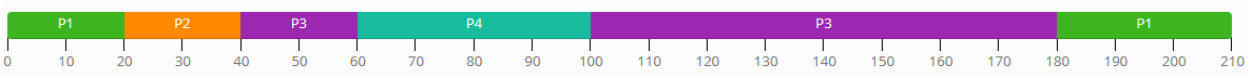
پاسخنامه تمرین سوم سیستم‌های عامل

سوال ۱)

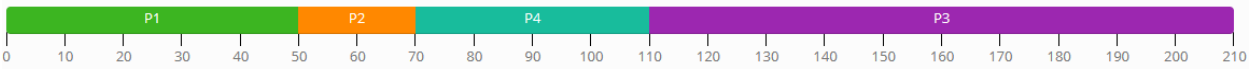
نمودار زمان‌بندی پردازنده برای پردازه‌های ذکر شده به شکل زیر خواهد بود:

Priority:

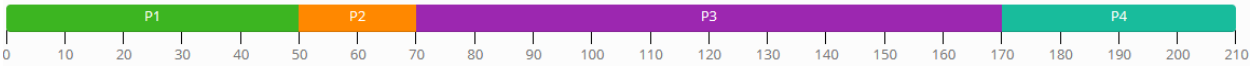


(در صورت non-preemptive بودن، پردازه‌های P1، P3، P4 و P2 به صورت متوالی و پشت سرهم تماما اجرا می‌شوند.)

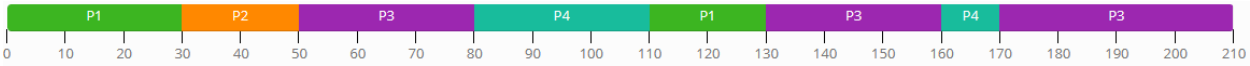
SJF:



FCFS:



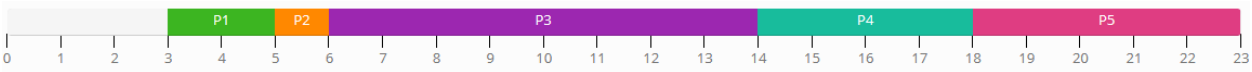
RR:



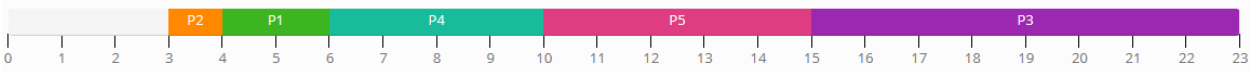
سوال ۲)

زمان‌بندی پردازنده برای هر الگوریتم مطابق موارد زیر خواهد بود:

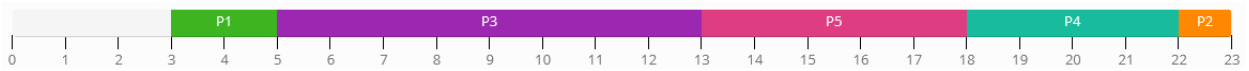
FCFS:



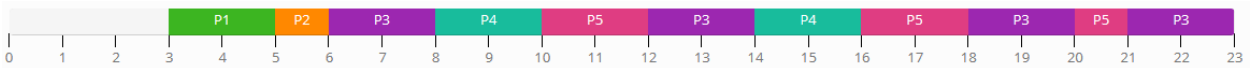
SJF:



Non-preemptive priority:



RR:



با توجه به زمان‌بندی‌ها فوق، به سوالات پاسخ می‌دهیم:

الف) مقدار Turnaround time هر پردازه:

	FCFS	SJF	Non-preemptive priority	RR
P1	2	3	3	2
P2	3	1	1	3
P3	11	20	20	20
P4	15	7	7	13
P5	20	12	12	18

ب) مقدارwaiting time هر پردازه:

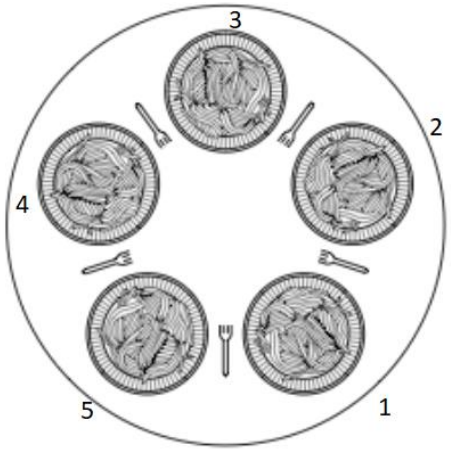
	FCFS	SJF	Non-preemptive priority	RR
P1	0	1	1	0
P2	2	0	0	2
P3	3	12	12	12
P4	11	3	3	9
P5	15	7	7	13
Average	6.2	4.6	4.6	7.2

ج) مطابق جدول قبل، زمان انتظار کمینه متعلق به الگوریتم SJF (و در اینجا Non-preemptive priority) می‌باشد.

سوال ۳) یک پیاده‌سازی بسیار ساده از wait و signal برای سمافور در زیر مشاهده می‌شود:

```
wait(Semaphore s){
   while(s<=0);
   s.value = s.value - 1;
}
signal(Semaphore s){
   s.value = s.value + 1;
}
```

اگر توابع فوق به صورت اتمی اجرا نشوند، امکان قبضه شدن (preempt) شدن در هر نقطه از مراحل هرکدام از توابع وجود دارد، بنابراین اگر یک سمافور دارای مقدار ۱ باشند و wait برای آن توسط دو پردازه به صورت همزمان اجرا شود، امکان دارد که هردو به صورت همزمان اقدام به کم‌کردن مقدار s.value کنند، که باعث نقض قاعده‌ی انحصار متقابل می‌شود.



سوال ۴) روش ذکر شده در این سوال، نوعی از روش Arbitrator solution برای حل مساله‌ی فیلسوف‌های خورنده می‌باشد. این روش با فرض اینکه تابع take_forks با استفاده از روش‌های قفل از race condition جلوگیری می‌کند، مشکل بن‌بست (Dead lock) را حل می‌کند؛ ولی مشکل قحطی همچنان پابرجاست.

الف) همانطور که ذکر شد، مشکل Dead lock وجود ندارد.

ب) بله، به دلیل به ترتیب گذاشتن چنگال‌ها امکان قحطی وجود دارد؛ یک مثال این حالت در ادامه بیان شده است:

فیلسوف‌های ۱ و ۳ در حال خوردن هستند، و فیلسوف ۲ منتظر خوردن است. حال ۳ چنگال‌هایش را می‌گذارد، ولی ۲ همچنان نمی‌تواند شروع به خوردن کند (چرا که ۱ نیز باید چنگال‌هایش را بگذارد). سپس قبل از اینکه ۱ چنگالش را بگذارد، ۳ دوباره شروع به خوردن می‌کند. این بار با اتمام خوردن ۱، فیلسوف ۲ باید منتظر ۳ بماند. اگر این چرخه تکرار شود، همواره فیلسوف ۲ از خوردن باز می‌ماند.

سوال 5) یک راه حل ساده برای این مساله، می‌تواند به شکل زیر باشد:

```
line_1(){
   while (true) {
      wait(mutex);
      serve_bread();
      signal(mutex);
   }
}

line_2(){
```

```
  while (true) {
    wait(mutex);
    serve_bread();
    signal(mutex);
  }
}
```

سوال 6)

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

این روش همانطور که مشخص است، مشابه روش Peterson می‌باشد ولی شرط عبور از قفل آن نسبت به Peterson سخت‌گیرانه‌تر است. بنابراین این روش شروط انحصار مقابل و انتظار محدود را ارضا می‌کند، ولی شرط پیشرفت محقق نشده و مشکل انتظار مشغول نیز وجود دارد.

سوال 7) انتظار مشغول تکنیکی است که در آن یک فرایند، مکرراً یک شرط خاص را بررسی می‌کند تا ببیند آیا آن شرط برقرار است یا خیر؛ مثلا یک حلقه while بدون اینکه قطعه کدی را اجرا کند، فقط در انتظار باطل شدن شرط تکرار برای خروج است. وجود این نوع انتظار باعث می‌شود که یک پردازه بدون اینکه کار مفیدی انجام دهد، منابع اجرایی سیستم را در اختیار داشته باشد و در ظاهر مشغول به کار باشد، در حالی که کاری انجام نمی‌شود.

علاوه بر انتظار مشغول، انتظار محدود (Bounded wait)، انتظار چرخش (Spin wait)، انتظار مسدود شده (blocked wait یا Sleep wait) و انتظارهایی مانند انتظار برای I/O و انتظار پردازه برای شروع به اجرا (تغییر وضعیت از ready به run) وجود دارد.

مشکل انتظار مشغول از طریق استفاده از sleep یا Block کردن یک پردازه قابل جلوگیری است، البته این روش‌ها نیز برای پردازه سربار خودشان را دارند.