گزارش ازمایش ۶

همگام سازی

نام نام خانوادگی: روژینا کاشفی نام استاد: سرکار خانم علیزاده تاریخ:۱۴۰۰/۹/۱۷

بخش اول)

چون فضای مشترکی بین نویسنده و خواننده ها داریم، ممکن است ناسازگاری داده رخ دهد. همچنین متغیری که تعداد خواننده ها را ذخیره میکند، ممکن است توسط چندین خواننده بصورت همزمان دستکاری شود.

به همین دلیل کد را بصورت زید اجرا می کنیم. بین نویسنده و خواننده ها یک فضای اشتراکی داریم به نام count که به وسیله سمافور از آن محافظت می کنیم تا که در هر لحظه که نویسسنده به آن دسترسی دارد، خواننده ای نتواند از آن چیزی بخواند و زمانی که یک یا چند خواننده در حال خواندن از آن هستند، نویسنده نتواند روی آن چیزی بنویسد و محتوا را تغییر دهد. متغیر cnt_reader هم نشان دهنده تعداد خواننده هایی است که درحال خواندن از حافظه مشترک هستند. چون همه خواننده ها به این متغیر دسترسی دارند، آن را با استفاده از mutex محافظت کرده ای تا در هرلحظه از زمان فقط یک خواننده بتواند مقدار آن را تغییر دهد.

وقتی اولین reader به بافر دسترسی میابد باید آن را lock کند و وقتی آخرین reader کارش تمام شد lock را رها میکند . فرآیند writer زمانی می تواند مقداري بنویسد که فرآیند reader به بافر دسترسی نداشته باشد و تا اتمام عملیات نوشتن، فرآیند readerقادر به خواندن نیست.

reader_writer.c

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

int *count = 0;

sem_t write_lock;

int reader_cnt = 0;

pthread_mutex_t mutex;


int shmid;

void *reader(void *r) {

    key_t key = ftok("shm", 1);
    shmid = shmget(key, 10 * sizeof(int), 0444 | IPC_CREAT);
    count = (int *)shmat(shmid, NULL, 0);

    pthread_mutex_lock(&mutex);

    reader_cnt++;

    if(reader_cnt == 1) {
        sem_wait(&write_lock);
    }

    pthread_mutex_unlock(&mutex);

    printf("Reader %d read count ->  %d, pid is -> %ld\n",*((int *)r),*count, (long)getpid());

    pthread_mutex_lock(&mutex);
```

```
42
43      reader_cnt--;
44
45      if(reader_cnt == 0) {
46          sem_post(&write_lock);
47      }
48      pthread_mutex_unlock(&mutex);
49  }
50
51
52  void *writer(void *w) {
53      *count = 0;
54
55      key_t key = ftok("shm", 1);
56      shmid = shmget(key, 10 * sizeof(int), 0666 | IPC_CREAT);
57      count = (int *)shmat(shmid, NULL, 0);
58      sem_wait(&write_lock);
59
60      while (1){
61              *count = *count + 1;
62              printf("Writer changed count to -> %d, pid is -> %ld\n",*count,(long)getpid());
63              if(*count >= 5)
64                  break;
65          }
66
67      sem_post(&write_lock);
68
69  }
70
71  int main() {
72      key_t key = ftok("shm", 1);
73      shmid = shmget(key, 10 * sizeof(int), 0666 | IPC_CREAT);
74      count = (int *)shmat(shmid, NULL, 0);
75
76      pthread_t wr,rd1, rd2;
77
78      pthread_mutex_init(&mutex, NULL);
79      sem_init(&write_lock,0,1);
80
81      int num1 = 1;
82      int num2 = 2;
83
84      //create reader threads
85      pthread_create(&rd1, NULL, (void *)reader, (void *)&num1);
86      pthread_create(&rd2, NULL, (void *)reader, (void *)&num2);
87
88      //create writer thread
89      pthread_create(&wr, NULL, (void *)writer, (void *)&num1);
90
91      //join reader threads
92      pthread_join(rd1, NULL);
93      pthread_join(rd2, NULL);
94
95      //join writer thread
96      pthread_join(wr, NULL);
97
98      pthread_mutex_destroy(&mutex);
99      sem_destroy(&write_lock);
00
01      shmdt(count);
02      return 0;
03  }
```

خروجی زیر مشاهده میکنیم بعد از اتمام ۲ reader ۵ writer کاره خود را شروع کردند.

```
rojina@ubuntu:~/Desktop/OsLab/Lab6$ gcc -pthread readWrite.c -o rw
rojina@ubuntu:~/Desktop/OsLab/Lab6$ ./rw
Reader 1 read count ->  0, pid is -> 3443
Reader 2 read count ->  0, pid is -> 3443
Writer changed count to -> 1, pid is -> 3443
Writer changed count to -> 2, pid is -> 3443
Writer changed count to -> 3, pid is -> 3443
Writer changed count to -> 4, pid is -> 3443
Writer changed count to -> 5, pid is -> 3443
```

بخش دوم)

در این بخش یک mutex برای چوب های غذا تعریف می کنیم تا در هرلحظه فقط یک فیلسوف بتواند از آن استفاده کند. هر فیلسوف را هم بصورت یک ریسمان تعریف کرده ایم که یا فکر میکند و یا غذا می خورد. هر فیلسفو اول چوب سمت راستش را برمیدار و بعد چوب سمت چپش را برمیدارد. در اینجا ممکن است بن بست رخ دهد. اگر همه فیلسوف ها چوب سمت راست خوذشان را برداشته باشند، همگی منتظر هستند تا فیلسوف سمت چپشان چوبش را روی میز بگذارد تا بردارد و غذا بخورد و چون همگی منتظر هم هستند، همه به بن بست می خورند.

برای حل این مشکل، کد را به گونه ای نوشته ایم که هرکس زمانی که چوب سمت راستش را برمیدارد، اگر بتواند چوب سمت چپش را بردارد که غذا خوردن را شروع می کند. درغیراینصورت، چوبی که برداشته است را روی میز میگذارد و دوباره اقدام به برداشتن چوب ها می کند. همچنین اگر چوب راست را نتوانست بردارد ولی چوپ سمت راست را برداشت، آن را هم میگذارد روی میز تا دفعه بعد که دوباره تالش کند. دراین حالت دیگر بن بست نخواهیم داشت.

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

#define CHOPSTICKS_NUM 5
#define PHILSOOPH_NUM 5

pthread_mutex_t chopsticks[CHOPSTICKS_NUM];


void *philosoph_handler (void* args) {
  int id = *((int*)args);
  int next_id = 0;
  if (id < 4) {
    next_id = id + 1;
  }

  int possible1;
  int possible2;

  printf("Philosopher[%d] is Thinking... \n", id);
  sleep(rand() % 10);

  while(1){

    possible1 = pthread_mutex_trylock(&chopsticks[id]);
    possible2 = pthread_mutex_trylock(&chopsticks[next_id]);

    if (possible1 == 0){
        if (possible2 == 0){
                printf("Philosopher[%d] is Eating by chopstick[%d] and chopstick[%d]...\n", id, id, next_id);
                        sleep(rand() % 10);
                        printf("Philosopher[%d] finished eating!\n", id);
                        pthread_mutex_unlock(&chopsticks[id]);
                        pthread_mutex_unlock(&chopsticks[next_id]);
                        break;
                }
        else {
                pthread_mutex_unlock(&chopsticks[id]);//because there aren't two chopsticks we put locked chopstick on
                                                      //table by unlocking related mutex object
                }
                }
        if (possible2 == 0) {
                pthread_mutex_unlock(&chopsticks[next_id]);//because there aren't two chopsticks we put locked chopstick
                                                           // on table by unlocking related mutex object
                }
        }
}

void main() {
  pthread_t philosophs[PHILSOOPH_NUM];
  int ids[5];
  for (int i = 0; i < CHOPSTICKS_NUM; i++) {
    ids[i] = i;
    //make all chopsticks a mutex object, null make a mutex by default attributes
    pthread_mutex_init(&chopsticks[i], NULL);
  }

  for (int i = 0; i < PHILSOOPH_NUM; i++) {
    // run a thread for each philsooph by philsoph_handler and get ids as args of this function
    pthread_create(&philosophs[i], NULL, philosoph_handler, &ids[i]);
  }
  //wait for all philsooph(thread) to finish their job
  for (int i = 0; i < PHILSOOPH_NUM; i++) {
    pthread_join(philosophs[i], NULL);
  }

}
```

خروجی مطابق زیر است.

```
rojina@ubuntu:~/Desktop/OsLab/Lab6$ gcc -pthread philosophers.c -o ph
rojina@ubuntu:~/Desktop/OsLab/Lab6$ ./ph
Philosopher[0] is Thinking...
Philosopher[2] is Thinking...
Philosopher[1] is Thinking...
Philosopher[3] is Thinking...
Philosopher[4] is Thinking...
Philosopher[0] is Eating by chopstick[0] and chopstick[1]...
Philosopher[3] is Eating by chopstick[3] and chopstick[4]...
Philosopher[0] finished eating!
Philosopher[1] is Eating by chopstick[1] and chopstick[2]...
Philosopher[1] finished eating!
Philosopher[3] finished eating!
Philosopher[2] is Eating by chopstick[2] and chopstick[3]...
Philosopher[4] is Eating by chopstick[4] and chopstick[0]...
Philosopher[4] finished eating!
Philosopher[2] finished eating!
```