به نام خدا

دانشکده مهندسی کامپیوتر

ازمایشگاه سیستم های عامل

ازمایش نهم : سمافورها (Semaphores)

اعضای گروه:

حسین تاتار – 40133014

محمد امین فرح بخش – 40131029

خرداد 1404

تمرین 1: برنامه اي بنويسيد كه همگام سازي بين چندين نخ را برقرار كند. نخها سعی میکنند به منبعی به اندازه 2 دسترسی پیدا کنند. به سوالات زیر پاسخ دهید.

جواب: کد این تمرین به صورت زیر میباشد:

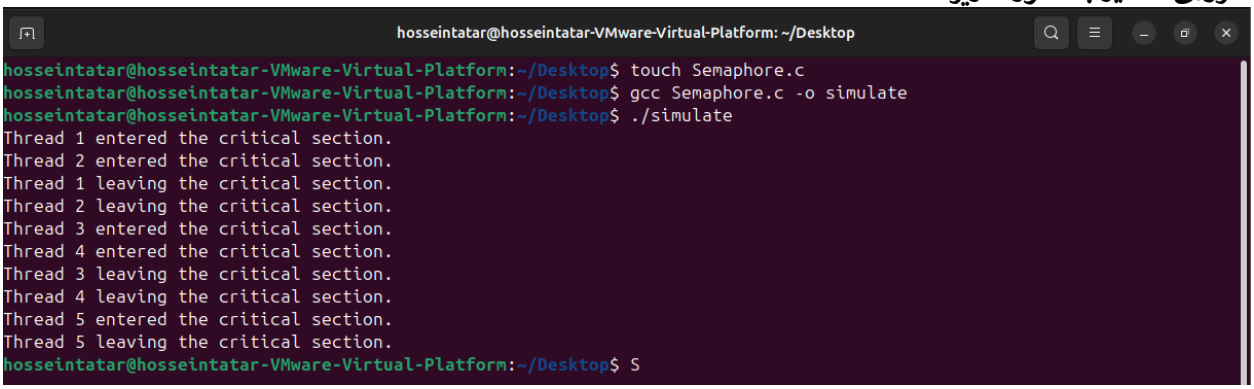
Semaphore.c
~/Desktop

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

#define THREAD_NUM 5

sem_t sem;

void* access_resource(void* arg) {
    int tid = *((int*)arg);

    sem_wait(&sem); // Attempt to access the resource (decrement semaphore)
    printf("Thread %d entered the critical section.\n", tid);

    sleep(1); // Simulate resource usage

    printf("Thread %d leaving the critical section.\n", tid);
    sem_post(&sem); // Release the resource (increment semaphore)

    return NULL;
}
int main() {
    pthread_t threads[THREAD_NUM];
    int ids[THREAD_NUM];

    // Initialize semaphore with value 2 (allows 2 threads to access simultaneously)
    sem_init(&sem, 0, 2);

    for (int i = 0; i < THREAD_NUM; ++i) {
        ids[i] = i + 1;
        pthread_create(&threads[i], NULL, access_resource, &ids[i]);
    }

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

    sem_destroy(&sem);

    return 0;
}
```

این برنامه یک شبیه‌سازی از دسترسی چند نخ (Thread) به یک منبع مشترک با ظرفیت محدود (مثلاً ظرفیت ۲) است.
در اینجا:

- از سمافور شمارشی استفاده شده که مقدار اولیه‌اش ۲ است.
- برنامه ۵ نخ (thread) ایجاد می‌کند.
- هر نخ سعی می‌کند وارد «بخش بحرانی» شود؛ یعنی جایی که باید از منبع مشترک استفاده کند.
- تنها ۲ نخ می‌توانند به‌طور هم‌زمان وارد بخش بحرانی شوند. چون مقدار اولیه سمافور ۲ است.
- سایر نخ‌ها منتظر می‌مانند تا یکی از نخ‌ها منبع را آزاد کند با sem_post
- هر نخ پس از استفاده از منبع برای ۱ ثانیه با sleep(1) ، آن را آزاد می‌کند و نخ بعدی اجازه ورود می‌گیرد.
- در نهایت، برنامه تا پایان کار همه نخ‌ها صبر می‌کند (pthread_join) و بعد به پایان می‌رسد.

خروجی کد نیز به صورت زیر است:



الف) مقدار اولیه متغیر سمافور چیست؟

مقدار اولیه سمافور تعیین‌کننده **تعداد مجاز دسترسی همزمان به منبع مشترک** است. در مثال بالا مقدار اولیه برابر با 2 است چون فقط دو نخ می‌توانند همزمان وارد بخش بحرانی شوند.

ب) چرا از تابع pthread_join()در برنامه استفاده می‌کنیم؟

برای اینکه **برنامه اصلی منتظر اتمام اجرای نخ‌ها بماند** .اگر این تابع را استفاده نکنیم، ممکن است برنامه اصلی پیش از پایان نخ‌ها خاتمه یابد و خروجی ناقص باشد یا نخی اجرا نشود.

ج) چرا پارامتر چهارم در pthread_create() برابر با NULL است؟

این پارامتر، آرگومان ورودی تابع نخ است. در صورتی که نیاز به ارسال اطلاعات خاصی نباشد می‌توان آن را NULL قرار داد. البته در کد بالا از آن استفاده کردیم و مقدار آدرس &ids[i] را دادیم. وقتی NULL می‌دهیم یعنی تابع نخ به هیچ آرگومانی نیاز ندارد.

د) اهمیت استفاده از sleep(1) در توابع fun1 و fun2 چیست؟

sleep(1) باعث می‌شود اجرای نخ برای یک ثانیه متوقف شود. اینکار باعث میشود که:

- شبیه‌سازی استفاده از منبع (برای مشاهده تأثیر سمافور)
- کمک به مشاهده همزمانی یا هم‌زمان نبودن اجرای نخ‌ها
- جلوگیری از تداخل سریع نخ‌ها

هـ) چگونه از سمافورهای شمارشی استفاده کنیم؟

سمافور شمارشی با مقدار اولیه بیشتر از 1 ساخته می‌شود و نشان‌دهنده تعداد منابع در دسترس است. وقتی نخ وارد بخش بحرانی می‌شود sem_wait() مقدار سمافور را کاهش می‌دهد و وقتی کارش تمام شد sem_post() آن را افزایش می‌دهد. اگر مقدار سمافور به 0 برسد، نخ‌های دیگر منتظر می‌مانند تا منابع آزاد شوند.

تمرین 2: برنامه خوانندگان نویسندگان را بصورت کامل نوشته و سپس اجرا کنید. به سوالات زیر پاسخ دهید :

- ❖ آیا مشکلی وجود دارد؟
- ❖ در صورت وجود ناهماهنگی چه راهکاري ارائه میکنید؟

جواب: برای حل این مسئله، از **سمافورها** و **موتکس** استفاده می‌کنیم تا:

- **خوانندگان (Readers)** بتوانند **همزمان** به داده دسترسی داشته باشند.
- **نویسنده (Writer)** فقط وقتی بتواند بنویسد که **هیچ خواننده‌ای** در حال خواندن نباشد.

کد برنامه به صورت زیر است:

Open — rw.c — ~/Desktop

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

#define MAX_COUNT 10  // Terminate when buffer reaches this value

int buffer = 0;       // Shared data buffer
int readers_count = 0; // Count of active readers

sem_t mutex;          // Protects readers_count variable
sem_t rw_mutex;       // Controls access for writers vs readers

void *writer(void *arg) {
    int pid = *(int *)arg;
    while (buffer < MAX_COUNT) {
        sem_wait(&rw_mutex); // Wait until no readers are active

        // Critical Section (Writing)
        buffer++;
        printf("Writer (PID: %d) wrote count: %d\n", pid, buffer);

        sem_post(&rw_mutex); // Release lock
        sleep(1); // Simulate processing time
    }
    pthread_exit(NULL);
}
void *reader(void *arg) {
    int pid = *(int *)arg;
    while (buffer < MAX_COUNT) {
        // Entry Protocol for Readers
        sem_wait(&mutex); // Lock readers_count
        readers_count++;
        if (readers_count == 1) {
            sem_wait(&rw_mutex); // First reader blocks writers
        }
        sem_post(&mutex);

        // Reading Section (Multiple readers can enter)
        printf("Reader (PID: %d) read count: %d\n", pid, buffer);

        // Exit Protocol for Readers
        sem_wait(&mutex);
        readers_count--;
        if (readers_count == 0) {
            sem_post(&rw_mutex); // Last reader releases writers
        }
        sem_post(&mutex);

        sleep(1); // Simulate processing time
    }
    pthread_exit(NULL);
}
```

```c
int main() {
    pthread_t writer_thread, reader1_thread, reader2_thread;
    int writer_pid = 1, reader1_pid = 2, reader2_pid = 3;

    // Initialize semaphores
    sem_init(&mutex, 0, 1);    // Binary semaphore for readers_count
    sem_init(&rw_mutex, 0, 1); // Binary semaphore for writer access

    // Create threads
    pthread_create(&writer_thread, NULL, writer, &writer_pid);
    pthread_create(&reader1_thread, NULL, reader, &reader1_pid);
    pthread_create(&reader2_thread, NULL, reader, &reader2_pid);

    // Wait for threads to finish
    pthread_join(writer_thread, NULL);
    pthread_join(reader1_thread, NULL);
    pthread_join(reader2_thread, NULL);

    // Cleanup semaphores
    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);

    return 0;
}
```

در این برنامه ما داریم که :

- یک متغیر مشترک count داریم که مقدار اولیه‌اش صفر است.
- **یک writer** داریم که مقدار count را در هر دسترسی یک واحد افزایش می‌دهد و چاپ می‌کند.
- **دو reader** داریم که فقط مقدار count را می‌خوانند و آن را همراه با PID خود چاپ می‌کنند.
- چند reader می‌توانند همزمان بخوانند اما **در زمان نوشتن نباید هیچ reader یا writer دیگری به بافر دسترسی داشته باشد.**

خروجی برنامه به این صورت است:

```
hosseintatar@hosseintatar-VMware-Virtual-Platform:~/Desktop$ gcc rw.c -o simulate
hosseintatar@hosseintatar-VMware-Virtual-Platform:~/Desktop$ ./simulate
Writer (PID: 1) wrote count: 1
Reader (PID: 3) read count: 1
Reader (PID: 2) read count: 1
Writer (PID: 1) wrote count: 2
Reader (PID: 3) read count: 2
Reader (PID: 2) read count: 2
Writer (PID: 1) wrote count: 3
Reader (PID: 3) read count: 3
Reader (PID: 2) read count: 3
Writer (PID: 1) wrote count: 4
Reader (PID: 3) read count: 4
Reader (PID: 2) read count: 4
Writer (PID: 1) wrote count: 5
Reader (PID: 3) read count: 5
Reader (PID: 2) read count: 5
Writer (PID: 1) wrote count: 6
Reader (PID: 3) read count: 6
Reader (PID: 2) read count: 6
Writer (PID: 1) wrote count: 7
Reader (PID: 3) read count: 7
Reader (PID: 2) read count: 7
Writer (PID: 1) wrote count: 8
Reader (PID: 3) read count: 8
Reader (PID: 2) read count: 8
Writer (PID: 1) wrote count: 9
Reader (PID: 3) read count: 9
Reader (PID: 2) read count: 9
Writer (PID: 1) wrote count: 10
```

الف) آیا مشکلی وجود دارد؟

اگر همگام‌سازی انجام نشود (یعنی از سمافور استفاده نکنیم)، احتمال دارد چند reader یا یک reader و writer همزمان به متغیر دسترسی پیدا کنند، که باعث بروز شرط رقابت (Race Condition) و ناهماهنگی می‌شود.
همچنین مشکل Starvation (گرسنگی) نویسنده هم وجود دارد. خواننده‌ها پشت‌سر هم وارد بافر می‌شوند و چون شرط دسترسی نویسنده این است که هیچ خواننده‌ای فعال نباشد، در عمل دیگه هیچ‌وقت به نویسنده نوبت نمی‌رسد مخصوصاً اگر خواننده‌ها دائم فعال باشند.

ب) در صورت وجود ناهماهنگی چه راهکاری ارائه می‌کنید؟

1. اولویت دادن به نویسنده:
   - وقتی یک نویسنده منتظر است، خوانندگان جدید بلاک شوند تا نویسنده کارش را تمام کند.
   - اینکار با یک سمافور اضافی (write_priority) قابل پیاده‌سازی است.
2. محدود کردن تعداد خوانندگان همزمان:
   - اگر تعداد خوانندگان خیلی زیاد شود، نویسنده اصلاً فرصت نوشتن پیدا نمی‌کند.
   - می‌توان یک حداکثر تعداد خوانندگان همزمان تعریف کرد (مثلاً ۳ خواننده).
3. استفاده از pthread_rwlock در سیستم‌عامل‌های مدرن:
   - این مکانیزم بهینه‌تر از سمافورها عمل می‌کند و از قفل‌های خواندن-نوشتن پشتیبانی می‌کند.

تمرین 3: کد مسئله فیلسوفان غذاخور را پیاده سازي کنید و خروجی برنامه را به مدرس خود تحویل دهید. آیا ممکن است بن بست رخ دهد؟ در صورت امکان چگونگی ایجاد آن را توضیح دهید.

جواب: پیاده‌سازی مسئله فیلسوفان غذاخور (Dining Philosophers) در C با استفاده از سمافور ها:

Open eating_philosophers.c ~/Desktop

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

#define NUM_PHILOSOPHERS 5  // Number of philosophers/chopsticks

sem_t chopsticks[NUM_PHILOSOPHERS]; // One semaphore per chopstick

void *philosopher(void *arg) {
    int id = *(int *)arg;
    int left = id;
    int right = (id + 1) % NUM_PHILOSOPHERS; // Right chopstick wraps around

    while (1) {
        // Thinking phase
        printf("Philosopher %d is thinking...\n", id);
        sleep(1);

        // Attempt to pick up chopsticks
        printf("Philosopher %d tries to pick left chopstick (%d)\n", id, left);
        sem_wait(&chopsticks[left]); // Wait for left chopstick
        printf("Philosopher %d picks the left chopstick (%d)\n", id, left);

        printf("Philosopher %d tries to pick right chopstick (%d)\n", id, right);
        sem_wait(&chopsticks[right]); // Wait for right chopstick
        printf("Philosopher %d picks the right chopstick (%d)\n", id, right);

        // Eating phase
        printf("Philosopher %d begins to eat\n", id);
        sleep(1); // Simulate eating time

        // Release chopsticks
        printf("Philosopher %d has finished eating\n", id);
        sem_post(&chopsticks[left]); // Release left chopstick
        printf("Philosopher %d leaves the left chopstick (%d)\n", id, left);
        sem_post(&chopsticks[right]); // Release right chopstick
        printf("Philosopher %d leaves the right chopstick (%d)\n", id, right);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int ids[NUM_PHILOSOPHERS];

    // Initialize all chopstick semaphores to 1 (available)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&chopsticks[i], 0, 1);
    }

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }

    // Wait for all threads (though they run indefinitely)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Cleanup semaphores (unreachable in this infinite loop example)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&chopsticks[i]);
    }

    return 0;
}
```

اجزای اصلی کد شامل بخش های زیر است:

1. سمافورها(Semaphores) : هر چوب غذاخوری با یک سمافور (sem_t) نمایش داده می‌شود. مقدار اولیه هر سمافور ۱ است (یعنی چوب غذاخوری آزاد است).
2. فیلسوف‌ها(Threads) : هر فیلسوف یک نخ (thread) جداگانه است. رفتار در تابع philosopher() تعریف شده است برای هر فیلسوف.
3. رفتار هر فیلسوف: فکر کردن (thinking)، برداشتن چوب چپ، برداشتن چوب راست، غذا خوردن (eating) و یا رها کردن چوب‌ها. برای برداشتن چوب‌ها از sem_wait() (کاهش سمافور) و برای رها کردن از sem_post() (افزایش سمافور) استفاده می‌شود.

خروجی کد نیز به این صورت میباشد:

(خروجی برنامه باید شبیه تصویر داده‌شده باشد، اما ممکن است در اجراهای مختلف رفتار متفاوتی نشان دهد (بسته به زمان‌بندی نخ‌ها) ولی در کل این یک نمونه از خروجی است.)

```
hosseintatar@hosseintatar-VMware-Virtual-Platform:~/Desktop$ touch eating_philosophers.c
hosseintatar@hosseintatar-VMware-Virtual-Platform:~/Desktop$ gcc eating_philosophers.c -o simulate
hosseintatar@hosseintatar-VMware-Virtual-Platform:~/Desktop$ ./simulate
Philosopher 0 is thinking...
Philosopher 2 is thinking...
Philosopher 1 is thinking...
Philosopher 3 is thinking...
Philosopher 4 is thinking...
Philosopher 0 tries to pick left chopstick (0)
Philosopher 0 picks the left chopstick (0)
Philosopher 0 tries to pick right chopstick (1)
Philosopher 0 picks the right chopstick (1)
Philosopher 0 begins to eat
Philosopher 1 tries to pick left chopstick (1)
Philosopher 2 tries to pick left chopstick (2)
Philosopher 2 picks the left chopstick (2)
Philosopher 2 tries to pick right chopstick (3)
Philosopher 2 picks the right chopstick (3)
Philosopher 2 begins to eat
Philosopher 4 tries to pick left chopstick (4)
Philosopher 4 picks the left chopstick (4)
Philosopher 4 tries to pick right chopstick (0)
Philosopher 3 tries to pick left chopstick (3)
Philosopher 0 has finished eating
Philosopher 0 leaves the left chopstick (0)
Philosopher 2 has finished eating
Philosopher 2 leaves the left chopstick (2)
Philosopher 2 leaves the right chopstick (3)
Philosopher 2 is thinking...
Philosopher 1 picks the left chopstick (1)
Philosopher 1 tries to pick right chopstick (2)
Philosopher 1 picks the right chopstick (2)
```

سوال : آیا ممکن است بن‌بست (Deadlock) رخ دهد؟

بله زیرا این پیاده‌سازی مستعد بن‌بست است. که شرایط ایجاد بن‌بست به این صورت میباشد:

همه فیلسوفان همزمان چوب سمت چپ را بردارند: هر فیلسوف i، چوب i را برمی‌دارد (مثلاً فیلسوف ۰ چوب ۰، فیلسوف ۱ چوب ۱، و ...). در این صورت هیچ فیلسوفی نمی‌تواند چوب سمت راستش را بردارد، چون همه چوب‌ها قفل شده‌اند، در نتیجه همه فیلسوفان منتظر می‌مانند و سیستم قفل می‌شود.

خروجی محتمل در بن‌بست در کد میتواند به صورت زیر باشد:

Philosopher 0 picks left chopstick 0

Philosopher 1 picks left chopstick 1

Philosopher 2 picks left chopstick 2

Philosopher 3 picks left chopstick 3

Philosopher 4 picks left chopstick 4

کد اجرا نشده و در یک حالت قفل میرود. (همه منتظر چوب راست میمانند).

راه‌حل‌های جلوگیری از بن‌بست

1. حداقل یک فیلسوف چوب‌ها را با ترتیب معکوس بردارد: مثلاً فیلسوف ۴ اول چوب ۰ را بردارد (به جای چوب ۴). با اینکار حلقه‌ی انتظار را می‌شکند.
2. استفاده از sem_trywait به جای sem_wait: اگر چوب راست موجود نبود، چوب چپ را رها کند.
3. محدود کردن تعداد فیلسوفان همزمان: فقط N-1فیلسوف اجازهٔ غذا خوردن داشته باشند.
4. الگوریتم سلسله‌مراتبی(Hierarchical) : که در ان چوب‌ها را شماره‌گذاری کنیم و فیلسوفان همیشه اول چوب با شمارهٔ کمتر را بردارند.