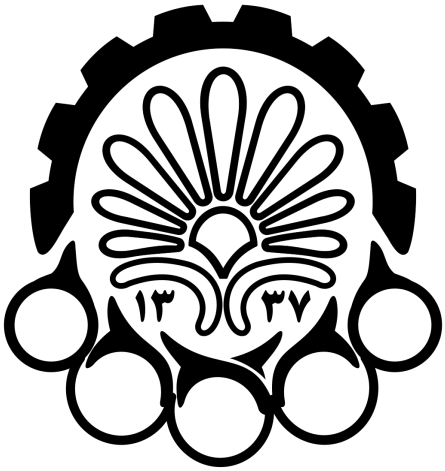
سیستم‌های عامل

دکتر جوادی



دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

رضا آدینه پور ۴۰۲۱۳۱۰۵۵

تمرین سری سوم

۲۰ آذر ۱۴۰۲

سیستم‌های عامل

تمرین سری سوم

رضا آدینه پور ۴۰۲۱۳۱۰۵۵

دانشکده مهندسی کامپیوتر

سوال اول

فرض کنید پردازه های زیر را داریم:

-	Arrival time	Priority number	CPU burst
P0	5	1	20
P1	0	4	40
P2	0	2	20
P3	13	3	5
P4	30	4	5
P5	17	1	10

۱. برای زمان‌بندی های FCFS و SJF و SRT و اولویت پسگیر نحوه تخصیص CPU به پردازه‌ها را نشان دهید. (اگر در زمان‌بندی اولویت پسگیر دو پردازه شرایط یکسان برای انتخاب شدن داشتند آنی را انتخاب کنید که زودتر آمده باشد، همچنین‌در سایر زمانبند ها آنی را انتخاب کنید که پر اولویت تر است)

۲. میانگین طول عمر، زمان انتظار، زمان پاسخ و بازده CPU پردازه ها را برای هر یک از زمان‌بندی های فوق، محاسبه کنید.

پاسخ

محاسبات میانگین طول عمر، زمان انتظار، زمان پاسخ و زمان بازده به صورت زیر حساب شده اند.

- **Average Turnaround Time:**
  Average Turnaround Time $= \frac{\sum Turnaround\ Time}{Nnumber\ of\ process}$
  Turnaround Time $= Exit\ Time\ - Arrival\ Time$
- **Average Waiting Time:**
  Average Waiting Time $= \frac{\sum Waiting\ Time}{Nnumber\ of\ process}$
  Waiting Time $= Turnaround\ Time\ - CPU\ Burst\ Time$
- **Average Response Time:**
  Average Response Time $= \frac{\sum Response\ Time}{Nnumber\ of\ process}$
  Response Time $= Start\ Time\ - Arrival\ Time$
- **Average CPU Utilization:** Average Response Time $= \frac{\sum CPU\ Burst\ Time}{Total\ execution\ time}$

۱. زمان‌بندی FCFS بدین صورت است که پردازه‌ها بر اساس زمان ورودی به صف اجرا قرار می‌گیرند و اولین پردازه وارد صف اجرا در ابتدا، اولویت بالا تری دارد. بنابر این ترتیب اجرای پردازه‌ها به صورت زیر است:

- **ترتیب اجرا:** P۱، P۲، P۰، P۵، P۳، P۴
- **میانگین طول عمر:** $(40 + 20 + 20 + 10 + 5 + 5) \div 6 = 16.67$
- **میانگین زمان انتظار:** $((0-0)+(0-0)+(5-0)+(17-0)+(13-17)+(30-13)) \div 6 = 6.83$
- **میانگین زمان پاسخ:** $((20-0)+(40-0)+(20-5)+(10-17)+(5-13)+(5-30)) \div 6 = 13.33$
- **میانگین بازده:** $(20 + 40 + 20 + 10 + 5 + 5) \div (5 + 40 + 20 + 5 + 5 + 10) = 0.367$

۲. زمان‌بندی SJF: در این الگوریتم، پردازه با کمترین زمان CPU burst در ابتدا انتخاب می‌شود. اگر دو پردازه دارای زمان مشابهی باشند، پردازه‌ای که زودتر وارد صف اجرا شده است، اولویت بالاتری دارد. بنابراین، ترتیب اجرای پردازه‌ها به صورت زیر است:

- **ترتیب اجرا:** P۱، P۲، P۵، P۳، P۴، P۰
- **میانگین طول عمر:** $(40 + 20 + 10 + 5 + 5 + 20) \div 6 = 16.67$
- **میانگین زمان انتظار:** $((0-0)+(0-0)+(17-0)+(13-17)+(30-13)+(5-30)) \div 6 = 6.83$
- **میانگین زمان پاسخ:** $((40-0)+(20-0)+(10-17)+(5-13)+(5-30)+(20-5)) \div 6 = 12.67$
- **میانگین بازده:** $(40 + 20 + 10 + 5 + 5 + 20) \div (5 + 40 + 20 + 5 + 5 + 10) = 0.367$

۳. زمان‌بندی SRT: در این الگوریتم، همانند SJF پردازه با کمترین زمان باقی‌مانده از CPU burst در هر لحظه انتخاب می‌شود. اگر دو پردازه دارای زمان مشابهی باشند، پردازه‌ای که زودتر وارد صف اجرا شده است، اولویت بالاتری دارد. بنابراین، ترتیب اجرای پردازه‌ها به صورت زیر است:

پاسخ

- **ترتیب اجرا:** P۱، P۲، P۳، P۳، P۵، P۴، P۴، P۰، P۰
- **میانگین طول عمر:** $(40 + 20 + 5 + 5 + 10 + 5 + 5 + 20 + 20) \div 9 = 13.89$
- **میانگین زمان انتظار:** $((0-0) + (0-0) + (13-0) + (18-13) + (17-18) + (28-17) + (33-28) + (38-33) + (58-38)) \div 9 = 7.33$
- **میانگین زمان پاسخ:** $((20-0) + (20-0) + (5-13) + (5-13) + (10-17) + (5-17) + (5-28) + (20-33) + (20-58)) \div 9 = 11.89$
- **میانگین بازده:** $(40+20+5+5+10+5+5+20+20) \div (5+40+20+5+5+10+5+20+20) = 0.444$

۱. **الگوریتم اولویت پسگیر:**

- **ترتیب اجرا:** P۱، P۲، P۵، P۰، P۳، P۴
- **میانگین طول عمر:** $(40 + 20 + 10 + 20 + 5 + 5)/6 = 16.67$
- **میانگین زمان انتظار:** $((0-0)+(0-0)+(17-0)+(5-0)+(13-5)+(30-13)) \div 6 = 7.83$
- **میانگین زمان پاسخ:** $((40-0)+(20-0)+(10-17)+(20-5)+(5-13)+(5-30)) \div 6 = 13.33$
- **میانگین بازده:** $(40 + 20 + 10 + 20 + 5 + 5) \div (5 + 40 + 20 + 10 + 5 + 5) = 0.682$

سوال دوم

با در نظر گرفتن زمان context switch = ۰ در هر مورد، نحوه زمان‌بندی پردازه ها را با یک نمودار Gantt نشان دهید:

۱. یک زمانبند چند لایه ای داریم که در آن پردازه ها دوبار وقت دارند در لایه اول زمان‌بندی شوند ،اگر کار آنها به پایان نرسید، وارد الیه دوم شده و یک بار هم وقت دارند آنجا زمان‌بندی شوند و در نهایت اگر کار آن‌ها همچنان به پایان نرسید وارد لایه آخر شده و تا زمانی که کار پردازشی آن‌ها به اتمام برسد در آنجا زمانبندی می شوند،داریم:

لایه اول: RR با کوانتوم زمانی ۴
لایه دوم: RR با کوانتوم زمانی ۸
لایه سوم: FCFS
و پردازه ها مطابق جدول زیر باشند:

-	Arrival time	CPU burst
P0	0	13
P1	0	10
P2	8	17
P3	24	40
P4	40	16
P5	80	4

و همه جا اولویت تخصیص پردازنده با پردازه ای باشد که کار در لایه بالاتر دارد و زمانبند غیرپسگیر باشد.

۲. یک زمان‌بندی کاملا پسگیر داریم که بر اساس اولویت کار می کند و اگر دو پردازه اولویت یکسان داشته‌باشند بر اساس SRT عمل می کند. در این زمانبند، به ازای هر ۲۰ واحد زمانی،در صورتی که کار پردازه به اتمام‌نرسیده باشد، یک واحد از عدد اولویت پردازه ها کم شود و پردازه ها مطابق جدول زیر هستند

-	Arrival time	CPU burst	Priority number
P0	45	9	2
P1	0	15	4
P2	10	30	3
P3	15	20	3
P4	50	26	4

پاسخ

نمودار Gantt برای لایه‌های چندگانه با الگوریتم‌های RR و FCFS:

۱. محاسبه زمان اجرای هر پردازه در هر لایه:

- لایه اول:

P0: 13, P1: 10, P2: 4

(زمان اجرا به اتمام می‌رسد)

- لایه دوم:

P2: 13, P3: 8, P4: 16

(زمان اجرا به اتمام می‌رسد)

- لایه سوم:

P3: 17, P5: 4

(زمان اجرا به اتمام می‌رسد)

۲. نمودار Gantt:

layer 1: $|P0|P1|P2|P2|P3|P4|$
layer 2: $|\quad P2 \quad | \quad P3 \quad |P4|$
layer 3: $|\quad P3 \quad | \quad P5 \quad |$
Time: 0 4 8 12 20 37 53 57 61

نمودار Gantt برای زمانبندی کاملاً پسگیر با اولویت‌های SRT و Priority:

۱. محاسبه زمان اجرای هر پردازه در هر لحظه:

- زمان ۰:

P1: 15, P2: 10

- زمان ۱۰:

P2: 20, P3: 20

- زمان ۲۰:

P0: 9, P3: 10

- زمان ۳۰:

P3: 10, P4: 26

- زمان ۵۶:

P4: 4

پاسخ

۱. نمودار Gantt:

Time: 0 10 20 30 40 50 50
layer 1: |$P1$ |$P2$ |$P3$ |$P4$ |$P0$ |$P3$ |$P4$|

سوال سوم

شروط Manual exclution و Progress و Bounded waiting را برای الگوریتم‌های زیر بررسی کرده و با دلیل توضیح دهید. فرض کنید i و j اعدادی هستند که اندیس پردازه را مشخص می‌کنند.

Listing 1: Code I

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

پاسخ

الگوریتم فوق یک نمونه از الگوریتم بسته‌بندی دوباره پترسون است که برای رسیدگی به مسئله‌ی تضمین منابع اشتراکی بین دو پردازه طراحی شده است. در این الگوریتم، سه شرط اصلی وجود دارد که باید برای آن‌ها اطمینان حاصل شود:

۱. شرط exclusion manual: این شرط مشخص می‌کند که در هر لحظه حداکثر یکی از پردازه‌ها می‌تواند در بخش بحرانی حضور داشته باشد. در این الگوریتم، این شرط توسط بخش

```
while (!flag[j] || turn == j
```

بررسی می‌شود. در این بخش، اگر flag پردازه j فعال باشد و همچنین متغیر turn برابر با j باشد، به این معنی است که پردازه j در بخش بحرانی قرار دارد و بنابراین پردازه فعلی باید منتظر شود تا پردازه j از بخش بحرانی خارج شود.

۲. شرط progress: این شرط تضمین می‌کند که اگر هیچیک از پردازه‌ها در بخش بحرانی قرار نداشته باشد و یک پردازه درخواست ورود به بخش بحرانی داشته باشد، آن پردازه به زودی وارد بخش بحرانی شود. در این الگوریتم، این شرط توسط بخش

```
flag[i] = true
```

بررسی می‌شود. در این بخش، flag پردازه فعلی فعال می‌شود تا نشان دهد که پردازه فعلی دسترسی به بخش بحرانی می‌خواهد.

۳. شرط bounded waiting: این شرط مشخص می‌کند که پردازه‌ها در صورت درخواست ورود به بخش بحرانی، به طور محدود در حال انتظار باشند و پردازه‌ای بی‌نهایت در انتظار نباشد. در الگوریتم فوق، این شرط توسط بخش

```
while (!flag[j] || turn == j)
```

بررسی می‌شود. در این بخش، اگر پرچم پردازه j فعال باشد و متغیر turn برابر با j باشد، به این معنی است که پردازه j در بخش بحرانی قرار دارد و پردازه فعلی باید منتظر شود تا پردازه j از بخش بحرانی خارج شود. این باعث می‌شود که پردازه‌ها در صورت درخواست ورود به بخش بحرانی، به صورتمحدود در حالت انتظار قرار بگیرند و پردازه‌ی فعلی نیز منتظر بماند تا بتواند وارد بخش بحرانی شود.

Listing 2: Code II

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

پاسخ

الگوریتم بالا نیز یک نمونه از الگوریتم بسته‌بندی دوباره پترسون است، اما با تفاوتی در شرط بررسی حلقه while دارد. در این الگوریتم، شرایط خواسته شده به شرح زیر بررسی می‌شوند:

۱. شرط exclusion manual: بخش بررسی شرط exclusion manual در الگوریتم فوق به وسیله عبارت

```
while (!flag[j] && turn == j)
```

انجام می‌شود. این بخش برای ورود به بخش بحرانی همزمانی پردازه‌ها را بررسی می‌کند. اگر پرچم پردازه j غیرفعال باشد و مقدار متغیر turn برابر با j باشد، به این معنی است که هیچ پردازه‌ای در حال حاضر در بخش بحرانی حضور ندارد و پردازه فعلی می‌تواند وارد بخش بحرانی شود. این شرط تضمین می‌کند که حداکثر یک پردازه در بخش بحرانی حضور داشته باشد.

۲. شرط progress: شرط progress در این الگوریتم به صورت ضمنی تضمین می‌شود. عبارت

```
flag[i] = true
```

بلافاصله پس از ورود به حلقه اجرا می‌شود و شرط progress را برآورده می‌کند. این عبارت مشخص می‌کند که پردازه فعلی درخواست ورود به بخش بحرانی دارد و آماده است در آن وارد شود. اگر هیچ پردازه‌ای در حالت بحرانی قرار نگیرد، پردازه فعلی به زودی وارد بخش بحرانی خواهد شد.

۳. شرط bounded waiting: خش بررسی شرط bounded waiting نیز با استفاده از عبارت

```
while (!flag[j] && turn == j)
```

انجام می‌شود. اگر پرچم پردازه j غیرفعال باشد و مقدار متغیر turn برابر با j باشد، به این معنی است که هیچ پردازه‌ای در حال حاضر در بخش بحرانی حضور ندارد و پردازه فعلی می‌تواند وارد بخش بحرانی شود. این باعث می‌شود که پردازه‌ها در صورت درخواست ورود به بخش بحرانی، به صورت محدود در حالت انتظار قرار بگیرند و پردازه‌ی فعلی نیز منتظر بماند تا بتواند وارد بخش بحرانی شود.

سوال چارم

دو پردازه برای حل مسئله‌ی ناحیه بحرانی از روش زیر استفاده کردند. متغیر های s۱ و s۲ بین دو پردازه مشترک هستند و یک مقدار Boolean دارند که در ابتدای اجرای برنامه به صورت تصادفی مقدار دهی شده اند.

```
while(s1 != s2);
  // critical section
s2 =! s1;
```

```
while(s1 == s2);
  // critical section
s2 == s1;
```

۱. بررسی کنید و توضیح دهید که هرکدام از سه شرط Progress و Manual exclution و Nounded waiting برآورده می‌شود یا خیر؟

۲. راه‌حلی برای عدم نقض هرکدام از شرط های بالا ارائه دهید.

پاسخ

۱. **بررسی شرایط:**

- **شرط Progress:**
در هر دو حلقه بالا، شرط Progress برآورده می‌شود. در هر حلقه، پردازه‌ها قبل از ورود به بخش بحرانی مقادیر متغیرهای s۱ و s۲ را بررسی می‌کنند و در صورتی که مقادیر آنها با هم متفاوت باشند، وارد بخش بحرانی می‌شوند. این شرط تضمین می‌کند که حداقل یک پردازه در هر حلقه در بخش بحرانی حضور داشته باشد.
- **شرط exclusion manual:**
در حلقه اول و دوم، شرط Exclusion manual برآورده می‌شود. زیرا پردازه‌ها قبل از ورود به بخش بحرانی، مقادیر s۱ و s۲ را بررسی می‌کنند و در صورتی که مقادیر آنها با هم برابر نباشند، وارد بخش بحرانی نمی‌شوند. این شرط تضمین می‌کند که همزمان حداکثر یک پردازه در بخش بحرانی حضور داشته باشد.
- **شرط waiting bounded:** در هیچ یک از حلقه‌ها، شرط Waiting bounded برآورده نمی‌شود. زیرا در هر حلقه، پردازه‌ها به صورت بی‌پایان منتظر می‌مانند تا مقادیر s۱ و s۲ با هم متفاوت یا برابر شوند. این باعث می‌شود که پردازه‌ها در صورتی که مقادیر s۱ و s۲ با هم برابر نباشند، به صورت بی‌نهایت در حالت انتظار قرار بگیرند.

۲. **راه‌حل برای عدم نقض شرایط:**

- برای رعایت شرط Waiting bounded می‌توان از یک تاخیر استفاده کرد. به این صورت که پردازه‌ها پس از بررسی مقادیر s۱ و s۲، یک تاخیر کوتاه داشته باشند و سپس دوباره مقادیر را بررسی کنند. این کار باعث می‌شود که پردازه‌ها در صورتی که مقادیر s۱ و s۲ با هم متفاوت یا برابر باشند،در حلقه انتظار کوتاهی داشته باشند و منتظر تغییر مقادیر باشند. این روش می‌تواند به صورت زیر پیاده‌سازی شود:

```
while(s1 != s2);
  // critical section
s2 =! s1;
delay();
```

سوال پنجم

۱. بدون استفاده از قفل و تنها با استفاده از دستور CompareAndSwap تابع زیر را به گونه ای کامل کنید که به صورت اتمی عملیات تفریق را انجام دهد. منظور از عملیات تفریق کم شدن مقدار V از حافظه ای که P به آن اشاره دارد. سپس توضیح دهید تضمینی برای انجام شدن این عملیات وجود دارد یا خیر

```
int sub(int *p, int v)
{
  // TODO
  return *p-v
}
```

برای استفاده از دستور CompareAndSwap از دستور زیر استفاده کنید:

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

پاسخ

برای انجام عملیات تفریق به صورت اتمی می‌تواند از تابع زیر استفاده کرد:

```
int sub(int *p, int v)
{
  int old_val, new_val;
  do
  {
        old_val = *p;
        new_val = old_val - v;
  } while (!CompareAndSwap(p, old_val, new_val));

  return new_val;
}
```

تابع CompareAndSwap بررسی می‌کند که مقدار فعلی در حافظه با مقدار قبلی که در تابع Compare-AndSwap به عنوان ورودی داده شده برابر است یا خیر. اگر برابر بود، مقدار جدید در مکان مورد نظر قرار داده می‌شود و عملیات تفریق انجام می‌شود. در غیر این صورت، عملیات تفریق انجام نمی‌شود و تکرار دیگری صورت می‌گیرد تا زمانی که مقدار فعلی در حافظه با مقدار قبلی تطابق داشته باشد.
با استفاده از این روش، تضمین می‌شود که تغییرات در مقدار حافظه از طریق عملیات تفریق انجام می‌شود و در صورتی که همزمان تغییر دیگری در مقدار حافظه صورت گیرد، تابع CompareAndSwap بازخوانی می‌شود و تغییر با موفقیت انجام نمی‌شود. به این ترتیب، عملیات تفریق به صورت اتمی و بدون تداخل با عملیات دیگر در حافظه انجام می‌شود.

۲. حال توضیح دهید CompareAndSwap چگونه کنترل همزمانی را در برنامه های چند نخی را بهبود می‌بخشد و در چه مواقعی کاربرد دارد؟

پاسخ

به طور کلی، تابع CompareAndSwap بهبود کنترل همزمانی در برنامه‌های چند تابع CompareAndSwap یک عملیات همزمانی است که در برنامه‌های چندنخی استفاده می‌شود و در بهبود کنترل همزمانی نخ‌ها نقش مهمی دارد. CAS به صورت اتمی عملیات خواندن و نوشتن را انجام می‌دهد، به این معنی که در یک عملیات تکمیل شده، مقدار قبلی در حافظه را بررسی کرده و اگر با مقدار تعیین شده برابر بود، مقدار جدید را در حافظه قرار می‌دهد. اگر مقدار قبلی با مقدار تعیین شده برابر نبود، تغییری انجام نمی‌شود.
با استفاده از تابع ،CAS می‌توان کنترل همزمانی بین نخ‌ها را بهبود بخشید و مشکلاتی که در همزمانی ممکن است پیش بیاید را رفع کرد. از جمله مواردی که CAS کاربرد دارد می‌توان به موارد زیر اشاره کرد:

- **همگام‌سازی دسترسی به منابع مشترک:** در برنامه‌های چندنخی، اگر نخ‌ها به منابع مشترک دسترسی داشته باشند، ممکن است همزمانی مشکلاتی ایجاد کند. با استفاده از تابع CAS، می‌توان همگام‌سازی دسترسی به منابع را بهبود بخشید و تداخل‌ها را کاهش داد.
- **تحقق عملیات اتمی:** در برخی موارد، نیاز است که یک عملیات را به صورت اتمی انجام داده و تضمین کنیم که هیچ تغییر همزمانی دیگری در آن زمان انجام نشده است. با استفاده از تابع CAS، می‌توان عملیات‌هایی مانند تفریق اتمی، اضافه کردن به مجموعه Atomic Add یا جایگزینی اتمی Swap Atomic را پیاده‌سازی کرد.
- **روش‌های بدون انتظار:** تابع CAS به صورت NonBlocking عمل می‌کند، به این معنی که اگر نخ دیگری قفل را نگه داشته باشد، نخ دیگری که از CAS استفاده می‌کند، به صورت مداوم بررسی می‌کند تا زمانی که قفل آزاد شود و عملیات را انجام دهد. این روش می‌تواند در بهبود عملکرد و کاهش هزینه‌های همزمانی در برنامه‌های چندنخی مؤثر باشد.

سوال ششم

در این تمرین می‌خواهیم نحوه عملکرد Scheduler های لینوکس را مورد بررسی قرار دهیم. به‌طور کلی در لینوکس سه نوع Scheduler وجود دارد:

1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR

در زبان C می‌توانیم مشخص کنیم که با چه Scheduler ای برنامه‌مان اجرا شود. برای این تمرین نیاز داریم که دو پردازه داشته باشیم و آنها را با Scheduler های مختلف تست کنیم.
برای نوشتن این برنامه به کتابخانه های زیر نیاز داریم:

```
   #include <sched.h>
#include <stdio.h>
#include <sys/resource.h>
#include <unistd.h>
```

برای تغییر Scheduler policy برنامه، از دستور زیر استفاده می‌شود:

```
   struct sched_param param
sched_setscheduler(getpid(), SCHED_FIFO, &param)
```

ساختار sched_param پارامتر های scheduler را مشخص می‌کند. برای تغییر اولویت پردازه از این استراکت استفاده می‌کنیم:

```
   param.sched_priority = PRIORITY_NUM
```

نکته: در اینجا هر چه مقدار بیشتر باشد، اولویت بالاتر است. و مقداری که اینجا تعیین میشود با مقداری که در ستون Priority پردازه در ستون Top می‌بینید متفاوت است.
بااستفاده از این دستورات، برنامه ای بنویسید و سناریو های زیر را برای دو پردازه همزمان تست کنید (برنامه ای بنویسید که زمان زیادی داشته باشد که بتوانید خروجی ها را مقایسه کنید. برای نمونه می توانید از قطعه کد زیر استفاده کنید:)

```
int n = 0;
while(1)
{
  n++;
  if(!(n % 10000000))
  {
        printf("FIFO running (n=%d) \n", n)
  }
}
```

نکته: تغییر scheduler یک پردازه نیازمند دسترسی Root است بنابر این برنامه را با sudo اجرا کنید.
نکته: دوپردازه را باید روی یک cpu اجرا کنیم تا نتیجه مد نظر را ببینیم. برای اجرای یک پردازه روی یک cpu core از دستور زیر استفاده می‌کنیم:

```
   sudo taskset -c 0 ./FIFO.o
```

سناریو ۱:
پردازه اول: SCHED_FIFO و priority=۱
پردازه دوم: SCHED_FIFO و priority=۱

سناریو ۲:
پردازه اول: SCHED_RR و priority=۱
پردازه دوم: SCHED_RR و priority=۱

سناریو ۳:
پردازه اول: SCHED_FIFO و priority=۱
پردازه دوم: SCHED_FIFO و priority=۲

پاسخ

برنامه نوشته شده به صورت زیر است:

Listing 3: code

```
#include <sched.h>
#include <stdio.h>
#include <sys/resource.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
  struct sched_param param;
  param.sched_priority = 1;

  sched_setscheduler(getpid(), SCHED_FIFO, &param);

  pid_t pid = fork();

  if (pid == 0)
  {
        int n = 0;
        while (1)
        {
          n++;
          if (!(n % 10000000))
          {
            printf("Running (n=%d)\n", n);
          }
          else if (pid > 0)
          {
                wait(NULL);
          }
          else
          {
                printf("Fork failed\n");
                return 1;
          }

          return 0;
          }
        }
}
```

سپس برنامه را با دستور زیر کامپایل و اجرا می‌کنیم:

```
gcc -o scheduler_program scheduler_main.c
sudo taskset -c 0 ./scheduler_program
```