بسمه تعالی

گزارش‌کار آزمایشگاه سیستم عامل

استاد مربوطه:

مهندس اوا انوری

آزمایش ششم:

Synchronization

اعضای گروه:

فروغ افخمی9831703

پاییز 1401

بخش 1:

توضیح کلی در مورد ازمایش:

در این بخش، یک فرایند writer و دو فرایند reader داریم که فرایند های reader فقط توانایی خواندن و فرایند writer قابلیت خواندن و نوشتن را دارد. در این مسئله چون از فضای اشتراکی استفاده می کنیم و متغیر countبین تمام فرایند ها مشترک است، امکان به وجود آمدن در زمان اجرا وجود دارد که باعث می شود نتایج عملکرد فرایند ها بر متغیر مشترک، به درستی اعمال نشود. مثال فرایندی متغیر را 3 بخواند در حالی که مقدار آن به 4 تغییر داده شده است. به همین دلیل از قفل در ناحیه بحرانی استفاده می کنیم که در اینجا از lock_mutex_pthreadاستفاده کرده ایم. دو قفل برای این برنامه لازم است، یک قفل برای متغیر مشترک countکه هنگامی که اولین فرایند(خواننده یا نویسنده) وارد ناحیه بحرانی می شود آن را قفل می کند. البته فرایندهای خواننده همزمان می توانند از روی متغیر بخوانند اما فرایند نویسنده نمی تواند. قفل دوم برای متغیر مشترک count_read است که تعداد فرایند های خواننده داخل ناحیه بحرانی را مشخص می کند. چون فرایند های خواننده می توانند هر کدام این متغیر را تغییر دهند، شرایط مسابقه ممکن است پیش بیاید که به همین منظور از قفل برای آن استفاده می کنیم. همچنین چون فرایند های خواننده متعددی همزمان می توانند بخوانند، باید قفل مربوط به متغیر مشترک اصلی (count) هنگامی آزاد شود که تمام فرایند های خواننده از ناحیه بحرانی مربوط به این متغیر خارج شده باشند که به همین منظور دانستن تعداد دقیق فرایند های خواننده ای که در حال خواندن هستند لازم است.

ابتدا یک struct به نام ShMemory تعریف می‌کنیم که در آن دو متغیر عدد صحیح برای شمارنده و شمارنده‌ی خواننده و دو متغیر سمافور نیز برای هر کدام وجود دارد.(در ادامه توضیح داده می‌شود)

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

#define READERS_NUM 2
#define MAX_COUNT 5
```

```
int shmid;

typedef struct {
    int count, read_count;
    pthread_mutex_t mutex;
    pthread_mutex_t rw_mutex;
} ShMemory;
```

توضیح کد بخش main:

سپس در main آبجکت از نوع ShMemory ،از حافظه ی اشتراکی attach میکنیم و مقدار اولیه ی countو count_reader را صفر میگذاریم. و متغیرهای سمافور را نیز مقداردهی اولیه میکنیم و حافظه را detachمیکنیم.همچنین در این بخش exceptionها را نیز handleکرده ایم.

سپس یک فرایند parent میسازیم و writer را برای ان صدا میزنیم.همچنین با استفاده از for دو فرایند فرزند دیگر ایجاد میکنیم و برای انها reader را صدا میزنیم.

در نهایت نیز با استفاده از یک for و waitکردن برای فرایند ها از اتمام کار فرایند های ایجاد شده اطمینان حاصل میکنیم.

سپس با استفاده از دستور کنترلی IPC_RMID حافظه مشترک مشخص شده توسط shmid را از سیستم حذف کنید و بخش حافظه مشترک و ساختار داده shmid_ds مرتبط با shmid را از میبریم.

کد بخش main:

```
int main(int argc, char const *argv[])
{
    if ((shmid = shmget(IPC_PRIVATE, sizeof(ShMemory), IPC_CREAT | 0666)) < 0)
{
        printf("ERROR: shmget");
        exit(-1);
    }

    ShMemory* shm_ptr;
    if ((shm_ptr = (ShMemory *)shmat(shmid, NULL, 0)) == (ShMemory *) -1) {
        printf("ERROR: shmat");
        exit(1);
    }

    pthread_mutexattr_t attr;
```

```
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);

    pthread_mutex_init(&(shm_ptr->mutex), &attr);
    pthread_mutex_init(&(shm_ptr->rw_mutex), &attr);

    if (shmdt(shm_ptr) == -1) {
        printf("ERROR: shmdt");
        exit(1);
    }

    int parent_id = getpid();

    // create writer process
    int pid = fork();
    if (pid == 0) {
        writer();
        return 0;
    }

    // create reader processes
    for (int i = 0; i < READERS_NUM; i++)
    {
        if (getpid() == parent_id)
            pid = fork();
        else
            break;
    }

    if (pid == 0) { // reader processes
        reader();
        return 0;
    }


    if (getpid() == parent_id) // parent process
    {
        // wait for writer and readers
        for (int i = 0; i < READERS_NUM + 1; i++)
        {
            wait(NULL);
        }
    }

    if(-1 == (shmctl(shmid, IPC_RMID, NULL)))
```

```
    {
        printf("ERROR: shmctl");
        exit(1);
    }
    return 0;
}
```

کد بخش **reader , writer**:

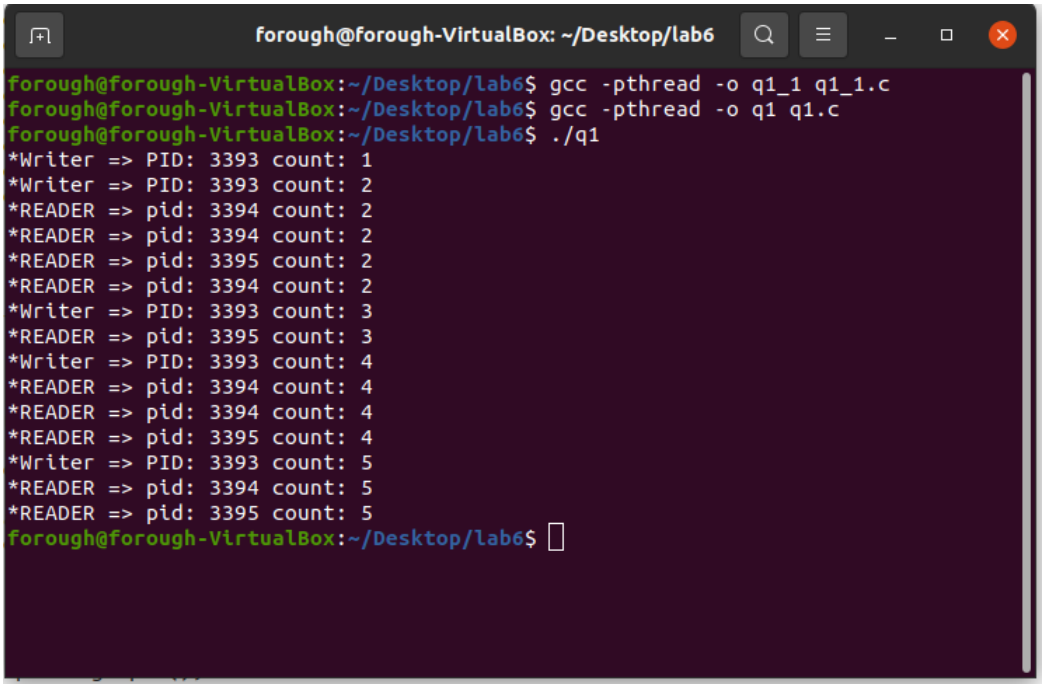
```
void reader() {
    ShMemory* shmem;
    if ((shmem = (ShMemory *)shmat(shmid, NULL, 0)) == (ShMemory *) -1) {
        print("ERROR: shmat");
        exit(1);
    }
    int pid = getpid();

    int finish = 0;
    while (finish == 0) {
        pthread_mutex_lock(&(shmem->mutex));
        (shmem->read_count)++;
        if (shmem->read_count == 1) {
            pthread_mutex_lock(&(shmem->rw_mutex));
        }
        pthread_mutex_unlock(&(shmem->mutex));

        printf("*READER => pid: %d count: %d\n", pid, shmem->count);
        if (shmem->count >= MAX_COUNT) {
            finish = 1;
        }

        pthread_mutex_lock(&(shmem->mutex));
        (shmem->read_count)--;
        if (shmem->read_count == 0) {
            pthread_mutex_unlock(&(shmem->rw_mutex));
        }
        pthread_mutex_unlock(&(shmem->mutex));
        sleep(0.1);
    }

    if (shmdt(shmem) == -1) {
        printf("ERROR: shmdt");
        exit(1);
    }
}
```

```
void writer() {
    ShMemory* shmem;
    if ((shmem = (ShMemory *)shmat(shmid, NULL, 0)) == (ShMemory *) -1) {
        printf("ERROR: shmdt");
        exit(1);
    }
    int pid = getpid();

    int finish = 0;
    while (finish == 0) {
        pthread_mutex_lock(&(shmem->rw_mutex));

        shmem->count++;
        printf("*Writer => PID: %d count: %d\n", pid, shmem->count);
        if (shmem->count >= MAX_COUNT) {
            finish = 1;
        }

        pthread_mutex_unlock(&(shmem->rw_mutex));
        sleep(0.05);
    }

    if (shmdt(shmem) == -1) {
        printf("ERROR: shmdt");
        exit(1);
    }
}
```

نحوه عملکرد reader و writer در ویدیو به طور کامل توضیح داده شد اما در اینجا نیز به صورت مختصر عملکرد این دو را توضیح میدهیم.

روش استفاده شده به این صورت است که هنگامی که اولین reader ای که شروع به خواندن از buffer می کند، writer را با استفاده از mutex متوقف می کند و وقتی آخرین reader کارش تموم می شود به writer سیگنال می دهیم که آزاد است که نوشتن را انجام دهد. برای اینکار به یک متغیر read_count نیاز داریم تا وقتی زمانی که reader ای شروع به خواندن می کند آن را یکی زیاد و وقتی کارش تمام شد آن را یکی کم کنیم. ممکن است برای این متغیر هم race condition رخ دهد، پس قبل از نوشتن در این متغیر، از یک mutex دیگر به نام rw_mutex استفاده می کنیم تا وقتی یک reader در حال نوشتن در آن است بقیه reader ها به آن دسترسی نداشته باشند.

نمونه خروجی :



همانطور که میبینیم شرایط مسابقه پیش نیامده و کد به درستی عمل کرده است.

بخش2 :

توضیح کلی بخش 2 :

در این بخش هر فیلسوف در واقع یک thread است که قصد دسترسی به chopstick های کنار خود برای غذا خوردن دارد. برای تامین انحصار متقابل در منابع مشترک(chopsticks)در این روش از mutex استفاده کرده ایم به این صورت که برای هر چوب یک lock_mutex داریم که هنگامی که یکی از فیلسوف ها بخواهد آنها را استفاده کند قفل شده و اتمام استفاده فیلسوف، unlock می شود تا فیلسوف های دیگر بتوانند از آن استفاده کنند.

به اختصار در این بخش می‌خواهیم مسئله‌ی فیلسوف های غذاخور را پیاده سازی کنیم و از بروز شرایط مسابقه جلوگیری کنیم.

توضیح تابع philosopher:

قبل از آن *void می گذاریم تا بتوانیم به pthread_create() آن را پاس بدهیم. در این تابع با توجه به تعداد دفعات تعریف شده برای غذا خوردن فیلسوفان، عملیات lock کردن چوب غذای مورد نظر، از بین پنج چوب انجام می‌شود و بعد از تمام شدن کار، چوب غذا آزاد می‌شود. این تابع با توجه به ورودی پیغام مناسب را برای هر مرحله چاپ می‌کند.

تابع philosopher:

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

#define EAT_TIMES 1

pthread_mutex_t chopstick[5];

void *philosopher(int n)
{
    for (int i = 0; i < EAT_TIMES; i++)
    {
        printf("Philosopher %d is thinking.\n", n + 1);
```

```
        pthread_mutex_lock(&chopstick[n]);
        pthread_mutex_lock(&chopstick[(n + 1) % 5]);

        printf("Philosopher %d is eating using chopstick[%d] and
chopstick[%d].\n", n + 1, n, (n + 1) % 5);
        sleep(1);

        pthread_mutex_unlock(&chopstick[n]);
        pthread_mutex_unlock(&chopstick[(n + 1) % 5]);

        printf("Philosopher %d finished eating.\n", n + 1);
    }
}
```

در تابع main پنج thread را تعریف کرده و با pthread_mutex_init() پنج قفل می‌سازیم. سپس برای ساخت thread ها، تابع philosopher را به pthread_create() پاس می‌دهیم. در حلقه آخر نیز با pthread_join() مطمئن می‌شویم که کار هر ۵ thread تمام شود.

کد main:

```
int main()
{
    pthread_t threads[5];

    for (int i = 0; i < 5; i++)
    {
        pthread_mutex_init(&chopstick[i], NULL);
    }

    for (int i = 0; i < 5; i++)
    {
        pthread_create(&threads[i], NULL, (void *) philosopher, (void
*)(intptr_t) i);
    }

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

    return 0;
```

```
}
```

در ادامه دو نمونه خروجی را می‌بینیم:

```
forough@forough-VirtualBox:~/Desktop/lab6$ gcc -pthread -o q2 q2.c
forough@forough-VirtualBox:~/Desktop/lab6$ ./q2
Philosopher 1 is thinking.
Philosopher 1 is eating using chopstick[0] and chopstick[1].
Philosopher 3 is thinking.
Philosopher 3 is eating using chopstick[2] and chopstick[3].
Philosopher 4 is thinking.
Philosopher 2 is thinking.
Philosopher 5 is thinking.
Philosopher 1 finished eating.
Philosopher 3 finished eating.
Philosopher 5 is eating using chopstick[4] and chopstick[0].
Philosopher 2 is eating using chopstick[1] and chopstick[2].
Philosopher 5 finished eating.
Philosopher 4 is eating using chopstick[3] and chopstick[4].
Philosopher 2 finished eating.
Philosopher 4 finished eating.
forough@forough-VirtualBox:~/Desktop/lab6$ ./q2
Philosopher 2 is thinking.
Philosopher 2 is eating using chopstick[1] and chopstick[2].
Philosopher 1 is thinking.
Philosopher 4 is thinking.
Philosopher 4 is eating using chopstick[3] and chopstick[4].
Philosopher 3 is thinking.
Philosopher 5 is thinking.
Philosopher 2 finished eating.
Philosopher 1 is eating using chopstick[0] and chopstick[1].
Philosopher 4 finished eating.
Philosopher 3 is eating using chopstick[2] and chopstick[3].
Philosopher 1 finished eating.
Philosopher 5 is eating using chopstick[4] and chopstick[0].
Philosopher 3 finished eating.
Philosopher 5 finished eating.
forough@forough-VirtualBox:~/Desktop/lab6$
```

با توجه به نمونه خروجی‌های زیر، برنامه به طور صحیح کار می‌کند و هرگز یک چوب هم‌زمان در اختیار دو فیلسوف قرار نمی‌گیرد.

سوال: آیا ممکن است بن بست رخ دهد؟ در صورت امکان چگونگی ایجاد آن را توضیح دهید.

بله؛ امکان به وجود آمدن بن بست وجود دارد چون تعداد منابع مورد نیاز برای غذا خوردن بیش از یک است و برداشتن و قفل چوب ها می تواند به شکل دورانی انجام بگیرد به این شکل فیلسوف ها همزمان قصد غذا خوردن داشته باشند هر کدام چوب سمت راستی خود را بردارند. در این حالت هر فیلسوف یک چوب برداشته و منتظر می شود تا فیلسوف سمت چپ چوب خود را زمین بگذارد که چون هرکدام یک چوب دارند، پیشرفت حاصل نشده و هیچکدام نمی توانند غذا بخورند و زمانی نامتناهی را برای آزاد شدن منابع باید صبر کنند که این موضوع همان بن بست را ایجاد می کند. همچنین باید توجه شود امکان به وجود آمدن قحطی هم وجود دارد چون فیلسوف ها نوبت دهی نشده اند و ممکن است یک فیلسوف به خاطر زودتر برداشتن چوب ها، چندین بار

غذا بخورد در حالی که فیلسوف دیگر هنوز غذا نخورده است. پس این الگوریتم از fairness نسبی برای جلوگیری از ایجاد قحطی برخوردار نیست.