آزمایش 6

فاطمه راق 40013045

حامد فلاحی 9931043

خلاصه

ایجاد هماهنگی بین چند فرآیند در حال اجرا در این گزارش کار مد نظر ماست. به طور کلی، زمانی که فرآیندها به صورت همزمان اجرا میشوند و منابع بین آنها مشترک است احتمال بروز شروط مسابقه وجود دارد که در آن برنامه الزاماً در هر بار اجرا، پاسخ یکسانی تولید نخواهد کرد. برای جلو گیری از این مساله، نیاز به همگامسازی است. در این آزمایش هدف بررسی بیشتر این مساله است.
همچنین در بخش دوم این گزارش کار، به مبحث deadlock اشاره میشود و از دید کلی این مبحث را بررسی کرده و مشکلاتی که ممکن است بوجود بیاید را پیدا میکند و آن را معیار قرار میدهد تا مشکل را برطرف کند.

(HW1

در بخش 1، از ما خواسته شده که محیطی آماده کنیم که دو فرآیند در آن وجود داشته باشند و یکی از فرآیند ها هم بنویسد و هم بخواند. و فرایند دیگری، فقط بتواند عملیات خواندن را انجام دهد. باتوجه به اینکه در این فرایند ها ما کد مورد نظر را طبق عکس های زیر و با روش ارتباط بین فرایند های shared memory پیاده سازی کردیم، میدانیم که درصورت نبود حالت سمافور، به هنگام اجرای این کد باعث میشود بین عملیات خواندن و نوشتن ترتیب مشخصی وجود نداشته باشد و هر فرایندی که به منبع زودتر دسترسی پیدا کند، منبع را طبق خواسته خود تغییر دهد(البته چون که ما فقط یک فرایند نوشتن داریم، پس ممکن است نوشتن در زمانی قرار بگیرد که عملیات خواندن، مقدار های متفاوتی و بدون اولویتی از منبع بخواند.) به همین دلیل خروجی خواندن ها هیچوقت برای ما ثابت نیستند و در نتیجه برای خواندن و نوشتن منابع و ترتیب انجام عملیات و لغو کردن شرط مسابقه و همچنین گذاشتن شرط ورود فقط یک فرایند مشخص به ناحیه بحرانی، ما باید با استفاده ا روش سمافور این مشکل را حل کنیم و به هنگام ورود فرایند ها به ناحیه بخرانی، یکی از فرایند هارا متوقف کنیم تا زمانی که فرایندی که داخل ناحیه بحرانی است، تمام شود از از این ناحیه خارج شود. به همین دلیل از روش سمافور استفاده کرده و این مشکل را حل میکنیم:

+(جواب سوال مورد نظر در متن بالا داده شده است)

RW.c - Notepad

File Edit View

```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>

#define MAX_COUNT 15

sem_t reader_writer;
sem_t mutex;

int numberReader = 0;
int *cnt = 0;


void *writer(void *wno)
{
    *cnt = 0;
    int shmid = shmget(0x125, 10*sizeof(int), 0666 | IPC_CREAT);
    cnt = (int *)shmat(shmid, NULL, 0);
    sem_wait(&reader_writer);
    for (int i = 1; i <= MAX_COUNT; i++)
        {

            *cnt=*cnt+1;
            printf("Writer %d write cnt --> %d\n",(*((int *)wno)),*cnt);
        }
    sem_post(&reader_writer);
```

RW.c - Notepad

File Edit View

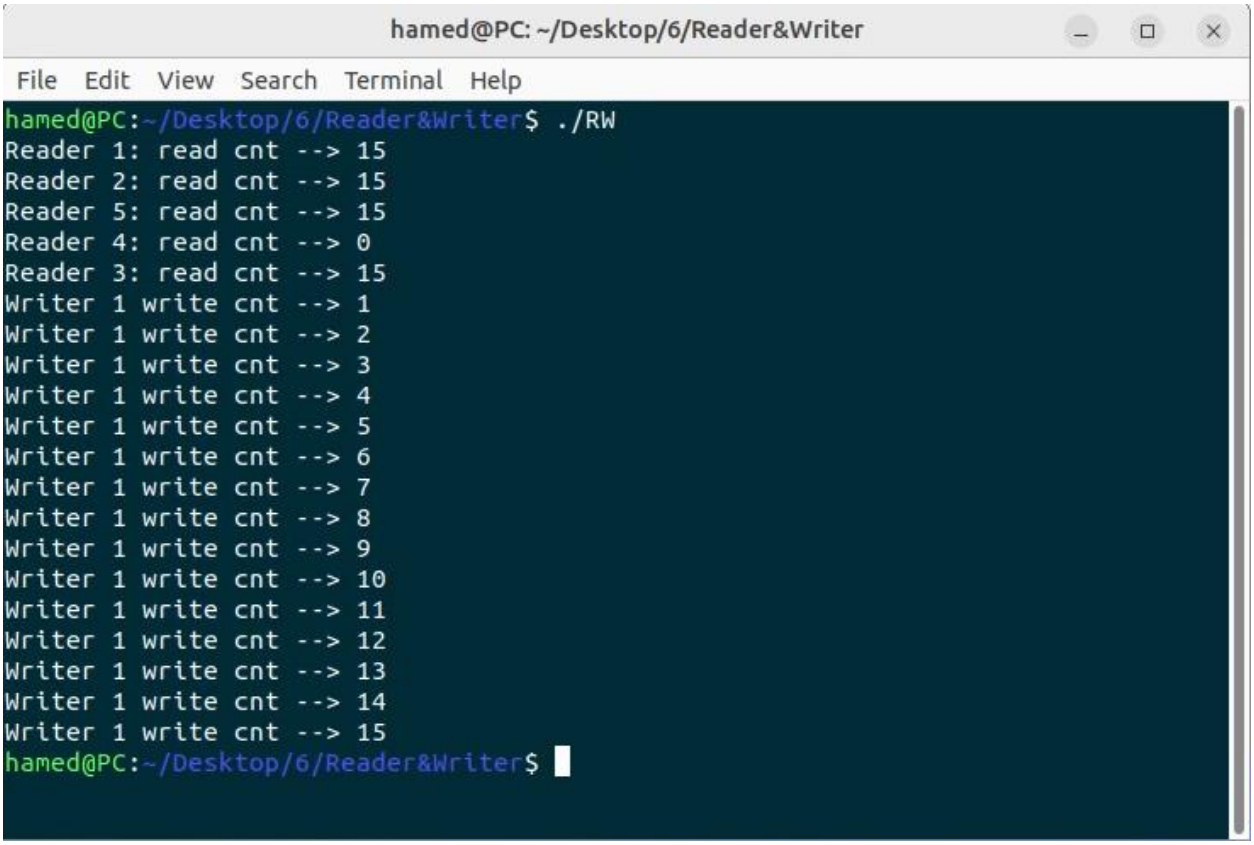
```
        }
    sem_post(&reader_writer);

}

void *reader(void *rno)
{

    int shmid = shmget(0x125, 10*sizeof(int), 0444 | IPC_CREAT);
    cnt = (int *)shmat(shmid, NULL, 0);
    sem_wait(&mutex);
    numberReader++;
    if(numberReader == 1) {
        sem_wait(&reader_writer);
    }
    sem_post(&mutex);
    printf("Reader %d: read cnt --> %d\n",*((int *)rno),*cnt);

    sem_wait(&mutex);
    numberReader--;
    if(numberReader == 0) {
        sem_post(&reader_writer);
    }
    sem_post(&mutex);
}

int main()
{
    int shmid = shmget(0x125, 10 * sizeof(int), 0666 | IPC_CREAT);
    cnt = (int *)shmat(shmid, NULL, 0);
    pthread_t read[10],write[5];
    sem_init(&mutex,0,1);
    sem_init(&reader_writer,0,1);
```

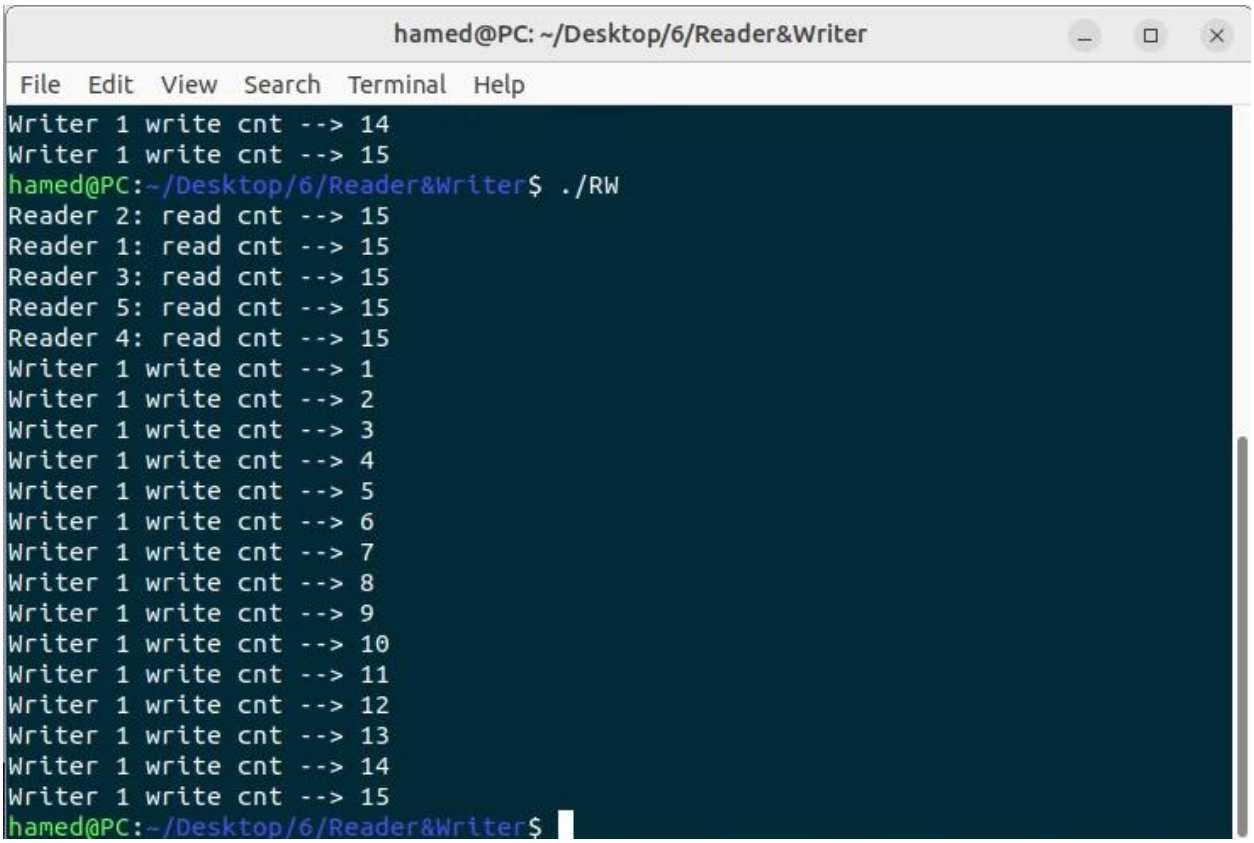
RW.c - Notepad

File Edit View

```
    sem_post(&mutex);
}

int main()
{
    int shmid = shmget(0x125, 10 * sizeof(int), 0666 | IPC_CREAT);
    cnt = (int *)shmat(shmid, NULL, 0);
    pthread_t read[10],write[5];
    sem_init(&mutex,0,1);
    sem_init(&reader_writer,0,1);

    int a[10] = {1,2,3,4,5};

    for(int i = 0; i < 5; i++) {
        pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
    }
    for(int i = 0; i < 1; i++) {
        pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
    }

    for(int i = 0; i < 5; i++) {
        pthread_join(read[i], NULL);
    }
    for(int i = 0; i < 1; i++) {
        pthread_join(write[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&reader_writer);

    return 0;

}
```

در عکس های اجرای کد زیر میبینیم که نتیجه با هربار چاپ شدن، اولویت را ابتدا به فرایند خواندن، سپس پس از اتمام خواندن ها، اولویت را به نوشتن میدهد:


hamed@PC: ~/Desktop/6/Reader&Writer
File Edit View Search Terminal Help
hamed@PC:~/Desktop/6/Reader&Writer$ ./RW
Reader 1: read cnt --> 15
Reader 2: read cnt --> 15
Reader 5: read cnt --> 15
Reader 4: read cnt --> 0
Reader 3: read cnt --> 15
Writer 1 write cnt --> 1
Writer 1 write cnt --> 2
Writer 1 write cnt --> 3
Writer 1 write cnt --> 4
Writer 1 write cnt --> 5
Writer 1 write cnt --> 6
Writer 1 write cnt --> 7
Writer 1 write cnt --> 8
Writer 1 write cnt --> 9
Writer 1 write cnt --> 10
Writer 1 write cnt --> 11
Writer 1 write cnt --> 12
Writer 1 write cnt --> 13
Writer 1 write cnt --> 14
Writer 1 write cnt --> 15
hamed@PC:~/Desktop/6/Reader&Writer$


hamed@PC: ~/Desktop/6/Reader&Writer
File Edit View Search Terminal Help
Writer 1 write cnt --> 14
Writer 1 write cnt --> 15
hamed@PC:~/Desktop/6/Reader&Writer$ ./RW
Reader 2: read cnt --> 15
Reader 1: read cnt --> 15
Reader 3: read cnt --> 15
Reader 5: read cnt --> 15
Reader 4: read cnt --> 15
Writer 1 write cnt --> 1
Writer 1 write cnt --> 2
Writer 1 write cnt --> 3
Writer 1 write cnt --> 4
Writer 1 write cnt --> 5
Writer 1 write cnt --> 6
Writer 1 write cnt --> 7
Writer 1 write cnt --> 8
Writer 1 write cnt --> 9
Writer 1 write cnt --> 10
Writer 1 write cnt --> 11
Writer 1 write cnt --> 12
Writer 1 write cnt --> 13
Writer 1 write cnt --> 14
Writer 1 write cnt --> 15
hamed@PC:~/Desktop/6/Reader&Writer$

(HW2

بخش 2 نیز از ما خواستار است که محیطی طراحی کنیم شامل تعدادی فرایند (معادل فیلسوف ها) و تعدادی منبع به اندازه تعداد فرایند ها (چوب های غذاخوری) باشد. هر فیلسوف مدتی تفکر میکند و وقتی گرسنه شد دو چوب غذا خوری لازم دارد تا از غذای وسط میز بخورد. فیلسوف در یک زمان مشخص می تواند فقط یک چوب بردارد و اگر چوب در اختیار فیلسوف کناری باشد، قاعدتاً نمیتواند آن را بردارد. پس از خوردن غذا چوبها را روی میز میگذارد تا بقیه در صورت نیاز از آنها استفاده کنند.
مشکل این عملکرد این است که درصورت اینکه فلیسوف ها هرکدام یک چوب را بردارند، چون در دست هر فلیسوف فقط یک چوب هست، پس نمیتواند هیچ فلیسوفی غذای خود را میل کند. چون همه فلیسوف ها نیاز به 2 چوب دارند و در این شرایط، مشکل deadlock رخ میدهد. برای حل این مشکل ما در الگوریتم کد از روشی استفاده کردیم که هرگاه فلیسوف ها چوب های ابتدایی خود را برداشتند و پس از گذشت زمان و برای برداشتن چوب بعدی، اگر ان چون دست فرایند دیگر بود، این فرایند چوب خود را آزاد کند تا فرایند دیگر به مشکل بن بست نخورد و تمام فرایند ها به انتها برسند. چون برای تمام شدن تمام فرایند ها نیاز است حداقل 1 فرایند یا فیلسوف از 2 چوب استفاده کند و پس از اتمام، منابع خود را آزاد کند تا به بقیه نیز چوب غذاخوری برسد و مشکل بنبست به طور کلی برطرف شود.
الگوریتم کد به شکل زیر است و باتوجه به خروجی ای که میبینیم، میفهمیم که در نهایت همه فیلسوف ها غذای خودشان را تمام میکنند و محیط به پایان میرسد.

Philsoophers.c - Notepad

File Edit View

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

#define PHILSOOPH_NUM 5

pthread_mutex_t chopsticks[PHILSOOPH_NUM];


void *philosoph_handler (void* philId)
 {
  int right = *((int*)philId);
  int left = (right+1)% PHILSOOPH_NUM ;
  int rightForklock;
  int leftForklock;

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

  while(1){

    rightForklock = pthread_mutex_trylock(&chopsticks[right]);
    leftForklock = pthread_mutex_trylock(&chopsticks[left]);

    if (rightForklock == 0){
      if (leftForklock == 0){
            printf("Philosopher[%d] is Eating by chopstick[%d] and chopstick[%d]...\n", right, right, left);
                  sleep(0.3);
                  printf("Philosopher[%d] finished eating!\n", right);
                  pthread_mutex_unlock(&chopsticks[right]);
                  pthread_mutex_unlock(&chopsticks[left]);
                  break;
```

Philsoophers.c - Notepad

File Edit View

```
                sleep(0.3);
                printf("Philosopher[%d] finished eating!\n", right);
                pthread_mutex_unlock(&chopsticks[right]);
                pthread_mutex_unlock(&chopsticks[left]);
                break;
            }
        else {
            pthread_mutex_unlock(&chopsticks[right]);
          }
      }
    if (leftForklock == 0) {
            pthread_mutex_unlock(&chopsticks[left]);
      }
  }
}

void main() {
  pthread_t philosophs[PHILSOOPH_NUM];
  int ids[5];
  for (int i = 0; i < PHILSOOPH_NUM; i++) {
    ids[i] = i;
    pthread_mutex_init(&chopsticks[i], NULL);
  }

  for (int i = 0; i < PHILSOOPH_NUM; i++) {
    pthread_create(&philosophs[i], NULL, philosoph_handler, &ids[i]);
  }
  for (int i = 0; i < PHILSOOPH_NUM; i++) {
    pthread_join(philosophs[i], NULL);
  }

}
```

hamed@PC: ~/Desktop/6/diningPhilosophers

File Edit View Search Terminal Help

```
hamed@PC:~/Desktop/6/diningPhilosophers$ ./Philsoophers
Philosopher[0] is Thinking...
Philosopher[1] is Thinking...
Philosopher[2] is Thinking...
Philosopher[3] is Thinking...
Philosopher[4] is Thinking...
Philosopher[0] is Eating by chopstick[0] and chopstick[1]...
Philosopher[0] finished eating!
Philosopher[4] is Eating by chopstick[4] and chopstick[0]...
Philosopher[4] finished eating!
Philosopher[3] is Eating by chopstick[3] and chopstick[4]...
Philosopher[3] finished eating!
Philosopher[1] is Eating by chopstick[1] and chopstick[2]...
Philosopher[1] finished eating!
Philosopher[2] is Eating by chopstick[2] and chopstick[3]...
Philosopher[2] finished eating!
hamed@PC:~/Desktop/6/diningPhilosophers$
```