دانشگاه صنعتی اصفهان

دانشکده مهندسی برق و کامپیوتر

عنوان: تکلیف سوم درس سیستم‌های عامل ۱

نام و نام خانوادگی: علیرضا ابره فروش

شماره دانشجویی: ۹۸۱۶۶۰۳

نیم سال تحصیلی: پاییز ۱۴۰۰

مدرّس: دکتر محمّدرضا حیدرپور

دستیاران آموزشی: مجید فرهادی - دانیال مهرآیین - محمّد نعیمی

فهرست مطالب

۱ سوال اول

۱.۱ آ

Table 1:

	Process	Thread
1.	Process means any program is in execution.	Thread means segment of a process.
2.	Process takes more time to terminate.	Thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	Process is less efficient in term of communication.	Thread is more efficient in term of communication.
6.	Multi programming holds the concepts of multi process.	We don’t need multi programs in action for multiple threads because a single process consists of multiple threads.
7.	Process is isolated.	Threads share memory.
8.	Process is called heavy weight process.	A Thread is lightweight as each thread in a process shares code, data and resources.
9.	Process switching uses interface in operating system.	Thread switching does not require to call a operating system and cause an interrupt to the kernel.
10.	If one process is blocked then it will not effect the execution of other process	Second thread in the same task could not run, while one server thread is blocked.
11.	Process has its own Process Control Block, Stack and Address Space.	Thread has Parents’ PCB, its own Thread Control Block and Stack and common Address space.
12.	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
13.	Changes to the parent process does not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.

۲.۱ ب

- register setِthreadها بین threadها غیر مشترک است. این رجیسترها وضعیت پردازنده را مشخص می‌کنند و threadها عملیات محاسباتی مستقل از یکدیگر دارند بنابراین virtual registerهای نمیتوانند مشترک باشند. مثلا Program Counter(PC) و Stack Pointer(SP) در هر thread مجزا و مخصوص به همان thread است.
- TIDِ هر thread یکتا و مختص همان thread است. در واقع داخل یک پروسس، TID وجه تمییز بین threadهای یک پروسس است. پس نمی‌تواند مشترک باشد.
- stackِ هر thread، متغیرهای محلی و آدرس بازگشتی را به صورت مجزا ذخیره می‌کنند.

۳.۱ ج

برای مدلسازی بسیاری از مسائل موجود در دنیای واقعی که به نحوی چند کار باهم همزمانی دارند به برنامه‌های concurrent نیاز داریم و نمی‌توانیم تنها به متدهای برنامه‌نویسیِ خطی و ترتیبی اتکا کنیم. برنامه‌نویسی چندنخی در این زمینه به ما کمک می‌کند، به طوریکه به هر یک از کارهای همزمان یک thread نظیر می‌کنیم و این تعاملِ threadها باهم روند یک کار را به طور دقیق مدل می‌کند.

۴.۱ د

در پردازنده‌های تک هسته‌ای هنگامی که یک thread منتظر I/O سنکرون است، scheduler، threadهای دیگر را(برای مثال read()) schedule می‌کند و هدر رفت زمان کمتر رخ می‌دهد. از دیگر مزایای برنامه‌نویسی چندنخی در پردازنده‌های تک هسته‌ای عدم جابه‌جایی ترتیب اجرای دستورات در بسیاری از پیاده‌سازی‌ها است. مشکل data race نیز به دلیل عدم وجود پیچیدگی در ترتیب اجرای دستورات، رفع می‌شود.

۵.۱ ه

جججججججججججججججج

۶.۱ و

Ticketlockها بر خلاف Spinlockها مشکل گرسنگی مفرط ندارند. Fetch and Add نیز دچار گرسنگی مفرط نمی‌شود.

۷.۱ ز

۴ منبع و ۳ رشته داریم. درنتیجه حداقل یکی از رشته‌ها حداقل ۲ تا از منابع را اشغال می‌کند و circular wait رخ نمی‌دهد. پس Deadlock نمی‌تواند رخ دهد.

۲ سوال دوم

۱.۲ طول بافر محدود

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

sem_t unocc; //occupied slots
sem_t occ; //unoccupied slots

void* produce(void* arg)
{
        while(1)
        {
                sem_wait(&unocc);
        sleep(rand() % 100 * 0.01);
        sem_post(&occ);
        sleep(rand() % 100 * 0.01);
        }
}

void* consume(void* arg)
{
        while(1)
        {
                sem_wait(&occ);
        sleep(rand() % 100 * 0.01);
        sem_post(&unocc);
        sleep(rand() % 100 * 0.01);
        }
}

int main(int argv, char* argc[])
```

```
{
        int size = 0;
        printf("Enter buffer size: ");
        scanf("%d", &size);
        pthread_t producer, consumer;
        pthread_attr_t a1;
        sem_init(&occ, 0, 0);
        sem_init(&unocc, 0, size);
        pthread_attr_init(&a1);
        pthread_attr_setdetachstate(&a1, PTHREAD_CREATE_JOINABLE);
        if(pthread_create(&producer, &a1, produce, 0))
        {
                printf("Failed to create producer thread!\n");
                exit(-1);
        }
        if(pthread_create(&consumer, &a1, consume, 0))
        {
                printf("Failed to create consumer thread!\n");
                exit(-1);
        }
        pthread_attr_destroy(&a1);
        if(pthread_join(producer, 0))
        {
                printf("Failed to join producer thread!\n");
        }
        if(pthread_join(consumer, 0))
        {
                printf("Failed to join consumer thread!\n");
        }
        pthread_exit(0);

        return 0;
}
```

۲.۲ طول بافر بی‌نهایت

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

sem_t occ; //occupied slots

void* produce(void* arg)
{
        while(1)
        {
        sem_post(&occ);
        sleep(rand() % 100 * 0.01);
        }
}

void* consume(void* arg)
{
        while(1)
        {
                sem_wait(&occ);
        sleep(rand() % 100 * 0.01);
        }
}

int main(int argv, char* argc[])
{
        pthread_t producer, consumer;
        pthread_attr_t a1;
        sem_init(&occ, 0, 0);
        pthread_attr_init(&a1);
        pthread_attr_setdetachstate(&a1, PTHREAD_CREATE_JOINABLE);
        if(pthread_create(&producer, &a1, produce, 0))
```

```
        {
                printf("Failed to create producer thread!\n");
                exit(-1);
        }
        if(pthread_create(&consumer, &a1, consume, 0))
        {
                printf("Failed to create consumer thread!\n");
                exit(-1);
        }
        pthread_attr_destroy(&a1);
        if(pthread_join(producer, 0))
        {
                printf("Failed to join producer thread!\n");
        }
        if(pthread_join(consumer, 0))
        {
                printf("Failed to join consumer thread!\n");
        }
        pthread_exit(0);
        return 0;
}
```

۳ سوال سوم

برای اینکه ثابت کنیم روش ارائه شده مسئله Critical Section را حل می‌کند باید ثابت کنیم که:

- در هر زمان تنها یک پروسس می‌تواند داخل critical section باشد. اگر پروسس‌های دیگری بخواهند به critical section دسترسی پیدا کنند باید صبر کنند تا آزاد شود.(Mutual Exclusion)
- اگر یک پروسس از critical section استفاده نمی‌کند نباید پروسس دیگری را از دسترسی به آن بازدارد. به عبارت دیگر هر پروسسی می‌تواند وارد یک critical section شود اگر آزاد باشد.(Progress)
- هر پروسس الزاما باید یک زمان انتظار محدود داشته باشد و تا ابد برای دسترسی به critical section منتظر نماند.(Bounded Waiting)

حال هر یک از موارد بالا را ثابت می‌کنیم.

- Mutual Exclusion: توجه شود که یک پروسس تنها زمانی که شرط ذیل ارضا شود، وارد critical section می‌شود:
مقدار flagِ هیچ پروسس دیگری برابر in_cs نشود. از آنجایی که پروسس قبل از اینکه وضعیت پروسس‌های دیگر را چک کند مقدار flagِ خود را برابر با in_cs قرار می‌دهد، تضمین می‌شود که هیچ دو پروسسی به طور همزمان وارد critical section نخواهند شد.

- Progress: شرایطی را در نظر بگیرید که چند پروسس به طور همزمان بخواهند مقادیر flagشان را برابر با in_cs قرار دهند و سپس چک کنند که آیا پروسس دیگری flagش را برابر با in_cs قرار داده است یا خیر. وقتی این اتفاق رخ می‌دهد، همه پروسس‌ها متوجه می‌شوند که پروسس‌های رقیب وجود دارند و وارد پیمایش بعدیِ while(1)ِ بیرونی می‌شوند و مقادیر flagشان به want_in برمی‌گردانند. حال تنها پروسسی که مقدارِ turnش را برابر با in_cs قرار می‌دهد، پروسسی است که نزدیک ترین index به turn را دارد. هرچند ممکن است که پروسس‌هایی جدید با indexهای حتی نزدیک‌تر به turn وجود داشته باشند که تصمیم بگیرند در این نقطه وارد critical section شوند ونتیجتا قادر باشند به شکل همزمان مقدار flagِ خود را برابر با in_cs قرار بدهند. این پروسس‌ها درنهایت متوجه می‌شوند که پروسس‌های رقیبی وجود دارند که ممکن است پروسسی که درحال وارد شدن به critical section است را بازنشانی کند. هرچند در هر پیمایش، مقادیر indexِ پروسس‌هایی که مقادیر flagِ خود را برابر با in_cs قرار داده‌اند، به turn نزدیک‌تر شده و در نهایت به شرط ذیل می‌رسند:
تنها یک پروسس flagش را برابر با in_cs قرار می‌دهد و هیچ پروسس دیگری که indexش بین این پروسس و turn قرار دارد، flagش را برابر با in_cs قرار نمی‌دهد. پس این تنها پروسسی است که وارد critical section می‌شود.

- Bounded Waiting: این شرط به این دلیل ارضا می‌شود که هرگاه پروسسِ k بخواهد وارد critical section شود flagش دیگر برابر با idle نمی‌باشد. در نتیجه، هر پروسسی که indexش بین k و turn قرار نداشته باشد نمی‌تواند وارد critical section بشود. در این حال، همه پروسس‌هایی که indexشان بین k و turn قرار دارد و مایلند که وارد critical section شوند(باتوجه به اینکه سیستم همواره پیشروی می‌کند)، فورا وارد critical section می‌شوند و مقدار turn به طور یکنواخت به k نزدیک می‌شود. درنهایت چه turn با k برابر شود و چه هیچ پروسسی وجود نداشته باشد که indexش بین k و turn قرار داشته باشد، پروسسِ k وارد critical section می‌شود.

۴ سوال چهارم

۵ سوال پنجم

منابع

[1] https://gateoverflow.in/150841/Tlb-and-page-fault