**گزارش دستورکار ششم آزمایشگاه سیستم­های عامل**

نگار موقتیان، 9831062

1. **مساله خوانندگان - نویسندگان**

در این قسمت از آزمایش مطابق دستورکار می­خواهیم دو پردازۀ خواننده و یک پردازۀ نویسنده ایجاد کنیم که بر روی یک خانۀ مشترک از حافظه داده­ای نوشته و یا از روی آن می­خوانند. در برنامۀ نوشته شده یک پدر با دو فرزند داریم که پدر وظیفۀ آپدیت کردن مقدار count بر روی حافظه و فرزندان وظیفۀ خواندن آن را دارند. همچنین شرط خاتمۀ خواندن و نوشتن رسیدن مقدار count به عدد 2 (رقم آخر شماره دانشجویی) در نظر گرفته شده­است.

در ابتدای برنامه کتابخانه­های مورد نیاز اضافه شده­اند.

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

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

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

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

اگر هم داخل پردازه­های فرزند بودیم تا زمانی که به بیشینه مقدار دلخواه count نرسیده­ایم مقدار خانۀ حافظۀ 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);

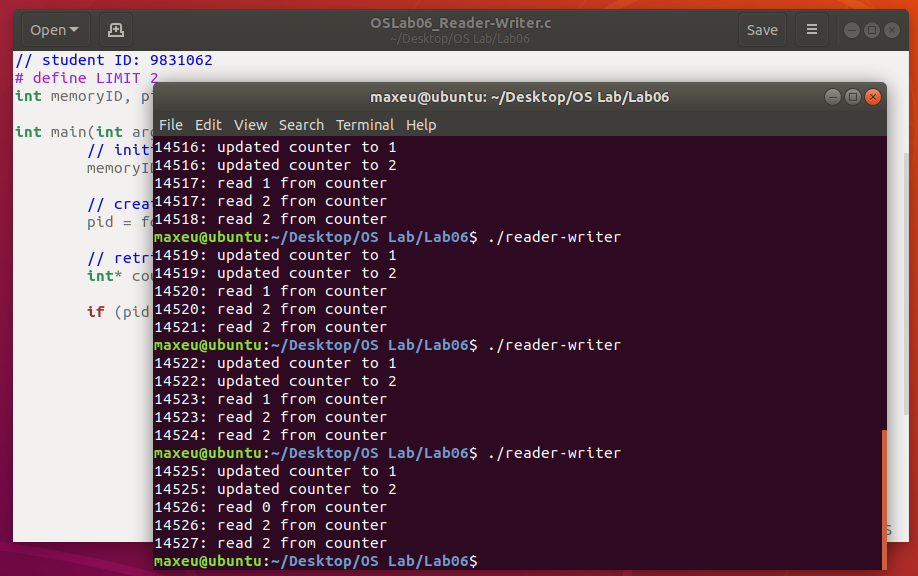
printf("%d: read %d from counter\n", getpid(), tmp);

}

return 0;

}

حال می­توانیم برنامۀ 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.



همانطور که مشاهده می­شود پردازۀ پدر مقدار 2 را بر روی حافظۀ مشترک نوشته اما پردازۀ 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 پردازه­ای بر روی حافظۀ مشترک مقدار نوشته و دیگری آن را می­خواند. بنابراین اگر برای مثال ابتدا پردازۀ فرزند مقدار count را خوانده، context switch انجام شود، پردازۀ پدر مقدار count را دو بار افزایش داده و دوباره نوبت به پردازۀ فرزند برسد تا مقدار count را چاپ کند مقدار صفر چاپ خواهد شد، در حالی که مقدار اصلی count در این لحظه برابر با 2 است.

برای حل این مشکل می­توان از یک mutex lock با قفل مشترک (از نوع pthread\_mutex\_t) میان پردازه­ها استفاده کرد، به این نحو که پیش از هر خواندن و یا نوشتن بر حافظۀ 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 پردازه به طور همزمان بر روی این حافظۀ مشترک کار نمی­کنند و شرایط مسابقه پیش نخواهد آمد.

1. **مسالۀ فیلسوف­های غذاخور**

در این قسمت از آزمایش می­خواهیم مطابق توضیحات دستورکار راه حلی برای مسالۀ فیلسوف­های غذاخور ارائه کنیم. در این برنامه پنج thread داریم که هر کدام نمایانگر یک فیلسوف هستند و 5 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 1 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 اجازۀ 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 while (1) قرار دهیم.

