او در حال خوردن نباشند و چوب ها آزاد باشند) و سپس بعد از اینکه خوردن غذا این فیلسوف تمام شود چوب ها را روی میز میگذارد و همچنین به دو همسایه خود نیز اطلاع می دهد که چوب ها را گذاشته و آنها می توانند استفاده کنند (تعارف میزنه بهشون 😊) و این عملیات می تواند تا تعداد بیشمار یا تعداد معینی ادامه یابد.

خروجی:

```
javad@javad-HP-350-G1:~/Desktop/OSLab/project6$ gcc -pthread -o Dining_Philosophers Dining_Philosophers.c
javad@javad-HP-350-G1:~/Desktop/OSLab/project6$ ./Dining_Philosophers
philosopher 1 is thinking !!
philosopher 2 is thinking !!
philosopher 3 is thinking !!
philosopher 4 is thinking !!
philosopher 5 is thinking !!
philosopher 5 is eating using chopstick[4] and chopstick[5]
philosopher 5 finished eating
philosopher 4 is eating using chopstick[3] and chopstick[4]
philosopher 1 is eating using chopstick[5] and chopstick[1]
philosopher 1 finished eating
philosopher 2 is eating using chopstick[1] and chopstick[2]
philosopher 2 finished eating
philosopher 4 finished eating
philosopher 3 is eating using chopstick[2] and chopstick[3]
philosopher 3 finished eating
javad@javad-HP-350-G1:~/Desktop/OSLab/project6$
```

همگی فیلسوف ها زمانی که سر میز می آیند ابتدا در حالت thinking هستند. سپس یکی از آنها شروع به خوردن می کند و وقتی کارش پایان یافت اطلاع finished را به همسایه ها می دهد و ...

در این جا با این شرط کار انجام شده است که هر یک از افراد فقط یک مرتبه غذا بخورند اما میتوان بخش while در تابع phil_thread_handler را فعال کرد تا به صورت بی نهایت کار ادامه پیدا کند.