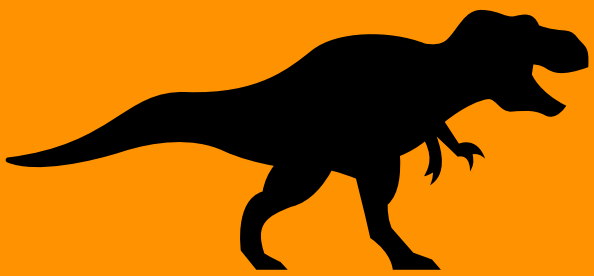
SYNCHRONIZATION

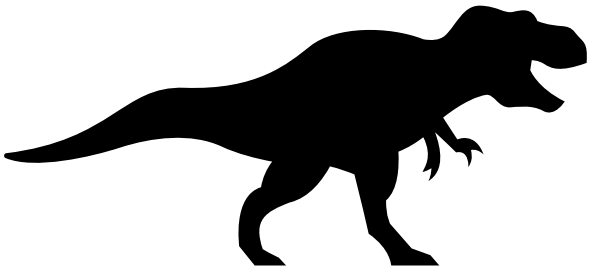
OPERATING SYSTEM



COLLABORATORS NAME & ID :
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071

SYNCHRONIZATION

OPERATING SYSTEM



COLLABORATORS NAME & ID :
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071

OPERATING SYSTEM LABORATORY

SYNCHRONIZATION

INSTRUCTOR :

ARMAN GHEISARI

COLLABORATORS :

MARYAM ALIKARAMI

MAHAN AHMADVAND

QUESTION 1

مساله ی خوانندگان-نوسیندگان را پیاده سازی کنید.
برنامه ی مربوطه را بصورت کامل نوشته و سپس اجرا کنید.
آیا مشکلی وجود دارد؟ در صورت وجود ناهماهنگی چه راهکاری ارائه می کنید؟

مساله ی خوانندگان و نویسندگان را پیاده سازی کرده ایم، مشکلی که مشاهده کردیم آن بود که ممکن است فرآیند Reader مقداری را بخواند که آخرین مقدار Writer نباشد و در هر بار اجرای برنامه شرایط یکسانی ایجاد نشود و برنامه دچار شریط race شود، برای جلوگیری از race condition باید ابتدا مساله ی critical section را حل کنیم.

برای حل این مساله از راه حل سمافور استفاده کرده ایم که سه شرط Mutual exclusion، Progress و Bounded waiting را برآورده می کند.

در واقع از POSIX Semaphore برای حل این مشکل استفاده کرده ایم.

پیاده سازی آن مطابق شکل زیر است :

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



typedef struct {
    int count, rc;
    sem_t mutex;
    sem_t data;
} SharedData;



int root_pid;
int id;

void writer();
void reader();

int main()
{

    SharedData* sharedData;
    int pid;


    id = shmget(IPC_PRIVATE, sizeof(SharedData), IPC_CREAT | 0666);


    sharedData = (SharedData *)shmat(id, NULL, 0);



    root_pid = getpid();

    sem_init(&(sharedData->mutex), 1, 1);
    sem_init(&(sharedData->data), 1, 1);
    sharedData->count = 0;
    sharedData->rc = 0;
    pid = fork();
    if (pid == 0) { //writer process
        writer();
        return 0;
    }

    for (int i = 0; i < 5; i++)
    {
        if (getpid() == root_pid)
            pid = fork();
        else
            break;
```

```c
    if (pid == 0) { //reader process
        reader();
        return 0;
    }


    if (getpid() == root_pid) // parent process
    {
        wait(NULL); // wait on writer
        for (int i = 0; i < 5; i++) // wait on readers
        {
            wait(NULL);
        }
    }


    return 0;
}

void reader() {
    SharedData* sharedData;
    sharedData = (SharedData *)shmat(id, NULL, 0);

    int pid = getpid();
    bool max = 0;
    while(!max){
        sem_wait((&sharedData->mutex));
        sharedData->rc = sharedData->rc + 1;
        if(sharedData->rc == 1) {
        sem_wait((&sharedData->data));
        }
        sem_post((&sharedData->mutex));
        printf("Reader:\tPID: %d\tcount: %d\n", pid, sharedData->count);
        if(sharedData->count >= 5){
            max = 1;
        }
        sem_wait((&sharedData->mutex));
        sharedData->rc = sharedData->rc - 1;
        if(sharedData->rc == 0) {
        sem_post((&sharedData->data));
        }
        sem_post((&sharedData->mutex));

    }

}

void writer() {

    SharedData* sharedData;
    sharedData = (SharedData *)shmat(id, NULL, 0);

    int pid = getpid();
    bool max = 0;
    while(!max){
        sem_wait((&sharedData->data));
        sharedData->count++;
        if(sharedData->count >= 5){
            max = 1;
        }
```

```
122             printf("Writer:\tPID: %d\tcount: %d\n", pid, sharedData->count);
123             sem_post((&sharedData->data));
124         }
125     }
```

همانطور که می بینیم برای استفاده از سمافور ابتدا کتابخانه ی semaphore.c را include کرده ایم.

ما در این پیاده سازی از دو سمافور mutex و data استفاده کرده ایم.

سمافور mutex را برای همگام سازی متغیر rc یا reader counter استفاده کرده ایم(توجه شود این متغیر نیز ممکن است به دلیل وجود داشتن در critical section دچار شرایط مسابقه شود در نتیجه نیاز به یک متغیر سمافور برای این قسمت احساس می شود).

سمافور data نیز برای همگام سازی reader و writer استفاده شده است تا در تغییر متغیر count دچار شرایط مسابقه نشویم.

حال با استفاده از shared memory، حافظه مشترک با متغیر های rc و count و دو سمافور mutex و data ایجاد کرده ایم.

سپس با استفاده از دستور fork فرزند هایی برای پردازه پدر(root) ایجاد کرده ایم(یک پردازه برای writer و پنج پردازه برای reader ها).

حال می خواهیم یک سناریو را بررسی کنیم :

ابتدا خواننده برروی سمافور mutex یک wait می زند و مقدار آن را صفر می کند زیرا می خواهیم که متغیر rc دچار شرایط مسابقه نشود، حال اگر اولین خواننده باشیم برروی سمافور data یک wait می زنیم و مقدار آن را صفر می کنیم،

تا نویسنده نتواند در حال خواندن مقداری را برروی حافظه ی مشترک بنویسد، سپس روی سمافور mutex، post می زنیم و این باعث می شود که از حالت بلاکینگ خارج شود زیرا می خواهیم خوانندگان دیگر نیز قابلیت خواندن را داشته باشند،

هنگامی که خواندن تمام شد باید یکبار دیگر روی متغیر mutex، wait کنیم و مقدار rc را کاهش دهیم، حال اگر آخرین پردازه ی read باشیم باید متغیر data را نیز آزاد کنیم و در واقع writer را از حالت blocking خارج کنیم، برای این کار باید روی سمافور data، post بزنیم تا از حالت بلاکینگ خارج شود و در انتها هر کدام از reader ها که مقدار rc را کم کردند باید روی سمافور mutex، post بزنند، تا دیگر reader ها بتوانند متغیر rc را تغییر بدهند.

حال کد مربوطه را با استفاده از gcc -pthread -o rw rw.c کامپایل کرده ایم و خروجی زیر را گرفته ایم که نشان

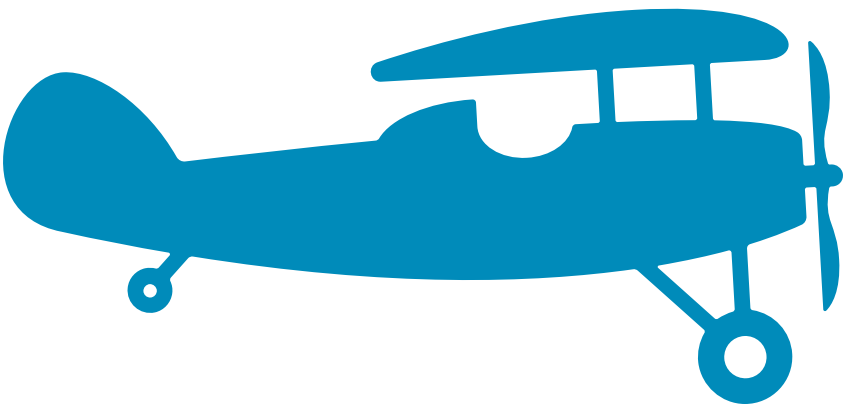
دهنده ی آن است که دیگر race condition نداریم :

توجه شود که می توانستیم از راه حل های دیگری نیاز مساله ی بحرانی یا critical section را حل کنیم، اما باید توجه کنیم که راه حل پیشنهادی باید سه شرط Mutual exclusion، Progress و Bounded waiting را برآورده می کند.

Mutual exclusion: When a thread is executing in its critical section, no other threads can be executing in their critical sections.

Progress: If no thread is executing in its critical section, and if there are some threads that wish to enter their critical sections, then one of these threads will get into the critical section.

Bounded waiting: After a thread makes a request to enter its critical section, there is a bound on the number of times that other threads are allowed to enter their critical sections, before the request is granted




THE END