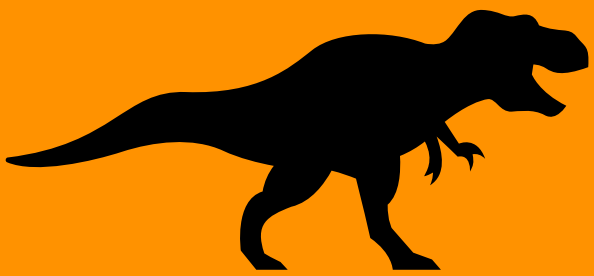
SYNCHRONIZATION

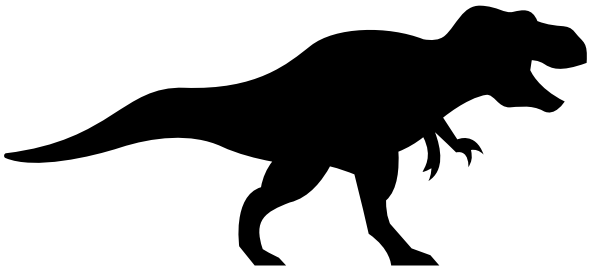
OPERATING SYSTEM



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

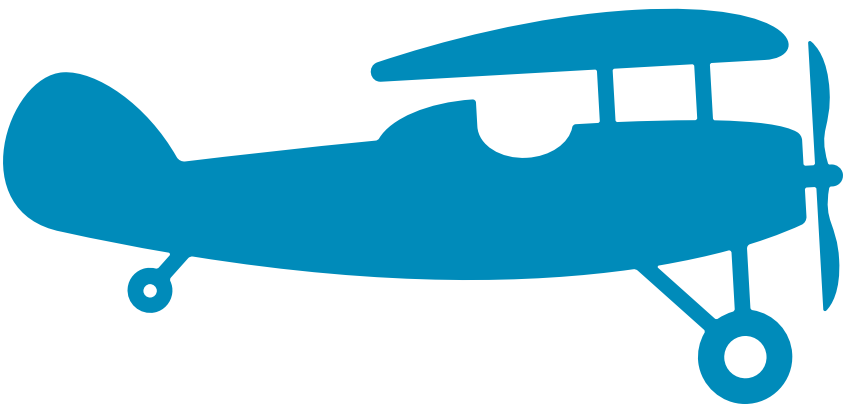
دهنده ی آن است که دیگر race condition نداریم :

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



QUESTION 2

قسمت الف) آیا ممکن است بن‌بست رخ دهد؟ در صورت امکان چگونگی ایجاد آن را توضیح دهید.

بله در این مساله ممکن است که بن‌بست یا Deadlock اتفاق بیفتد. بن‌بست حالتی است که در آن هیچ کدام از پردازه‌ها یا نخ‌ها اجرا نمی‌شوند. فرض کنید شرایطی باشد که همه‌ی فیلسوف‌ها به یک باره گشنه شوند و دست از فکر کردن بکشند، در این حالت اگر چنگال‌ها به صورت درستی بین آن‌ها تقسیم شود، در آن واحد حداقل یک فیلسوف و ماکسیمم دو فیلسوف باید قادر به غذا خوردن باشند اما حالتی از تقسیم چنگال‌ها وجود دارد که در آن هیچ فیلسوفی نتواند غذا بخورد.

فرض کنید که کد شبیه‌ساز فیلسوف‌ها، شامل مراحل زیر است:

- فکر کردن

- برداشتن چنگال سمت راست

- برداشتن چنگال سمت چپ

- غذا خوردن

- گذاشتن چنگال سمت چپ

- گذاشتن چنگال سمت راست

ما در شرایطی هستیم که همه‌ی فیلسوف‌ها از مرحله‌ی فکر کردن خارج شده‌اند، حالا اگر همگی به صورت همزمان اقدام به برداشتن چنگال سمت راست خود کنند (پردازنده‌های چند هسته‌ای) یا به صورت سریالی و پشت

سر هم، ابتدا همگی چنگال‌های سمت راست را بردارند و بعد به سراغ چنگال‌های سمت چپ بروند (پردازنده‌های تک‌هسته‌ای با امکان ایجاد هم‌روندی)، در این حالت همه‌ی فیلسوف‌ها یک چنگال در دست راست دارند اما چنگال سمت چپ آن‌ها توسط فیلسوف کناری برداشته شده‌است پس هیچ کدام از آن‌ها نمی‌توانند غذا بخورند. که همان حالت بن‌بست است که در آن هیچ پردازه‌ای نمیتواند اجرا شود.

راه حل مشکل استفاده از mutex است. به این صورت که هر کدام از چنگال‌ها را به صورت یک mutex تعریف می‌کنیم و زمانی که یک فیلسوف آن چنگال را برمیدارد، چنگال را lock می‌کنیم و هر وقت که غذاخوردنش تمام شد، آن را unlock می کنیم. اما همانطور که بالاتر گفتیم تنها همین کار کافی نیست و باید یک سمافور دیگر هم تعریف کنیم که ترتیب برداشتن چنگال‌ها را کنترل کند تا چنگال چپ و راست، همزمان برداشته شوند.

استفاده از سمافور با استفاده از semaphore POSIX به روش‌های متفاوتی می‌تواند صورت گیرد:

در حالت اول می‌توانیم کل فرآیندهای بعد از فکر کردن را یک ناحیه‌ی بحرانی در نظر بگیریم و با استفاده از یک سمافور، مطمئن شویم که تنها یک فیلسوف وارد این ناحیه می‌شود. در این حالت ما شروط انحصار متقابل(Mutual Exclusion)، پیشرفت(Progress) و انتظار محدود(Bounded Waiting) را برآورده کرده‌ایم اما این پیاده‌سازی بهترین بازدهی ممکن را ندارد چرا که در یک لحظه، تنها یک فیلسوف که داخل ناحیه‌ی بحرانی بعد از سمافور lock است می‌تواند غذا بخورد.

```
sem_t lock;
pthread_t phil[5];
pthread_mutex_t chop[5];

void *handle_philo (void *i){

    int id = (int)(long) i;
    while(1){

        //think
        think(id);
        //pickup
        sem_wait(&lock);
            pickup(id);
            //eat
            eat(id);
            //finish eating
            finish(id);
            //putdown
            putdown(id);
        sem_post(&lock);
    }
}

void pickup(int phil){
    pthread_mutex_lock(&chop[phil]);
    pthread_mutex_lock(&chop[(phil+1)%5]);
}

void putdown(int phil){
    pthread_mutex_unlock(&chop[phil]);
    pthread_mutex_unlock(&chop[(phil+1)%5]);
}
```

در حالت دوم یک سمافور pick_up و یک سمافور put_down را تعریف میکنیم. از سمافور pick_up استفاده میکنیم تا مطمئن شویم که چنگال چپ و راست به صورت همزمان و طی یک فرآیند اتمیک برداشته می‌شوند و از put_down هم استفاده می‌کنیم تا عملیات finish و پایین گذاشتن چنگال چپ و راست نیز با هم انجام شوند:

```
sem_t pick_up;
sem_t put_down;
pthread_t phil[5];
pthread_mutex_t chop[5];
int count[5];
```

```
void *handle_philo (void *i){

    int id = (int)(long) i;

    count[id] = 1;
    while(1){

        //think
        think(id);
        sleep(2);

        //pickup
        sem_wait(&pick_up);
            pickup(id);
            //eat
            eat(id);
        sem_post(&pick_up);

        sleep(2);

        //putdown
        sem_wait(&put_down);
            //finish eating
            finish(id);
            putdown(id);
        sem_post(&put_down);

        count[id]++;
    }
}
```

```
void pickup(int phil){
    pthread_mutex_lock(&chop[phil]);
    pthread_mutex_lock(&chop[(phil+1)%5]);
}
```

راست

چپ

```
void putdown(int phil){
    pthread_mutex_unlock(&chop[phil]);
    pthread_mutex_unlock(&chop[(phil+1)%5]);
}
```

در این حالت در هر لحظه تنها یک فیلسوف میتواند "اقدام به برداشتن چنگال" کند و این تصمیم با استفاده از pick_up نشان داده می‌شود. تا زمانی که هر دو چنگال چپ و راست را بردارد و sem_post(&pick_up) این حالت را نشان دهد، بقیه‌ی فیلسوف‌هایی که میخواهند "اقدام به برداشتن چنگال" کنند باید پشت sem_wait(&pick_up) منتظر بمانند.

با این پیاده‌سازی، حداکثر دو فیلسوف می‌توانند به صورت همزمان غذا بخورند.

از آرایه‌ی count استفاده کرده‌ایم تا میزان توزیع cpu بین این ۵ نخ را اندازه‌گیری کنیم و مطمئن شویم که قحطی رخ نمیدهد.

توضیحات اضافی:

استفاده از sleep برای آن است که بتوانیم جریان اجرا شدن کد را دنبال کنیم. به علاوه تمامی متغیرها، نخ‌ها و سمافورهای استفاده شده در این کد، در تابع main مقداردهی شده‌اند. توابع Eat، finish و think وضعیت فیلسوف phil را چاپ می‌کند.

انتظار می‌رود که با این پیاده‌سازی هر سه شرط انحصار متقابل، پیشرفت و انتظار محدود برآورده شوند و بن‌بست و قحطی نیز رخ ندهند.

خروجی‌های نمونه‌ی کد:

```
^C
oslab@OSLab-VirtualBox:~/Desktop/Lab 6$ ./DP
hi
Phil 2 is thinking
Phil 4 is thinking
Phil 0 is thinking
Phil 1 is thinking
Phil 3 is thinking
Phil 2 is eating with chop[2] and chop[3] for the [1]th time
Phil 4 is eating with chop[4] and chop[0] for the [1]th time
Phil 4 finished eating
Phil 4 is thinking
Phil 2 finished eating
Phil 2 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [1]th time
Phil 0 finished eating
Phil 0 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [1]th time
Phil 3 is eating with chop[3] and chop[4] for the [1]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 3 finished eating
Phil 3 is thinking
Phil 2 is eating with chop[2] and chop[3] for the [2]th time
Phil 4 is eating with chop[4] and chop[0] for the [2]th time
Phil 4 finished eating
Phil 4 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [2]th time
Phil 2 finished eating
Phil 2 is thinking
Phil 0 finished eating
Phil 0 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [2]th time
Phil 3 is eating with chop[3] and chop[4] for the [2]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 3 finished eating
Phil 3 is thinking
Phil 2 is eating with chop[2] and chop[3] for the [3]th time
Phil 4 is eating with chop[4] and chop[0] for the [3]th time
Phil 4 finished eating
Phil 4 is thinking
Phil 2 finished eating
Phil 2 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [3]th time
Phil 3 is eating with chop[3] and chop[4] for the [3]th time
```

```
Phil 0 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [3]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 2 is eating with chop[2] and chop[3] for the [3]th time
Phil 2 finished eating
Phil 2 is thinking
Phil 3 is eating with chop[3] and chop[4] for the [3]th time
Phil 3 finished eating
Phil 3 is thinking
Phil 4 is eating with chop[4] and chop[0] for the [3]th time
Phil 4 finished eating
Phil 4 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [4]th time
Phil 0 finished eating
Phil 0 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [4]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 2 is eating with chop[2] and chop[3] for the [4]th time
Phil 2 finished eating
Phil 2 is thinking
Phil 3 is eating with chop[3] and chop[4] for the [4]th time
Phil 3 finished eating
Phil 4 is eating with chop[4] and chop[0] for the [4]th time
Phil 3 is thinking
Phil 4 finished eating
Phil 4 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [5]th time
Phil 0 finished eating
Phil 0 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [5]th time
Phil 1 finished eating
```

```
Phil 4 finished eating
Phil 4 is thinking
Phil 4 is eating with chop[4] and chop[0] for the [39078]th time
Phil 4 finished eating
Phil 4 is thinking
Phil 4 is eating with chop[4] and chop[0] for the [39079]th time
Phil 4 finished eating
Phil 4 is thinking
Phil 3 is eating with chop[3] and chop[4] for the [48227]th time
Phil 3 finished eating
Phil 3 is thinking
Phil 3 is eating with chop[3] and chop[4] for the [48228]th time
Phil 3 finished eating
Phil 3 is thinking
Phil 3 is eating with chop[3] and chop[4] for the [48229]th time
Phil 3 finished eating
Phil 3 is thinking
Phil 3 is eating with chop[3] and chop[4] for the [48230]th time
Phil 3 finished eating
Phil 3 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [40582]th time
Phil 0 finished eating
Phil 0 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [40583]th time
Phil 0 finished eating
Phil 0 is thinking
Phil 0 is eating with chop[0] and chop[1] for the [40584]th time
Phil 2 is eating with chop[2] and chop[3] for the [48332]th time
Phil 0 finished eating
Phil 0 is thinking
Phil 2 finished eating
Phil 2 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [43038]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [43039]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [43040]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [43041]th time
Phil 1 finished eating
Phil 1 is thinking
Phil 1 is eating with chop[1] and chop[2] for the [43042]th time
Phil 1 finished eating
Phil 3 is eating with chop[3] and chop[4] for the [48231]th time
^C
```


THE END