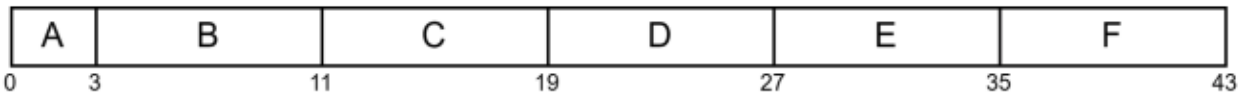
پاسخنامه تمرین چهارم - سیستم‌های عامل دکتر جوادی - بهار ۱۴۰۲

---

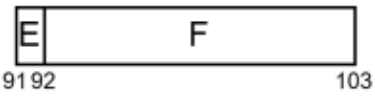
۱. یک سیستم تک پردازندهاي با صف بازخورد چند سطحی)Multi-level Feedback Queue(را درنظر بگیرید. به صف اول الگوریتم چرخشی با برش زمانی معادل 8 میکروثانیه داده شده است. به سطح دوم الگوریتم جرخشی با برش زمانی معادل 16 میکروثانیه و سطح سوم به ترتیب ورود)FCFS(زمانبندي شده است. فرض کنید 6 کار همگی در زمان صفر به سیستم وارد میشوند و زمان اجراي آنها به ترتیب 3، 8، 12، 20، 25، 35 میکروثانیه است. متوسط Turnaround Time کارهاي فوق در این سیستم چقدر خواهد بود؟ به طور کامل توضیح دهید.

در یک سیستم با صف بازخورد چند سطحی، صف‌های ابتدایی اولویت بالاتری نسبت به صف‌های پایین‌تر دارند. وقتی یک پردازه برای زمانبندی وارد سیستم می‌شود ابتدا وارد صف با بیشترین اولیت شده و اگر کار او بعد از اختصاص یک مرحله پردازنده پایان نیابد به یک صف پایین‌تر منتقل میشود. این اتفاق تکرار می‌شود تا در نهایت پردازه‌ی مورد نظر پایان یابد. در این سیستم نیز ابتدا پردازه‌ها وارد صف اول شده و به اندازه‌ی نهایتا ۸ میکروثانیه پردازنده به آنها تعلق می‌گیرد. پردازه‌های A و B پایان می‌یابند اما باقی پردازه‌ها به صف پایین‌تر منتقل می‌شوند.



در این صف نیز مشابه صف قبل به پردازه‌ها نهایتا ۱۶ میکروثانیه پردازنده تعلق می‌گیرد. پردازه‌ی C و D به اتمام می‌رسند.

پردازه‌های باقیمانده در صف نهایی به صورت FCFS به صورت زیر سرویس داده می‌شوند.



متوسط زمان رفت و برگشت:

average turnaround time = $\frac{3+11+47+59+92+103}{6} = \ 52.5\mu s$

---

۲. جدول پردازه‌های زیر را داریم:

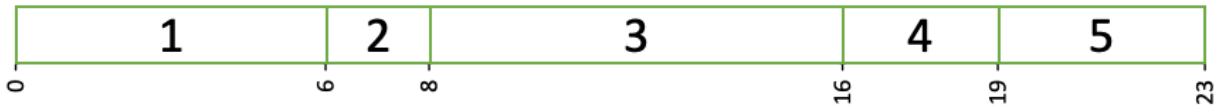
Process	Arrival Time	Burst Time
P1	0	6
P2	1	2
P3	2	8
P4	3	3
P5	4	4

با استفاده از الگوريتم‌هاي زمانبندي زير، اين پردازه‌ها را زمانبندي كنيد (نمودار Grantt هر زمانبندي را رسم كنيد):

- First Come First Serve Non-Preemptive
- Shortest Job First Non-Preemptive
- Shortest Remaining Job First Preemptive
- Round Robin with Quantum = 1 and Context Switch = 0.5
- Round Robin with Quantum = 4 and Context Switch = 0.5

الف) براي هر الگوريتم، ميانگين زمان انتظار، Turnaround Time و CPU Utilization را محاسبه كنيد. عملكرد هر الگوريتم و اينكه كدام يک را براي اين مجموعه از پردازه‌ها توصيه می كنيد، مقايسه و بحث كنيد.

الگوریتم FCFS



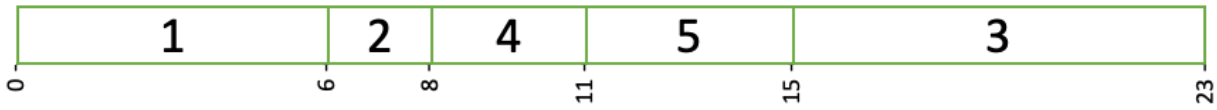
Turn around time , AVG = 46.8

P1	P2	P3	P4	P5
6	7	14	16	19

CPU utilization = 100%

برای همه‌ی وظایف ۱۰۰ درصد است.

الگوریتم SJF

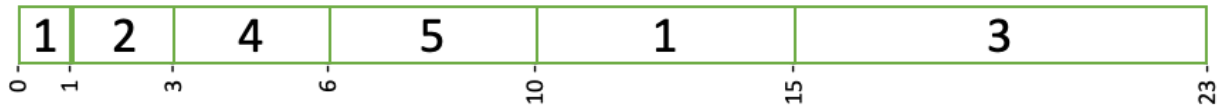


Turn around time , AVG = 44.2

P1	P2	P3	P4	P5
6	7	21	8	11

CPU utilization = 100%

الگوریتم SRJF

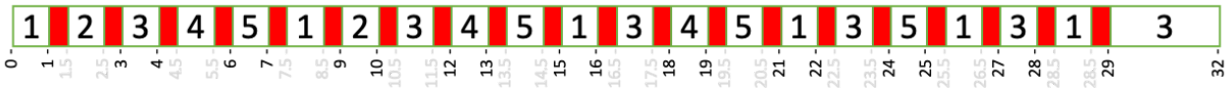


Turn around time , AVG = 42.2

P1	P2	P3	P4	P5
15	2	21	3	6

CPU utilization = 100%

الگوریتم RR - Q1



Turn around time , AVG = 87.7

P1	P2	P3	P4	P5
28.5	9	30	16	21

CPU utilization = (23 / 32) * 100 = 71.8%

الگوریتم RR - Q4

Turn around time , AVG = 63.5

P1	P2	P3	P4	P5
20.5	5.5	23	11.5	15

CPU utilization = (23 / 25) * 100 = 92%

مقایسه:

مقایسه‌ی بین الگوریتم‌ها بستگی به این دارد که چه معیاری برای ما مدنظر باشد. با توجه به معیار Turnaround time الگوریتم SRJF بهترین عملکرد را دارد و برای این تسک ست مناسب است. این الگوریتم و باقی الگوریتم‌های SJF و FSCF از نظر CPU Utilization عملکرد یکسانی دارند و انتخاب هر کدام از آن‌ها برای تسک ست منطقی است. در مقایسه‌ی این الگوریتم‌ها، RR زمان turnaround بدتری دارد اما همه‌ی تسک‌ها را به صورت همزمان جلو می‌برد و response time بهتری دارد. مقایسه‌ی این تسک‌ست‌ها از نظر utilization منطقی نیست چرا که تنها در دو الگوریتم RR مقدار زمان context switch در نظر گرفته شده‌است و در صورتی که در باقی الگوریتم‌ها هم این زمان را در نظر می‌گرفتیم utilization کمتر از ۱۰۰ می‌بود.

به طور کلی از آنجایی که تسک‌ها اولویت و ددلاین خاصی ندارند، انتخاب SRJF برای داشتن بهترین زمان رفت و برگشت و انتخاب FCFS برای بهترین میزان بهره‌وری توصیه می‌شود (در این الگوریتم preeemtion نداریم). SJF می‌تواند هر دوی این معیارها را برآورده کند گرچه زمان رفت و برگشت آن کمی از SRJF بدتر است.

ب) در الگوریتم چرخشی اگر اندازه کوانتوم زمانی نزدیک به تعویض پردازه باشد چه اتفاقی می‌افتد؟ اگر از تعویض پردازه خیلی بزرگتر باشد چه اتفاقی می‌افتد؟ کوانتوم زمانی بهینه چه نسبتی با تعویض پردازه باید داشته باشد؟ در خصوص پاسخ خود به دو زمانبندی آخر به سوالات پاسخ دهید.

به طور کلی اگر اندازه کوانتوم زمانی کوچک و نزدیک به تعویض پردازه باشد، آن‌گاه تعداد تعویض پردازه‌ها بسیار زیاد می‌شود و باعث می‌شود که context switch زیادی داشته باشیم. این مورد باعث کاهش بهره‌وری و افزایش سربار زمان‌بندی می‌شود و به هیچ وجه توصیه نمی‌شود. در صورتی که از تعویض پردازه خیلی بزرگتر باشد، عملکرد الگوریتم چرخشی به FCFS شباهت پیدا می‌کند. به طور کلی q باید از زمان تعویض بزرگتر باشد اما اگر خیلی بزرگ در نظر گرفته شود، مزیت‌هایی که RR برایمان فراهم می‌کند (زمان پاسخگویی پایین) را از دست می‌دهیم. در حالتی که زمان تعویض کمتر از ۱۰ میکروثانیه باشد، کوانتوم زمانی بین ۱۰ میلی‌ثانیه تا ۱۰۰ میلی‌ثانیه در نظر گرفته می‌شود، چیزی حدود ۱۰۰۰ برابر. (در عین حال از زمان burst در ۸۰ درصد از پردازه‌ها کوچکتر باشد)

---

۳. جفت پردازه‌هاي زير متغير شمارنده را به اشتراك می‌گذارند که قبل از شروع اجراي هر يک از پردازه‌ها مقدار اوليه 10 داده شده است:

```
Process A                    Process B
…                            …
A1: LD(counter,R0)           B1: LD(counter,R0)
    ADDC(R0,1,R0)                ADDC(R0,2,R0)
A2: ST(R0,counter)           B2: ST(R0,counter)
…                            …
```

الف) اگر پردازه‌هاي A و B بر روي يک سيستم اشتراك زمانی اجرا شوند، شش ترتيب وجود دارد که در آن دستورات LD و ST ممکن است اجرا شوند. براي هر يک از ترتيب اجرا، مقدار نهايی متغير شمارنده را بدست آوريد.

1. A1 A2 B1 B2 => counter = 13
2. A1 B1 A2 B2 => counter = 12
3. A1 B1 B2 A2 => counter = 11

4. B1 A1 B2 A2 => counter = 11
5. B1 A1 A2 B2 => counter = 12
6. B1 B2 A1 A3 => counter = 13

در دو سوال زیر از شما خواسته می‌شود که برنامه هاي اصلی را براي پردازه‌هاي A و B با اضافه کردن حداقل تعداد سمافورها و Signal & Wait تغییر دهید تا تضمین شود که نتیجه نهایی اجراي دو پردازه یک مقدار مشخص براي شمارنده خواهد بود. براي هر سمافوري که معرفی می کنید، مقادیر اولیه را مشخص کنید.

ب) سمافور اضافه کنید تا مقدار نهایی شمارنده 12 شود.

```
Semaphores: X=0, Y=0

Process A                          Process B
…                                  …
A1: LD(counter,R0)                 B1: LD(counter,R0)

    ADDC(R0,1,R0)                      ADDC(R0,2,R0)
    wait(x)                        signal(x); wait(y)
A2: ST(R0,counter)                 B2: ST(R0,counter)
…   signal(y)                      …
```

ج) سمافور اضافه کنید تا مقدار نهایی شمارنده 13 نشود.

```
Semaphores: X=0, Y=0

Process A                          Process B
…                                  …
A1: LD(counter,R0)                 B1: LD(counter,R0)
    signal(x)                          signal(y)
    ADDC(R0,1,R0)                      ADDC(R0,2,R0)
    wait(y)                            wait(x)
A2: ST(R0,counter)                 B2: ST(R0,counter)
…                                  …
```

۴. P1 و P2 پردازه‌هایی هستند که همزمان اجرا می شوند. P1 داراي دو بخش از کد است که بخش A توسط بخش B دنبال میشود. به طور مشابه، P2 داراي دو بخش است: C و سپس D. در هر پردازه اجراي به صورت متوالی پیش میرود، بنابراین ما تضمین می کنیم که A ≤ B، یعنی A قبل از B باشد. به طور مشابه. ما میدانیم که C ≤ D. هیچ حلقهاي در پردازه ها وجود ندارد. هر پردازه دقیقا یک بار اجرا می‌شود.

از شما خواسته می‌شود که سمافورها را به برنامه ها اضافه کنید - ممکن است لازم باشد از بیش از یک سمافور استفاده کنید. مقادیر اولیه هر سمافوري که استفاده می‌کنید را مشخص کنید. براي نمره کامل از حداقل تعداد سمافورها استفاده کنید و هیچ گونه محدودیت اولویت غیر ضروري را معرفی نکنید.

Semaphore initial values: _____

Process P1	Process P2
…Section A code…	… Section C code…
…Section B code…	…Section D code…

الف) دستورات WAIT(...) و SIGNAL(...) را به پردازه‌های زیر اضافه کنید تا محدودیت اولویت B≤ C برآورده شود، یعنی اجرای P1 قبل از شروع اجراي P2 به پایان برسد.

Semaphore initial values: S=0

Process P1	Process P2
	wait(S)
…Section A code…	… Section C code…
…Section B code… signal(S)	…Section D code…

ب) دستورات WAIT(...) و SIGNAL(...) را به پردازه‌هاي زير اضافه تا A ≤ D يا C ≤ B، به عنوان مثال، اجراي P1 و P2 نتوانند همپوشانی داشته باشند، اما اجازه داده شود به هر ترتيب انجام شوند.

Semaphore initial values: M=1

Process P1	Process P2
wait(M) …Section A code… …Section B code… signal(M)	wait(M) … Section C code… …Section D code… signal(M)

ج) دستورات WAIT(...) و SIGNAL(...) را به پردازه‌هاي زير اضافه کنيد تا A ≤ D و C ≤ B، يعنی بخش اول (A و C) هر دو پردازه اجرا خود را قبل از اينکه بخش دوم (B و D) اجراي خود را آغاز کند، کامل کنند.

Semaphore initial values: S=0, T=0

Process P1	Process P2
…Section A code… signal(S) wait(T) …Section B code…	… Section C code… signal(T) wait(S) …Section D code…

۵. شروط انحصار متقابل، پیشرفت و انتظار محدود را براي الگوريتم‌هاي زير بررسی کرده و دلیل خود را بنویسید.

الف)

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

انحصار متقابل: وجود دارد، فرآیندها برای ورود به بخش بحرانی نیاز دارند که نوبت به آنها تعارف شود. فرض می‌کنیم فرآیند i زودتر به حلقه while رسیده است. پس از آنکه فرآیند j نوبت را به فرآیند i تعارف کرد، فرآیند i وارد بخش بحرانی می‌شود. اما فرآیند i دیگر نمی‌تواند نوبت را به فرآیند j تعارف کند. پس فرآیند j نمی‌تواند وارد شود و انحصار متقابل حفظ می‌شود.

پیشرفت: به دلیل شرط !flag[j] اگر فقط یکی از فرآیندها اجرا شود، پیشرفت نخواهیم داشت اما اگر هر دو اجرا شوند و در قسمت remainder دچار قفل نشوند، پیشرفت وجود خواهد داشت.

انتظار محدود: وجود دارد، زیرا فرآیندی که از بخش بحرانی خارج شود، ناچار است دو مرتبه نوبت را به فرآیند دیگر تعارف کند. البته این مورد نیز به شرط آن است که قسمت remainder در زمان متناهی به اتمام برسد.

ب)

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

انحصار متقابل: وجود دارد، فرآیندی که نوبت را به فرآیند دیگر بدهد خود نخواهد توانست وارد بخش بحرانی شود تا هنگامی که فرآیند دیگر از بخش بحرانی خارج شود.

پیشرفت: وجود دارد، زیرا حالتی وجود ندارد که هر دو فرآیند قادر به عبور از حلقه while نباشند.

انتظار محدود: وجود دارد، چون به دلیل تعارف نوبت امکان ندارد که یک فرآیند برای همیشه پشت حلقه while باقی بماند.

راه حل ساده‌تر: این الگوریتم همان الگوریتم پترسون است زیرا تنها اندیس‌های i و j در همه کاربردهای متغیر flag با هم جا به جا شده‌اند. پس همه شروط برقرار هستند.

---

۶. الف) بدون استفاده از قفل و تنها با استفاده از دستورات compate-and-swap تابع زیر را به گونه‌اي کامل کنید که به صورت اتمی عملیات جمع را انجام دهد. منظور از عملیات جمع اضافه شدن مقدار v به حافظه‌اي است که p به آن اشاره دارد. سپس توضیح دهید تضمینی براي انجام شدن این عملیات وجود دارد یا خیر.

```
int add(int *p, int v)
{
	// TODO
	return *p + v;
}
```

پیاده‌سازی compare-and-swap را به صورت زیر در نظر بگیرید:

```
bool compare_and_swap(int *p, int old, int new)
{
	if(*p != old)
		return false;
	*p = new;
	return true;
}
```

```
bool compare_and_swap(int *p, int old, int new)
{
		if(*p != old)
			return false;
		*p = new;
		return true;
}


int add(int *p, int v)
{
		bool done = false;
		int value;
		while(!done) {
			value = *p;
			done = compare_and_swap(p, value, value + v);
		}
		return *p + v;
}
```

تضمینی برای انجام این عملیات وجود ندارد. به عبارت دیگر شرط انتظار محدود در اینجا برقرار نمی‌باشد. زیرا ممکن است همیشه فرآیندهای دیگر به طور همزمان در حال تغییر مقدار خانه p باشند و در اینصورت هیچگاه فرصت انجام عملیات جمع به صورت اتمی فراهم نخواهد شد.

ب) می‌دانید که براي پیاده‌سازي توابع accuire و release در قفل mutex باید از دستورات سخت افزاري اتمی استفاده کرد. این کار را استفاده از دستور test-and-set انجام دهید.

```
bool lock = false;
bool available = false;

acquire() {
        while (test_and_set(&lock));
        while (!available);
        available = false;
        lock = false;
}

release() {
        while (test_and_set(&lock));
        available = true;
        lock = false;
}
```

به این روش توابع acquire و release به صورت اتمی اجرا خواهند شد.

از آنجایی که این دو تابع تنها توابعی هستند که به متغیر available دسترسی دارند، بنابراین عملیات خواندن و نوشتن هیچگاه به طور همزمان در این متغیر انجام نخواهد شد.

برای سادگی میتوان دستور acquire را به صورت زیر نیز پیاده‌سازی کرد:

```
acquire() {
        while (!test_and_set(&available));
}
```