

گزارش دستورکار ششم آزمایشگاه سیستم‌های عامل

نگار موقتیان، ۹۸۳۱۰۶۲

۱. مساله خوانندگان - نویسندگان

در این قسمت از آزمایش مطابق دستورکار می‌خواهیم دو پرده خواننده و یک پرده نویسنده ایجاد کنیم که بر روی یک خانه مشترک از حافظه داده‌ای نوشته و یا از روی آن می‌خوانند. در برنامه نوشته شده یک پدر با دو فرزند داریم که پدر وظیفه آپدیت کردن مقدار count بر روی حافظه و فرزندان وظیفه خواندن آن را دارند. همچنین شرط خاتمه خواندن و نوشتن رسیدن مقدار count به عدد ۲ (رقم آخر شماره دانشجویی) در نظر گرفته شده‌است.

در ابتدای برنامه کتابخانه‌های مورد نیاز اضافه شده‌اند.

سپس یک ثابت تعریف شده که مقدار بیشینه count را نشان می‌دهد. پس از آن نیز دو متغیر به عنوان ID حافظه مشترک و مقداری که فراخوانی سیستمی fork() برمی‌گرداند تعریف شده‌اند.

در تابع main ابتدا با استفاده از دستور shmget حافظه مشترک را اختصاص داده‌ایم (با سایزی به اندازه یک int).

در ادامه با استفاده از دستور fork() && fork() دو پرده فرزند برای پرده اصلی ایجاد کرده‌ایم که قرار است به عنوان Reader کار کنند. پس از آن نیز حافظه مشترک را برای هر پرده بازیابی می‌کنیم.

حال اگر داخل پرده پدر بودیم (Writer) تا زمانی که به بیشینه مقدار دلخواه count نرسیده‌ایم مقدار خانه حافظه مشترک را افزایش داده و پیغام مناسب چاپ می‌کنیم. سپس با تمام شدن این عملیات منتظر پرده‌های فرزند می‌مانیم تا کار خود را به اتمام برسانند و بتوانیم حافظه مشترک را آزاد کنیم.

اگر هم داخل پرده‌های فرزند بودیم تا زمانی که به بیشینه مقدار دلخواه count نرسیده‌ایم مقدار خانه حافظه مشترک را خوانده و پیغام مناسب چاپ می‌کنیم.

```

#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>
#include<sys/time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<sys/ipc.h>
#include<sys/shm.h>

// student ID: 9831062
# define LIMIT 2
int memoryID, pid = 1;

int main(int argc, char *argv[]) {
    // initiate the shared memory
    memoryID = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0666);

    // create child processes
    pid = fork() && fork();

    // retrieve the shared memory
    int* counter = (int *) shmat(memoryID, NULL, 0);

    if (pid > 0) { // we are in the main (Writer) process
        while (*counter < LIMIT) {
            // increment the counter
            *counter = *counter + 1;
            printf("%d: updated counter to %d\n", getpid(), *counter);
        }

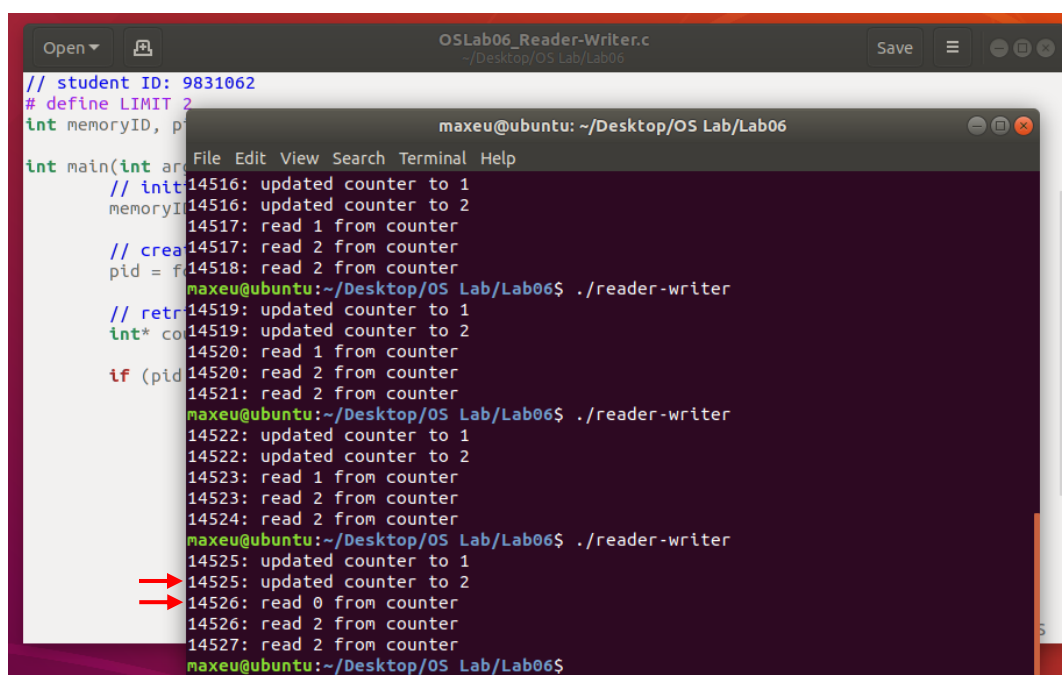
        // wait for the Readers to finish
        wait(NULL); wait(NULL);

        // free the shared memory
        shmctl(memoryID, IPC_RMID, NULL);
    }
    else { // we are in one of the child (Reader) processes
        int tmp;
        while ((tmp = *counter) < LIMIT)
            printf("%d: read %d from counter\n", getpid(), tmp);
        printf("%d: read %d from counter\n", getpid(), tmp);
    }

    return 0;
}

```

حال می‌توانیم برنامه مورد نظر را اجرا کنیم. در ابتدا ممکن است به نظر برسد که برنامه مشکلی ندارد اما با چند بار اجرای آن به نتیجه‌ای مانند زیر می‌رسیم.



The image shows a code editor window titled 'OSLab06_Reader-Writer.c' with the following code:

```
// student ID: 9831062
#define LIMIT 2
int memoryID, p;

int main(int argc, char* argv[]) {
    // init
    memoryID = 0;
    // create a new process
    pid = fork();
    // return to the parent process
    if (pid < 0) {
        // error
        return -1;
    }
    // parent process
    for (int i = 0; i < LIMIT; i++) {
        // write to memory
        printf("14516: updated counter to 1\n");
        printf("14516: updated counter to 2\n");
        // read from memory
        printf("14517: read 1 from counter\n");
        printf("14517: read 2 from counter\n");
        // read from memory
        printf("14518: read 2 from counter\n");
    }
    // child process
    for (int i = 0; i < LIMIT; i++) {
        // write to memory
        printf("14519: updated counter to 1\n");
        printf("14519: updated counter to 2\n");
        // read from memory
        printf("14520: read 1 from counter\n");
        printf("14520: read 2 from counter\n");
        // read from memory
        printf("14521: read 2 from counter\n");
    }
    // child process
    for (int i = 0; i < LIMIT; i++) {
        // write to memory
        printf("14522: updated counter to 1\n");
        printf("14522: updated counter to 2\n");
        // read from memory
        printf("14523: read 1 from counter\n");
        printf("14523: read 2 from counter\n");
        // read from memory
        printf("14524: read 2 from counter\n");
    }
    // child process
    for (int i = 0; i < LIMIT; i++) {
        // write to memory
        printf("14525: updated counter to 1\n");
        printf("14525: updated counter to 2\n");
        // read from memory
        printf("14526: read 0 from counter\n");
        printf("14526: read 2 from counter\n");
        // read from memory
        printf("14527: read 2 from counter\n");
    }
}
```

The terminal window shows the output of the program. The parent process (pid 14516) and the child process (pid 14517) are running. The output shows that the parent process updates the counter to 1 and then to 2, and the child process reads the counter. The output is as follows:

```
14516: updated counter to 1
14516: updated counter to 2
14517: read 1 from counter
14517: read 2 from counter
14518: read 2 from counter
maxeu@ubuntu:~/Desktop/OS Lab/Lab06$ ./reader-writer
14519: updated counter to 1
14519: updated counter to 2
14520: read 1 from counter
14520: read 2 from counter
14521: read 2 from counter
maxeu@ubuntu:~/Desktop/OS Lab/Lab06$ ./reader-writer
14522: updated counter to 1
14522: updated counter to 2
14523: read 1 from counter
14523: read 2 from counter
14524: read 2 from counter
maxeu@ubuntu:~/Desktop/OS Lab/Lab06$ ./reader-writer
14525: updated counter to 1
14525: updated counter to 2
14526: read 0 from counter
14526: read 2 from counter
14527: read 2 from counter
maxeu@ubuntu:~/Desktop/OS Lab/Lab06$
```

Two red arrows point to the lines '14526: read 0 from counter' and '14526: read 2 from counter' in the terminal output, indicating a race condition where the counter is read incorrectly.

همانطور که مشاهده می‌شود پردازش پدر مقدار ۲ را بر روی حافظه مشترک نوشته اما پردازش فرزند پس از آن مقدار صفر را چاپ کرده‌است. بنابراین در کد نوشته شده شرایط مسابقه وجود دارد و دلیل آن این است که بدون هیچ مکانیزم همگام سازی پردازش‌های بر روی حافظه مشترک مقدار نوشته و دیگری آن را می‌خواند. بنابراین اگر برای مثال ابتدا پردازش فرزند مقدار count را خوانده، پردازش پدر مقدار count را دو بار افزایش داده و دوباره نوبت به پردازش فرزند برسد تا مقدار count را چاپ کند مقدار صفر چاپ خواهد شد، در حالی که مقدار اصلی count در این لحظه برابر با ۲ است.

برای حل این مشکل می‌توان از یک mutex lock با قفل مشترک (از نوع pthread_mutex_t) میان پردازش‌ها استفاده کرد، به این نحو که پیش از هر خواندن و یا نوشتن بر حافظه مشترک count، قفل را در دست بگیریم (با استفاده از دستور pthread_mutex_lock) و پس از اتمام کار با این حافظه قفل را آزاد کنیم (با استفاده از دستور pthread_mutex_unlock). در این صورت می‌توانیم مطمئن باشیم دو پردازش به طور همزمان بر روی این حافظه مشترک کار نمی‌کنند و شرایط مسابقه پیش نخواهد آمد.

۲. مسأله فیلسوف‌های غذاخور

در این قسمت از آزمایش می‌خواهیم مطابق توضیحات دستورکار راه حلی برای مسأله فیلسوف‌های غذاخور ارائه کنیم. در این برنامه پنج thread داریم که هر کدام نمایانگر یک فیلسوف هستند و ۵ قفل از نوع mutex lock داریم که قرار است دسترسی به chopstick ها را کنترل و محدود کند.

در تابع thinkAndEat رفتار هر یک از فیلسوفان مشخص می‌شود. هر فیلسوف تا زمانی که بتواند هر دو chopstick دو طرفش را بردارد (دو منبع مورد نیاز خود را acquire کند) فکر می‌کند، سپس به محض این که هر دو chopstick دو طرفش به او اختصاص داده شد به مدت ۱ ثانیه (برای شبیه‌سازی) شروع به خوردن غذا می‌کند و پس از اتمام آن آن‌ها را به زمین می‌گذارد (منابع را release می‌کند) تا فیلسوف‌های دیگر بتوانند از آن استفاده کنند.

در این روش هم امکان deadlock و هم امکان قحطی وجود دارد، در حقیقت مدت زمان غذاخوردن فیلسوف‌ها مشخص است اما مدت زمان فکر کردن آن‌ها مشخص نیست و به عبارتی برای دسترسی به منابع bounded waiting نداریم.

به طور دقیق‌تر deadlock زمانی پیش می‌آید که همزمان تمام فیلسوف‌ها بخواهند غذا بخورند و طبق کد نوشته شده chopstick سمت راست خود را در ابتدا بردارند. حال هیچ chopstick ای بر روی میز نیست و هیچ یک از فیلسوف‌ها نمی‌تواند چیزی بخورد زیرا برای غذا خوردن به chopstick سمت چپ خود نیز نیاز دارند، در حالی که توسط فیلسوف سمت چپ اشغال شده‌است.

و اما قحطی زمانی ممکن است پیش بیاید که با توجه به شرایط چند فیلسوف خوش‌شانس دائماً پیش از فیلسوف‌های دیگر دو chopstick مورد نیاز خود را بدست بگیرند و اجازه استفاده از منابع را به دیگر فیلسوفان ندهد.

```

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

pthread_t philosophers[5];
pthread_mutex_t chopstick[5];

void *thinkAndEat(int n) {
    printf("philosopher %d is thinking!!\n", n + 1);

    // acquire the chopsticks
    pthread_mutex_lock(&chopstick[n]);
    pthread_mutex_lock(&chopstick[(n + 1) % 5]);

    // eat (it takes 1 second)
    printf("philosopher %d is eating using chopstick[%d] and chopstick[%d]!!\n",
n + 1, n, (n + 1) % 5);
    sleep(1);

    // release the chopsticks
    pthread_mutex_unlock(&chopstick[n]);
    pthread_mutex_unlock(&chopstick[(n + 1) % 5]);

    printf("philosopher %d finished eating!!\n", n + 1);
}

int main() {
    // initiate the mutex locks used for accessing the chopsticks
    for (int i=0; i<5; i++)
        pthread_mutex_init(&chopstick[i], NULL);

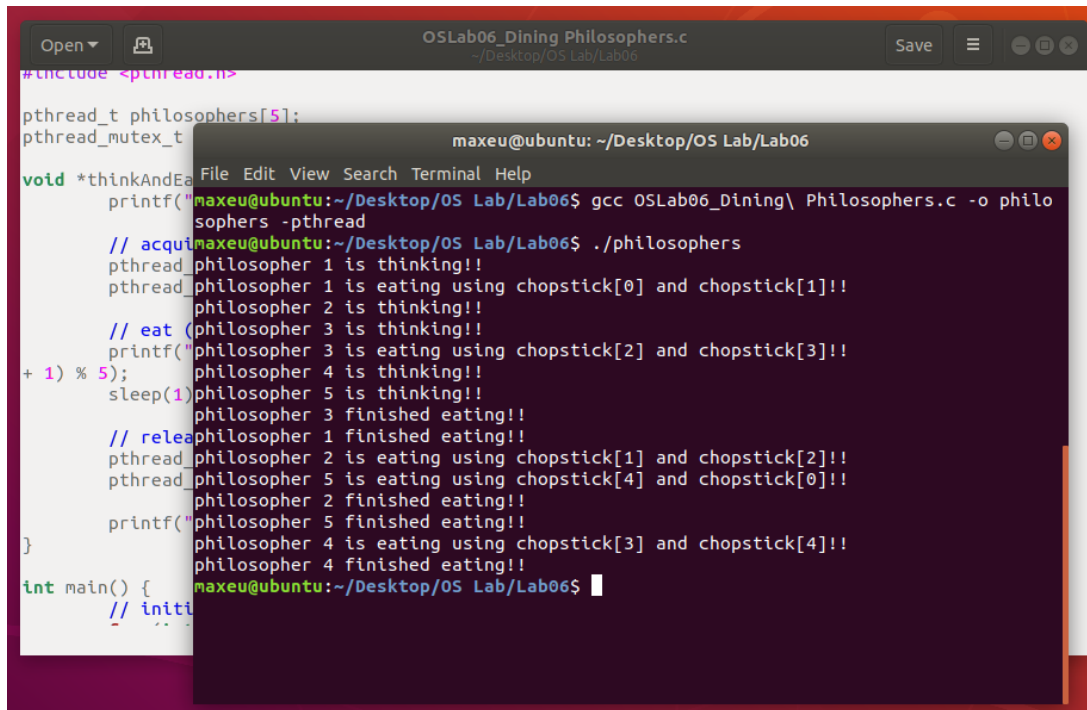
    // create the threads representing the philosophers
    for (int i=0; i<5; i++)
        pthread_create(&philosophers[i], NULL, (void *) thinkAndEat, (void
*)(intptr_t) i);

    // wait for all the threads to finish
    for (int i=0; i<5; i++)
        pthread_join(philosophers[i], NULL);

    return 0;
}

```

خروجی این برنامه به ازای یک بار غذا خوردن فیلسوفان مانند زیر است. اگر بخواهیم این روند همواره ادامه پیدا کند کافیست داخل تابع thinkAndEat خارج از تمام دستورات یک (1) while قرار دهیم.



The image shows a code editor window titled "OSLab06_Dining Philosophers.c" with a file explorer on the left. The code defines a function `thinkAndEat` and a `main` function. A terminal window is overlaid on the code, showing the compilation and execution of the program. The terminal output shows the sequence of actions for five philosophers: thinking, eating, and finishing eating, using chopsticks.

```
#include <pthread.h>

pthread_t philosophers[5];
pthread_mutex_t chopsticks[5];

void *thinkAndEat(void *arg) {
    int philosopher = *(int *)arg;
    printf("philosopher %d is thinking!!\n", philosopher);
    // acquire chopsticks
    pthread_mutex_lock(&chopsticks[philosopher]);
    pthread_mutex_lock(&chopsticks[(philosopher + 1) % 5]);
    // eat
    printf("philosopher %d is eating using chopstick[%d] and chopstick[%d]!!\n", philosopher, philosopher, (philosopher + 1) % 5);
    sleep(1);
    // release chopsticks
    pthread_mutex_unlock(&chopsticks[philosopher]);
    pthread_mutex_unlock(&chopsticks[(philosopher + 1) % 5]);
    printf("philosopher %d finished eating!!\n", philosopher);
}

int main() {
    // initialize philosophers
    for (int i = 0; i < 5; i++) {
        pthread_create(&philosophers[i], NULL, thinkAndEat, &i);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(philosophers[i], NULL);
    }
    return 0;
}
```

maxeu@ubuntu: ~/Desktop/OS Lab/Lab06
maxeu@ubuntu:~/Desktop/OS Lab/Lab06\$ gcc OSLab06_Dining\ Philosophers.c -o philosophers -pthread
maxeu@ubuntu:~/Desktop/OS Lab/Lab06\$./philosophers
philosopher 1 is thinking!!
philosopher 1 is eating using chopstick[0] and chopstick[1]!!
philosopher 2 is thinking!!
philosopher 3 is thinking!!
philosopher 3 is eating using chopstick[2] and chopstick[3]!!
philosopher 4 is thinking!!
philosopher 5 is thinking!!
philosopher 3 finished eating!!
philosopher 1 finished eating!!
philosopher 2 is eating using chopstick[1] and chopstick[2]!!
philosopher 5 is eating using chopstick[4] and chopstick[0]!!
philosopher 2 finished eating!!
philosopher 5 finished eating!!
philosopher 4 is eating using chopstick[3] and chopstick[4]!!
philosopher 4 finished eating!!
maxeu@ubuntu:~/Desktop/OS Lab/Lab06\$