بسمه تعالی

تمرین شماره ۴

سیستم‌های عامل

محسن کربلائی امینی، ۹۸۲۴۲۱۲۸

آبان ۱۴۰۲

سوال ۲:

- **Arbitrator:**

در این راه‌حل، یک نقش سوم برای پایش مصرف منابع به نام waiter یا گارسون اضافه می‌کنیم.
اصول این راه‌حل به این شکل است:

۱. وقتی یک فیلسوف می‌خواهد غذا بخورد، ابتدا از گارسون اجازه می‌گیرد.
۲. گارسون در یک زمان تنها به یک درخواست اجازه می‌دهد.
۳. فقط فیلسوفی که اجازه خوردن دارد چنگال‌ها را برمی‌دارد.
٤. رها کردن یک منبع نیازی به اجازه ندارد اما به گارسون اطلاع می‌دهند که منبع آزاد است.

در این راه‌حل، احتمال بوجود آمدن starvation وجود دارد. برای حل این مشکل باید در طراحی arbitrator یا همان گارسون مرکزی باید این مورد لحاظ شود تا این مشکل بوجود نیاید.

- Chandy/Misra:

اصول این راه‌حل:

۱. چنگال‌ها می‌توانند کثیف و یا تمیز باشند. مقدار دهی اولیه چنگال‌ها به کثیف خواهد بود.
۲. چنگال‌ها با id پایین‌تر به فیلسوف‌ها تخصیص داده می‌شوند.
۳. وقتی یک فیلسوف میخواهد غذا بخورد، باید چنگال کناری را از همسایه‌اش بگیرد. فیلسوف برای این کار به همسایه‌اش پیام می‌دهد.
٤. وقتی یک فیلسوف یک درخواست دریافت می‌کند:
   a. اگر چنگال تمیز باشد، نگه‌ش می‌دارد.
   b. اگر کثیف باشد، فیلسوف آن‌را تمیز می‌کند و آن را در اختیار همسایه‌اش قرار می‌دهد.
٥. بعد از اینکه یک فیلسوف از آن چنگال استفاده کرد، چنگال کثیف می‌شود و اگر همسایه‌ای درخواست چنگال کرده باشد، آن را تمیز می‌کند و در اختیار وی قرار می‌دهد.

این راه‌حل خوبی‌های زیادی دارد و از جمله آن‌ها حل شدن مسئله starvation می‌باشد.
مکانیزم تمیز و کثیف بودن یک راهی برای اعطای اولویت گرسنه‌ترین فیلسوف است.

سوال ۳:

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
            critical section
    flag[i] = false;
            remainder section
 } while (true);
```

این راه‌حل، شرط progress، به این معنا که اگر هیچ پراسسی در ناحیه بحرانی خودش نباشد، و پراسسی باشد که بخواهد وارد ناحیه بحرانی خود شود نباید به صورت نامتناهی به تعویق بیوفتد، را نقض می‌کند. تصور کنید پراسس pi شروع به اجرا می‌کند، نوبت را به j می‌دهد و منتظر می‌ماند تا j وارد ناحیه بحرانی خود شود. اما اگر پراسس pj هیچوقت اجرا نشود، pi باید به صورت نامتناهی منتظر بماند.

سوال ۴:

- Mutual Exclusion: این شرط در این قطعه کد رعایت می‌شود. یعنی هیچ دو پراسسی همزمان در ناحیه بحرانی خود قرار نمی‌گیرند. در ابتدا با استفاده از آرایه flag، هر پراسس آمادگی خود را برای ورود به ناحیه بحرانی اعلام می‌دارد.اگر تنها یک پراسس آماده ورود به ناحیه بحرانی باشد، مشکلی ندارد و پس از آن نوبت را به پراسس دیگر می‌دهد. اما اگر هر دو بخواهند وارد ناحیه بحرانی شوند بر اساس مقدار دهی اولیه turn و i وj، یکی از آنها وارد می‌شود و نوبت را به دیگری می‌دهد.
- Progress: این شرط هم صادق است. اگر یک پراسس بخواهد وارد ناحیه بحرانی خود شود، به صورت نامشخصی به تعویق نخواهد افتاد. در exit section هر پراسس، نوبت به پراسس دیگر واگذار می‌شود. در صورتی که پراسس دیگر نخواهد اجرا شود، مقدار flag مربوط به آن false خواهد بود، بنابراین اگر پراسسی که نوبت به آن assign نشده، بخواهد وارد ناحیه بحرانی شود، به تعویق نمی‌افتد. همچنین اگر پراسس دیگر بخواهد وارد ناحیه بحرانی خود شود، در exit section آن نوبت را پس می‌دهد و مقدار flag مربوط به خود را false می‌کند تا پراسس منتظر بعد از آن اجرا شود.

- Bounded Waiting: این شرط نیز صادق است. هنگامی که یک پراسس بخواهد وارد ناحیه بحرانی خود شود و قرار بر تعویق آن باشد، برای تعداد ورود باقی پراسس‌ها به ناحیه بحرانی محدودیتی وجود دارد که برابر است با ۱. یعنی در هر صورت اگر دو پراسس همزمان بخواهند که وارد ناحیه بحرانی خود شوند، به صورت نوبتی وارد ناحیه بحرانی خود می‌شوند.

سوال ۵:

```
typedef struct barrier}
   semaphore sem;
   semaphore mutex;
   int *N;
   int threadCounter;
{Barrier;

void init (Barrier *b , int M, int Ns} ([]
   b->sem=;۰
   b->mutex=;۱
   b->N = Ns;
   b->threadCounter=۰
{

void barrier_point (Barrier *b, int group} (
   wait(b->mutex(
   threadCounter;++
   signal(b->mutex(

      wait(b-> sem(
{
```

سوال ۶:

Shortest remaining time first:

گانت چارت:

P۱	CS	P۰	CS	P۳	CS	P۱	CS	P۲

۴ ۵ ۸ ۹ ۱۳ ۱۴ ۱۹ ۲۰ ۲۹

$$Average\ Waiting\ Time = \frac{[(۱۴-۴)+(۲۰-۱)+(۵-۴)+(۹-۷)]}{۴} = \frac{۳۲}{۴} = ۸$$

سوال ۷:

Round Robin: Q=۲

گانت چارت:

۲	۴	۶	۸	۱۰	۱۲	۱۳	۱۵	۱۷	۱۹
P۰	P۱	P۲	P۴	P۳	P۰	P۱	P۲	P۴	P۳

۲۱	۲۲	۲۳
P۲	P۴	P۲

Highest Ratio First: Q=۲,

Ci= burst time, Wi= Wait time

$$Ratio = \frac{Wi+Ci}{Ci} \qquad R_{t=۷}p۲ = \frac{۳}{۷} + ۱, R_{t=۷}p۳ = ۱, R_{t=۷}p۴ = ۱ + \frac{۲}{۵}$$

$$R_{t=۱۴}p۳ = ۱ + \frac{۷}{۴}, R_{t=۱۴}p۴ = ۱ + \frac{۹}{۵}$$

گانت چارت:

۴	۷	۱۴	۱۹	۲۳
P۰	P۱	P۲	P۴	P۳

روش HRN تنها به تعداد ۴ بار context switch خواهد داشت در صورتیکه در روش RR این عدد به ۱۲ می‌رسد. بنابراین روش HRN از نظر زمان اجرای همه پراسس‌ها جلوتر از روش RR خواهد بود. اما از نظر احساس اجرای موازی از RR ضعیف‌تر خواهد بود. همچنین از نظر میانگین زمان انتظار:

$$Average\ Waiting\ Time(RR)$$

$$= \frac{[(۱۰ - ۲) + (۱۲ - ۴) + (۱۳ - ۶ + ۲۲ - ۱۵) + (۱ + ۷ + ۳) + (۱ + ۷)]}{۵}$$

$$= \frac{۸ + ۸ + ۱۴ + ۱۲ + ۸}{۵} = \frac{۵۰}{۵} = ۱۰$$

$$Average\ Waiting\ Time(HRN) = \frac{[(۰) + (۲) + (۳) + (۹) + (۱۲)]}{۵} = \frac{۲۶}{۵}$$

سوال ۸:

۱. $a < b > ۰$

این الگوریتم می‌تواند EDF باشد. به این صورت که در گذر زمان، با در دست داشتن CPU به ددلاین خود نزدیک می‌شود اما در عین حال با پردازش، زمان مورد نیاز برای پردازشش نیز کاهش می‌یابد. در صورت در اختیار نداشتن CPU، پراسس همواره به ددلاین خود نزدیک‌تر می‌شود و اولویت آن افزایش می‌یابد. و این افزایش اولویت بیشتر از حالت اول می‌باشد.

۲. $b < a > ۰$

این الگوریتم می‌تواند SRTF باشد. در طی زمان و در اختیار داشتن CPU به دلیل کاهش زمان مورد نیاز برای پردازش پراسس، اولویت آن بالا می‌رود. در صورت در اختیار نداشتن پردازنده، با انجام‌شدن دیگر پراسس‌ها، اولویت پراسس idle افزایش می‌یابد اما این افزایش به اندازه حالت اول نمی‌باشد.

۳. $b < ۰ < a$

تنها حالتی که این مورد بتواند پیش بیاید یک الگوریتم Multilevel queue می‌باشد که در حالتی که پراسس‌ها در انتظار مانند، آن‌ها را به لایه‌ای با اولویت کمتر منتقل کند و در صورت پردازش اولویت آن‌ها را افزایش دهد.