فصل ۵
همروندی
SYNCHRONIZATION

استاد : علی روانگرد

- مفاهیم اولیه
- همگام سازی (Synchronization)
- اگر بین فرایندها وابستگی وجود داشته باشد، ترتیب درست انجام کارها باید رعایت شود.
- شرایط رقابتی
- فرایندها در فعالیت های بحرانی یکدیگر مداخله نکنند و شرایط رقابتی برای آن ها رخ ندهد.
- منبع بحرانی
- منبع غیراشتراکی (مانند چاپگر) که چند فرایند برای دسترسی به آن رقابت می کنند.
- ناحیه بحرانی (Critical Region)
- بخشی از برنامه که از منبع بحرانی استفاده می کند.

- گرسنگی
- فرض کنید هر یک از سه فرایند A و B و C متناوبا نیازمند دسترسی به منبع R هستند.
- وقتی A این منبع را در اختیار گیرد در فرایند دیگر در انتظار آن منبع، به تاخیر انداخته می شوند. با خروج A از ناحیه بحرانی، یکی از دو فرایند باید R را در اختیار گیرند.
- اگر C منبع R را بگیرد و قبل از پایان بخش بحرانی مجددا A درخواست R را بکند و بعد از پایان، اجازه به C داده شود و مکررا این عمل بین A و C ادامه یابد، B به صورت نامحدود از دسترسی به منبع R محروم می ماند و گرسنگی می کشد.

- مثال: خروجی های ممکن در صورت اجرای همروند و موازی دو پروسس زیر:

A(void){
 	
while(true){
cout <<
1;
cout<<2;
}
}
	B(void){
	
	while(true){
	cout<<3;
	cout<<4;
	}
	}

1234: A → B

3421: B→A

1342: A(1) → B → A(2)

3124: B(1) → A → B(2)

1324 : A(1)→B(1)→ A(2)→B(2)

3142 : B(1) → A(1) → B(2) → A(2)

- مثال: دو فرایند زیر به صورت هم روند اجرا می شوند. در صورتی که مقدار اولیه متغیر سراسری a صفر باشد، بعد از اجرای کامل دو فرایند، چه حالت هایی می تواند رخ دهد؟
- (امکان اجرای آنها به صورت interleaved نیز وجود دارد. یعنی در هر لحظه از اجرای فرایند، امکان وقوع وقفه و سوییچ به فرایند دیگر وجود دارد)
  - P1 → P2 ➔ a = b = c = 1
  - P2 → P1 ➔ a = 1, b = 0, c = 0
  - P2(1) → P1 →P2(2) ➔ a = 1, b=0, c=1

P1	P2
a = 1;	b = a; c = a;

- مثال
- چند خروجی حاصل از اجرای همروند دو پردازنده:

P1	P2
Repeat print “A” print ”B” forever	Repeat print ”C” print ”D” forever

- (AB) (CD) : P1 → P2
- A(CD)B : P1(1) → P2 → P1(2)
- ACDB: P1(1) → P2 → P1(2)

- همکاری فرایند های همروند
- شرایطی که باید رعایت شود تا یک همکاری درست و کارا بین فرایندهای هم روند برقرار باشد:
- ۱- انحصار متقابل(mutual exlcusion)
- از بین فرایندهایی که برای یک منبع یکسان دارای ناحیه بحرانی هستند، در هر لحظه فقط یک فرایند مجاز است که در ناحیه بحرانی خود باشد.

- ۲- پیشرفت (progressive)
- فرایندی که فعلا تصمیم به ورود به ناحیه بحرانی را ندارد و در ناحیه غیر بحرانی می باشد و دستورالعمل های عادی برنامه خود را اجرا می کند، نباید در تصمیم گیری برای ورود فرایند های دیگر به ناحیه بحرانی شرکت کند. (امکان ممانعت نداشته باشد)

- ۳- انتظار محدود (bounded waiting)
- باید مدت انتظار فرایندهایی که نیاز به ورود به ناحیه بحرانی دارند، محدود باشد. یعنی نباید دچار گرسنگی و بن بست شوند.
- گرسنگی: به مدت نامعلوم و بدون حد بالای مشخص، منتظر فرایندهای دیگر بودن.
- بن بست: تا ابد منتظر ورود به ناحیه های بحرانی خود بودن.

- البته علاوه بر رعایت ۳ شرط بالا مسئله را باید در حالت کلی حل کرد و فرضی برای ساده سازی راه حل به کار نبرد. همچنین الگوریتم در حالت قطعی و غیرتصادفی داشته باشد.

- رویکرد های انحصار متقابل
- برای تحقق انحصار متقابل، پیشنهادهای مختلفی وجود دارد.
- این راه حل ها را به صورت چهار رویکرد زیر، دسته بندی می کنیم.
- ۱- نرم افزاری
- ۲- با حمایت سخت افزار (با کمک دستورالعمل های خاص CPU)
- ۳- با حمایت سیستم عامل (با کمک فراخوان های سیستمی خاص)
- ۴- با حمایت زبان برنامه سازی (با کمک کامپایلر)

- رویکردهای نرم افزاری انحصار متقابل
- مستقیما توسط برنامه ها استفاده می شوند و وجود حافظه اشتراکی ضروری است.
- حمایتی از سیستم عامل و زبان برنامه سازی نداریم.
- الگوریتم Decker : اولین شخصی که یک راه حل نرم افزاری دو فرایندی برای مسئله انحصار متقابل ارائه داد.
- Decker با پنج مرحله تلاش راه حل درست رسید.

- تلاش اول (تناوب قطعی)

```
P0(){
   while(TRUE)
      {
         while(turn !=0);
         critical-section();
         turn=1;
         non-CS();
}
}
```

```
P1(){
   while(TRUE)
      {
         while(turn !=1);
         critical-section();
         turn=0;
         non-CS();
}
}
```

- شرط پیشرفت را رعایت نمی کند.
- سناریو:
- اگر p1 وارد ناحیه بحرانی شود و کارش تمام شده و به بخش غیربحرانی برود، turn برابر ۰ می باشد. حال نوبت p0 است که وارد ناحیه بحرانی شود، ولی میخواهد به مدت طولانی در ناحیه غیربحرانی بماند.
- p1 به سرعت کارش در ناحیه غیربحرانی تمام شده و قصد ورود مجدد به ناحیه بحرانی را دارد. اما چون turn برابر صفر است در حلقه انتظارمی ماند تا بالاخره P0 وارد ناحیه بحرانی شده و بعد از خروج، turn را ۱ کرده تا P1 بتواند وارد ناحیه بحرانی شود. پس p1 توسط فرایندی منتظر مانده بود که در ناحیه بحرانی نبود و جلوی پیشرفتش را گرفته بود.

- **تلاش دوم**

P0(){
while(TRUE)
{
while(flag[1]);
flag[0] = TRUE;
critical-section();
flag[0] = FALSE;
non-CS();
}
}
	P1(){
	while(TRUE)
	{
	while(flag[0]);
	flag[1] = TRUE;
	critical-section();
	flag[1] = FALSE;
	non-CS();
	}
	}

- هر فرایند دارای کلید مجزا برای ورود به ناحیه بحرانی است تا اگر فرایندی قصد استفاده ازناحیه بحرانی را نداشت، فرایند دیگر بتواند به ناحیه بحرانی حود دسترسی داشته باشد.

- **انحصار متقابل رعایت نمی شود.**
- سناریو:
- فرایند p0، flag[1] را خوانده و آن را false می بیند اما قبل از true کردن flag[0] ، p1 اجرا شود و flag[0] را خوانده و آن را false می بیند و برای ورود به ناحیه بحرانی flag[1] را true کرده و p1 وارد می شود. در این زمان به p0 سوئیچ شده و flag[0] را true کرده و این فرایند هم وارد ناحیه بحرانی می شود.
- **به علت امکان گرسنگی شرط انتظار محدود رعایت نمی شود.**
- سناریو:
- فرض کنید p0 در ناحیه بحرانی است و به p1 سوئیچ می شود. چون flag[0] برابرtrue است، p1 نمی تواند وارد ناحیه بحرانی شود. بعد از پایان کوانتوم، پردازنده به p0 داده شده و این فرایند سریعا ناحیه بحرانی اش را اجرا کرده و سعی به ورود مجدد به ناحیه بحرانی را دارد و ذاین اجازه به او داده می شود. P0 مجددا flag[0] را true کرده و اگر به p1 سوئیچ شود، باز هم نمی تواند اجازه ورود بگیرد.

- بنابراین تلاش برای دستیابی به ناحیه بحرانی تصادفی است و امکان قحطی وجود دارد.
- امکان ورود پی در پی یک فرایند به ناحیه بحرانی و عدم دستیابی فرایند دیگر به ناحیه بحرانی وجود دارد.

- **تلاش سوم**
- در تلاش دوم هر فرایند ابتدا وضعین فرایند مقاب را چک کرده و سپس flag خود را true می کند. بنابراین اگر هر دو به طور همزمان قصد ورود به ناحیه بحرانی داشته باشند، flag یکدیگر را false میبینند و با هم وارد می شوند. برای حل این مشکل دو سطر مسئله را عوض می کنیم.

```
P0(){
  while(TRUE)
    {
      flag[0]  =TRUE;
      while(flag[1]);
      critical-section();
      flag[0] = FALSE;
      non-CS();
}
}
```

```
P1(){
  while(TRUE)
    {
      flag[1]  =TRUE;
       while(flag[0]);
       critical-section();
       flag[1] = FALSE;
       non-CS();
}
}
```

- **به علت امکان بن بست، شرط انتظار محدود رعایت نمی شود.**
- سناریو:
- فرض کنید که p0، flag[0] را true کند ولی قبل از بررسی flag[1] در p0، به p1 سوئیچ شود و p1 ، flag[1] را true کند. در این صورت هر دوفرایند تا ابد در حلقه گرفتار شده و بن بست رخ می دهد.

- **تلاش چهارم (ادب و تعارف)**
- در تلاش قبلی هر فرایند می تواند روی حق خود برای ورود به بحش بحرانی اش پافشاری کند. در این روش هر فرایند متغیر flag خود را true کرده تا خواست خود برای ورود به بحش بحرانی را نشان دهد،اما آماده است flag را تعییر دهد تا به فرایند دیگر احترام بگذارد.
- یعنی فرایندی که قصد ورود به ناحیه را دارد، اگر ببیند گه فرایند مقابل هم میخواهد به ناحیه بحرانی وارد شود، flag خود را برای مدت کوتاهی false کرده تا فرایند مقابل وارد شود .

```
P0(){
  while(TRUE)
    {
      flag[0] =TRUE;
      while(flag[1]){
          flag[0]=FALSE;
          delay();
          flag[0]=TRUE;
       }
       critical-section();
       flag[0] = FALSE;
       non-CS();
}
}
```

```
P1(){
  while(TRUE)
    {
      flag[1] =TRUE;
      while(flag[0]){
          flag[1]=FALSE;
          delay();
          flag[1]=TRUE;
       }
       critical-section();
       flag[1] = FALSE;
       non-CS();
}
}
```

- **به دلیل امکان گرسنگی، شرط انتظار محدود رعایت نمی شود.**
- سناریو:
- چون ممکن است یک فرایند به مدت نامعلوم و بدون حد بالای مشخص، گرفتار قسمت تاخیر(delay) شود و فرایند مقابل به دفعات وارد ناحیه بحرانی شود.
- همچنین به علت امکان وجود Livelock، نیز شرط انتظار محدود رعایت نمی شود.

- **مشکل Livelock**
- در تلاش چهارم مشکل بن بست وجود ندارد، اما مشکل جدیدی به نام Livelock وجود دارد.
- با دنبال کردن اجرای زیر، این مشکل را توضیح می دهیم:
- ۱) p0، flag[0] را true کند.
- ۲) p1، flag[1] را true کند.
- ۳) p0، flag[1] را بررسی کند.
- ۴) p1، flag[0] را بررسی کند.
- ۵) p0، flag[0] را false کند.
- ۶) p1، flag[1] را false کند.

هر دو فرایند در سیک زمان به مدت کوتاه یکسان عقب نشینی می کنند. سپس با هم بر می گردند و مراحل بالا را تکرار می کنند. اگر این دنباله به طور نامحدود تکرار شود، ممکن است هیچ کدام از فرایند ها نتوانند وارد ناحیه بحرانی شوند.
البته این تکرار بن بست نمی باشد، چون با تغییر در سرعت نسبی فرایند ها، این چرخه شکسته می شود.

- **تلاش پنجم**
- ترکیب متغیر نوبت با متغیرهای پرچم

```
P0(){
  while(TRUE)
    {
      flag[0]  =TRUE;
      while(flag[1]){
          if( turn == 1){
              flag[0]=FALSE;
              while(turn == 1) do;
              flag[0]=TRUE;
          }
      }
      critical-section();
      turn = 1;
      flag[0] = FALSE;
      non-CS();
}
}
```

```
P1(){
  while(TRUE)
    {
      flag[1]  =TRUE;
      while(flag[0]){
          if (turn == 0){
              flag[1]=FALSE;
              while(turn == 0) do;
              flag[1]=TRUE;
          }
      }
      critical-section();
      turn = 0;
      flag[1] = FALSE;
      non-CS();
}
}
```

- هنگامی که p0 بخواهد وارد بحش بحرانی خود شود، در flag مربوط به خود مقدار true می گذارد. سپس flag مربوط به p1 را بررسی می کند که دو حالت رخ می دهد:
- الف - flag[1] برابر true باشد:
- اگر turn برابر یک باشد ، p0 به p1 احترام گداشته و با false کردن پرچمش منتظر می ماند. در این هنگام p0 کاری انجام نمی دهد تا turn برابر صفر شود و سپس flag خودش را true می کند.

- ب- flag[1] برابر false باشد:
- p0 وارد بخش بحرانی شده و بعد از خروج از بخش بحرانی، در flag خودمقدار false می گذارد تا بخش بحرانی را آزاد کند و در turn مقدار ۱ را قرار می دهد تا حق پافشاری را به p1 واگذارد.

- **خلاصه ای از وضعیت تلاش های Decker**
- در جدول زیر، هر حا که شرط رعایت می شود با علامت + مشخص شده است.

	انتظار متقابل	پیشرفت	انتظار محدود
تلاش اول	+	-	+
تلاش دوم	-	+	-
تلاش سوم	+	+	-
تلاش چهارم	+	+	-
تلاش پنجم	+	+	+

- **الگوریتم peterson**
- چندین سال بعد، peterson را حل ساده و زیبایی را برای حل مسئله انحصار متقابل ارائه کردکه برای n فرایند نیز قابل تعمیم است.
- فرض کنید هر دو به طور تقریبا همزمان (p1 کمی دیرتر)، قصد ورود به ناحیه بحرانی را دارند. هر دو فرایند، شماره خود را در turn ذخیره کرده ولی p1 که دیرتر شماره اش را ذخیره کرده، شماره اش در turn می ماند و turn برابر ۱ می شود. حال زمانی که هر دو به دستور while می رسند، p0 از حلقه عبور میکرده و وارد ناحیه بحرانی می شود ولی p1 در حلقه می چرخد(انتظار مشغول) و وارد ناحیه بحرانی نمی شود.

```
P0(){
  while(TRUE)
    {
      flag[0] =TRUE;
      turn = 0;
      while(turn == 0 && flag[1]);
      critical-section();
      flag[0] = FALSE;
      non-CS();
}
}
```

```
P1(){
  while(TRUE)
    {
      flag[1] =TRUE;
      turn = 1;
      while(turn == 1 && flag[0]);
       critical-section();
       flag[1] = FALSE;
       non-CS();
}
}
```

- **تلاش چهارم (ادب و تعارف)**
- در کتاب استالینگز، مقدار **turn** برعکس مقدار دهی و تست می شود که تفاوتی با روش بالا ندارد.
- به طور مثال برای **p0**، داریم:

```
turn =1;
while(turn == 1 && flag[1]);
```

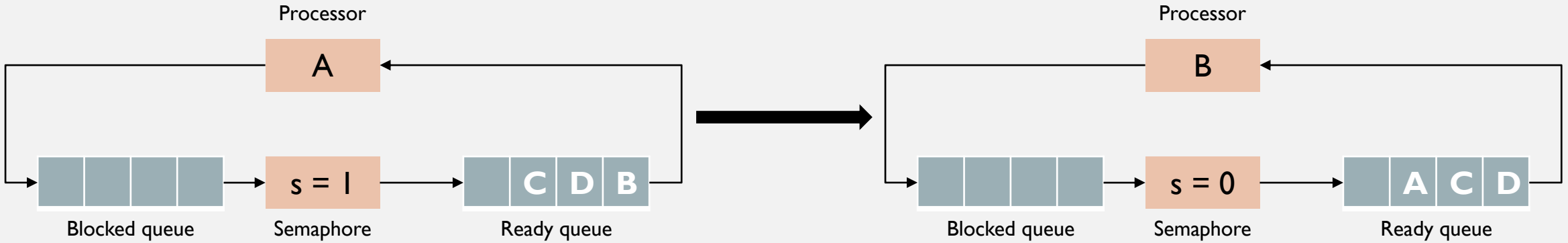
- **معایبی که در هر یک از تلاش های decker و روش peterson وجود دارند:**
- ۱- اگر یکی از فرایندها در داخل ناحیه بحرانی از کار بیافتد، فرایند دیگر تا ابد منتظر می ماند.
- ۲- متنی بر انتظار مشغول می باشند.

- **سمافور**
- راه حل های نرم افزاری و سخت افزاری که بررسی کردیم، دارای نقاط ضعف زیادی بودند.
- سمافور قدرت زیادی در برقراری انحصار متقابل دارد و از عهده انواع مختلف مسائل همگام سازی برمی آید.
- سمافور یک ساختار شامل فیلدهای زیر است:
- **۱- شمارنده صحیح (counter)**
- شمارش تعداد wakeupهایی که می خواهند هدر بروند.
- (ذخیره سیگنال ها برای استفاده های بعدی)
- **۲- صف (queue)**
- نگهداری فرایندهای بلوکه شده بر روی سمافور

```
Struct semaphore{
    int count;
    queuetype queue;
};
```

- **تابع wait**
- تابع wait، یک واحد از شمارنده کم کرده و اگر منفی شود، فرایند در صف، مسدود می شود.

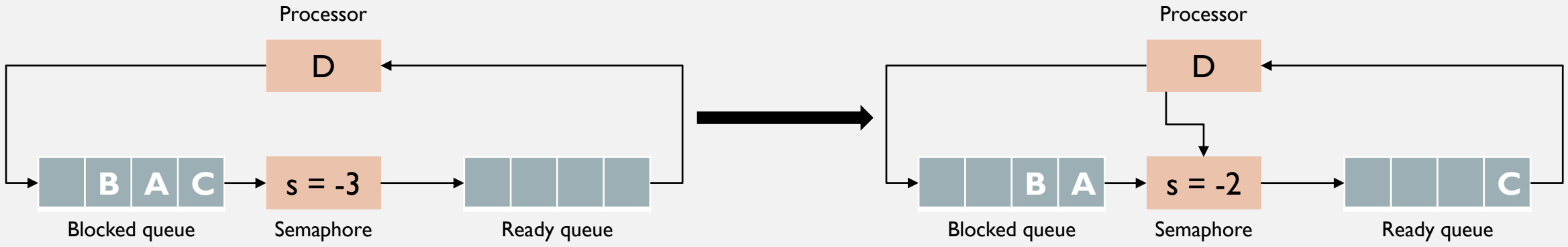
```
Void wait(semaphore s){
    s.count = s.count – 1;
    If(s.count < 0 ){
            place this process in s.queue;
            block this process;
    }
};
```



- **تابع signal**
- تابع **signal**، یک واحد از شمارنده اضافه می کند، اگر مقدار شمارنده بیشتر از صفر نشود، یک فرایند از قبل مسدود شده در صف، آزاد می شود.

```
Void signal(semaphore s){
    s.count = s.count + 1;
    If(s.count <= 0 ){
            remove a process from s.queue;
            place this process in ready queue;
    }
};
```

- نکته: در بعضی از متون از **down** یا **P** بجای **wait** و از **up** یا **V** بجای **signal** استفاده می شود.

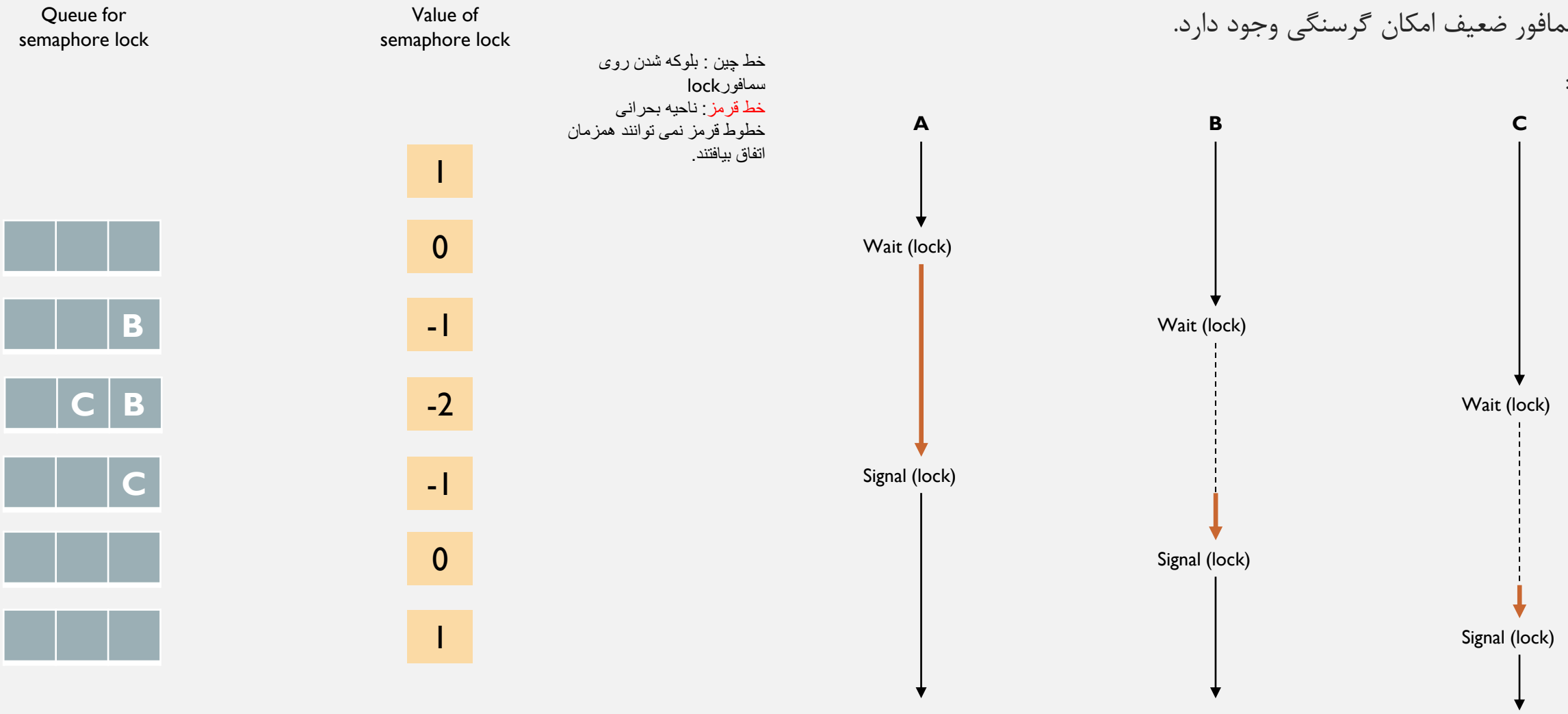


- **انحصار متقابل با استفاده از سمافور**
- انحصار متقابل برای دو فرایند p1 و p2:

```
Semaphore mutex = 1;
Void p (int i){
        while(TRUE){
                wait(mutex);
                crititcal_section();
                signal(mutex);
                non_critical_section();
}
}
```

- نکته:پیاده سازی انحصار متقابل با استفاده از سمافور، شرط های انحصار متقابل، پیشرفت و انتظار محدود را برآورده می کند.

- **سمافور قوی و ضعیف**
- در سمافورها از صفی برای نگهداری فرایندهای بلوکه شده استفاده می شود.
- اگر خروج از این صف به ترتیب ورود باشد (FIFO) به آن سمافور قوی می گویند و اگر چنین نباشد، به آن سمافور ضعیف می گویند.
- در سمافور ضعیف امکان گرسنگی وجود دارد.
- مثال:



- **سمافور عمومی و دودویی**
- **۱- عمومی**
- در سمافور عمومی که آن را بررسی کردیم، شمارنده می تواند مثبت، صفر یا منفی باشد.
- **۲- دودویی**
- در سمافور دودویی، شمارنده فقط میتواند مقادیر ۰ و ۱ باشد.

- قدرت سمافور دودویی و عمومی برابر است.
- **توابع wait و signal برای سمافور باینری**

```
Void wait(semaphore s){
    if (s.count = 1)
        s.count = 0;
    else{
        place this process in s.queue;
        block this process;
    }
}
```

```
Void signal(semaphore s){
    if(s.queue is empty)
        s.count = 1;
    else{
        remove a process from s.queue;
        place this process in ready queue;
}
}
```

- **همگام سازی با استفاده از سمافورها**
- سمافور دارای توانایی زیادی در حل مسائل همگام سازی است.
- مثال۱: می خواهیم ابتدا دستور S1 و سپس دستور S2 اجرا شود. از سمافوری به نام S با مقدار اولیه صفر استفاده می کنیم.

p1	p2
S1; signal(S);	wait(S); S2;

p1	p2
S1;	S2;

- تا زمانی که S1 اجرا نشود، اجرای S2 ممکن نیست.
- مثال ۲: مقدار a و b چند خواهد شد؟ (مقدار اولیه سمافور s برابر یک و مقدار اولیه متغیرهای a,b نیز برابر یک می باشند.)

p1	p2

.
.
.
a=a+2;
b=b+1;
signal(s);
.
.
b=b+1;
	a=3;
	wait(s);
	b=a+b;
	.
	.
	.
	wait(s);
	a=a+b;
	.

p1	p2

a=a+2;
b=b+1;
signal(s);
b=b+1;
	a=3;
	wait(s);
	b=a+b;
	wait(s);
	a=a+b;

- **مسئله تولید کننده و مصرف کننده**
- **یک تولید کننده یا بیشتر، نوعی داده را تولید و آنها را در بافری به اندازه n قرار می دهند.**
- **یک مصرف کننده، این اقلام را یکی یکی از بافر برمی دارد.**
- **در هر زمان مصرف کننده یا تولید کننده می تواند به بافر دسترسی داشته باشد.**
- سمافور های مورد نیاز:
- ۱- سمافور mutex: سمافوری برای رعایت شرط انحصار متقابل است، تا تولید کننده و مصرف کننده به طور همزمان به بافر دسترسی نداشته باشند. (با مقدار اولیه ۱)
- ۲- سمافور full : سمافوری برای شمارش تعداد خانه های پر بافر (با مقدار اولیه ۰)
- ۳- سمافور empty: سمافوری برای شمارش تعداد خانه های خالی بافر (با مقدار اولیه n)

- مسئله تولید کننده و مصرف کننده

```
void producer(){
    int item;
    while(TRUE){
        item = produce();
        wait(empty);
        wait(mutex);
        insert(item);
        signal(mutex);
        signal(full);
    }
}
```

```
void consumer(){
    int item;
    while(TRUE){
        wait(full);
        wait(mutex);
        item = remove();
        signal(mutex);
        signal(empty);
        consume();
    }
}
```

- **مسئله غذا خوردن فیلسوف ها**
- پنج فیلسوف ها داریم. زندگی هر فیلسوف از دو دوره متناوب خوردن و فکر کردن تشکیل شده است.
- هر فیلسوف یک بشقاب ماکارونی دارد. بین هر جفت از بشقاب ها، یک چنگال قرار دارد.(دور یک میز دایره ای پنج بشقاب و پنج چنگال قرار دارد)
- هر فیلسوف برای خوردن از دو چنگال طرفین بشقاب استفاده می کند.
- زمانی که هر فیلسوف گرسنه می شود، سعی می کند دو چنگال سمت چپ و راست خود را بردارد. اگر موفق شد، برای مدتی غذا می خورد و سپس چنگال ها را زمین می گذارد و به فکر ادامه می دهد.
- مسئله تغذیه فیلسوفان علاوه بر انحصار متقابل (که در یک زمان دو فیلسوف نمی توانند از یک چنگال استفاده کنند)، باید جوابگوی بن بست و گرسنگی نیز باشد.

```
void main(){
        parbegin(p(0), p(1), p(2), p(3), p(4));
}
```

هر فیلسوف ابتدا چنگال چپ و سپس چنگال راست را برمی دارد. بعد از تغذیه یک فیلسوف، دو چنگالی که استفاده می کرد را روی میز گذاشته و دیگران می توانند استفاده کنند.
سمافور room با مقدار اولیه ۴، برای این است که اجازه ورود به بیش از چهار نفر داده نشود.
اگر حداکثر چهار فیلسوف نشسته باشند، حداقل یک نفر به دو چنگال دسترسی خواهد داشت.
اگر از room استفاده نمی شد و اجازه ورود هزمان به پنج نفر داده می شد، بن بست اتفاق می افتاد.

```
semaphore room = 4;
semaphore fork[5] = {1};

void philosopher (int i){
        while(TRUE){
                think();
                wait(room);
                wait(fork[i]);
                wait(fork[(i+1)%5];
                eat()
                signal(fork[(i+1)%5]);
                signal(fork[i]);
                signal(room);
        }
}
```

- **مسئله خوانندگان و نویسندگان**
- **در این مسئله، ناحیه داده ای مثل فایل وجود دارد که بین تعدادی از فرایندها مشترک است.**
- **فرایندهای خواننده می خواهند از این ناحیه بخوانند و فرایندهای نویسنده می خواهند در آن بنویسند.**

- **شرایط این مسئله:**
- **۱- هر تعداد از خوانندگان می توانند به صورت همزمان از فایل بخوانند.**
- **۲- در هر زمان تنها یک فرایند ممکن است در این فایل بنویسد.**
- **۳- هنگامی که نویسنده ای در حال نوشتن است، هیچ خواننده ای نمی تواند فایل را بخواند.**
- برای مثال یک سیستم رزرواسیون هواپیمایی را در نظر بگیرید که تعداد زیادی فرایند در آن برای نوشتن و خواندن با یکدیگر رقابت می کنند.
- چند فرایند می توانند به طور همزمان پایگاه داده را بخوانند ولی اگر یک فرایند در حال به روز رسانی پایگاه داده باشد، فرایندهای دیگر حتی خوانندگان، نباید به پایگاه داده دسترسی داشته باشند.

- **خوانندگان اولویت دارند**
- تا زمانی که خواننده ای وجود دارد، به خوانندهه اجازه ورود می دهیم.

```
typedef int semaphore;
semaphore mutex=1;
semaphore w=1;
int rc=0;

void writer(){
        while(TURE){
                wait(w);
                writing();
                signal(w);
}
}
```

```
void reader(){
        while(TURE){
                wait(mutex);
                rc=rc+1;
                if(rc==1) wait(w);
                signal(mutex);
                reading();
                wait(mutex);
                rc = rc – 1;
                if(rc==0) signal(w);
                signal(mutex);
}
}
```

- سمافور w: اعمال انحصار متقابل
- سمافور mutex : اطمینان از تغییر متناسب rc
- متغیر سراسری rc : شمارش تعداد خوانندگان

- توسط if اول، به اولین حوالننده اجازه ورود داده می شود. (در صورتی که نویسنده ای فعال نباشد.)
- توسط if پایانی، بررسی می کنیم که اگر خواننده فعال دیگری وجود نداشته باشد، در صورت اینکه نویسنده بلوکه شده ای داشته باشیم، به آن اجازه داده شود.

- در مسئله خوانندگان و نویسندگان در حالتی که خوانندگان اولویت دارند، انتظار محدود رعایت نمی شود، چون امکان گرسنگی نویسندگان وجود دارد.