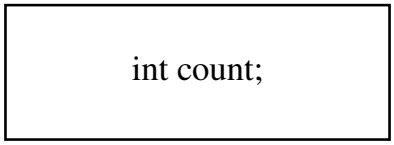
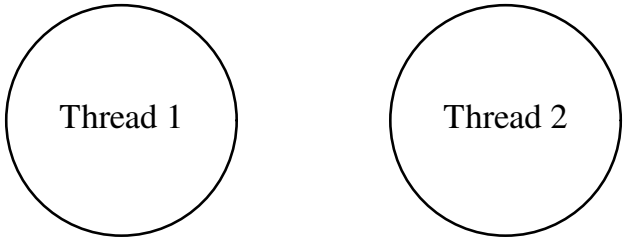
یادداشت‌های درس سیستم‌های عامل - بخش پنجم

در بخش‌های گذشته دو کاربرد حافظه‌ی مشترک را بررسی کرده‌ایم: الف) چند پردازه می‌توانند از سیستم عامل درخواست کنند که قسمتی از حافظه بین آنها مشترک باشند تا با کمک آن اطلاعات را انتقال دهند و ب) حافظه‌ی ریسه‌هایی که به یک پردازه تعلق دارند مشترک است. اما دسترسی همزمان به حافظه‌ی مشترک توسط دو پردازه یا دو ریسه مشکلات مهمی ایجاد می‌کند. در این بخش از درس، به این مشکل می‌پردازیم.

- یک سایت فروش اینترنتی ساده را در نظر بگیرید.





- برای سادگی فرض کنید که فقط یک کالا موجود هست و متغیر سراسری count تعداد موجودی این کالا را نشان می‌دهد.
- در فرآیند خرید کالا، اگر کالا موجود باشد مقدار آن کاهش می‌یابد و در غیر این صورت خطا گزارش می‌شود.

```
 1  int count;
 2
 3  void buy(void)
 4  {
 5       if (count == 0) {
 6            error("None left!\n");
 7            return;
 8       }
 9       count = count - 1;
10  }
```

- برای افزایش کارایی این سرور این فروشگاه، از ریسه‌ها استفاده می‌شوند: به ازای هر درخواست از سرور، یک ریسه ایجاد می‌شود تا به آن پاسخ دهد.

- فرض کنید مقدار متغیر count یک باشد و دو ریسه اجرای این تابع را شروع کنند.

```
 1  int count;
 2
 3  void buy(void)
 4  {
 5       if (count == 0) {
 6            error("None left!\n");
 7            return;
 8       }
 9       count = count - 1;
10  }
```

- برای سادگی فرض کنید هر خط توسط هر ریسه به صورت کامل اجرا شود و تعویض متن فقط بین خطها رخ می‌دهد (همیشه این طور نیست؛ هر خط توسط کامپایلر به تعدادی دستور پردازنده ترجمه می‌شود و بین این دستورات تعویض متن انجام می‌شود).

مثال - خروجی مطلوب

- جدول زیر یک حالت از اجرای این دو ریسه را نشان می‌دهد که روی یک پردازنده اجرا می‌شوند. ستون اول از چپ برچسب زمانی را نشان می‌دهد، ستون دوم شماره‌ی ریسه‌ی در حال اجرا را نشان می‌دهد، ستون سوم نشان می‌دهد چه خطی توسط ریسه اجرا می‌شود و ستون چهارم مقدار متغیر Count را نشان می‌دهد.

Time	Thread	Line	Count
1	1	5	1
2	1	9	0
3	2	5	0
4	2	6	0
5	2	7	0

```
 1  int count;
 2
 3  void buy(void)
 4  {
 5       if (count == 0) {
 6            error("None left!\n");
 7            return;
 8       }
 9       count = count - 1;
10  }
```

- در زمان یک ریسه‌ی اول خط پنجم را اجرا می‌کند و چون مقدار متغیر count یک است شرط این خط برقرار نیست و در زمان دو خط نهم اجرا می‌شود.
- ریسه‌ی دوم نیز خط پنجم، ششم و هفتم را اجرا می‌کند چون پس از خاتمه‌ی اجرای ریسه‌ی اول مقدار count صفر شد.
- همانطور که انتظار داشتیم، یکی از دو ریسه مقدار متغیر count را کاهش داد و ریسه‌ی دیگر خطا گزارش شد.

وضعیت رقابتی - مثال

- حالا فرض کنید دو ریسه به صورت زیر اجرا شوند.

Time	Thread	Line	Count
1	1	5	1
2	2	5	1
3	2	9	0
4	1	9	-1

```
 1  int count;
 2
 3  void buy(void)
 4  {
 5       if (count == 0) {
 6            error("None left!\n");
 7            return;
 8       }
 9       count = count - 1;
10  }
```

- در زمان یک، ریسه‌ی اول اجرا می‌شود، چون مقدار متغیر count برابر یک است شرط خط پنجم برقرار نیست و خط‌های ششم و هفتم اجرا نمی‌شوند.
- قبل از اینکه خط نهم اجرا شود تعویض متن رخ می‌دهد و ریسه‌ی دوم نیز خط پنجم را اجرا می‌کند. چون هنوز مقدار متغیر count یک هست، این ریسه هم خط‌های ششم و هفتم را اجرا نمی‌کند.
- در زمان سه، ریسه دوم خط نهم را اجرا می‌کند و در نتیجه مقدار متغیر count صفر می‌شود.
- در زمان چهار، ریسه اول نیز خط نهم را اجرا می‌کند و مقدار متغیر count به ۱- تغییر می‌کند.
- دسترسی همزمان دو ریسه به متغیر مشترک count موجب این خروجی ناخواسته شد.
- به این اتفاق وضعیت رقابتی گفته می‌شود.

وضعیت رقابتی و ناحیه‌ی بحرانی

- وضعیت رقابتی (Race Condition) به وضعیتی گفته می‌شود که: ترتیب اجرای پردازه‌ها یا ریسه‌ها موجب خروجی متفاوتی شود به طوری که در برخی از حالت‌ها، خروجی نامناسبی تولید شود.
- وضعیت رقابتی نتایج نامطلوبی را در برنامه‌ها ایجاد می‌کند و باید از آن جلوگیری کرد.
- به قسمت‌هایی از کد که به متغیر مشترک دسترسی دارند اصطلاحا ناحیه‌ی بحرانی Critical Section گفته می‌شود.
- در مثال قبل، خط پنجم تا نهم یک ناحیه‌ی بحرانی است چون به متغیر مشترک count دسترسی دارند.

```
 1  int count;
 2
 3  void buy(void)
 4  {
 5       if (count == 0) {
 6            error("None left!\n");
 7            return;
 8       }
 9       count = count - 1;
10  }
```

مثالی دیگر

- در مثال زیر مدیریت موجودی یک حساب را نشان می‌دهد.

```
 1  int balance;
 2
 3  void deposit(int x)
 4  {
 5       int n = balance + x;
 6       balance = n;
 7  }
 8  void withdraw(int x)
 9  {
10       int n;
11       if (balance < x) {
12            error("Account balance too low\n");
13            return;
14       }
15       n = balance - x;
16       balance = n;
17  }
```

- متغیر سراسری balance موجودی حساب را نشان می‌دهد و تابع deposit آن را افزایش و تابع withdraw آن را کاهش می‌دهد.
- اگر موجودی کافی نباشد، در برداشت مبلغ از حساب خطا گزارش می‌شود.
- دو ناحیه‌ی بحرانی در این قطعه کد وجود دارند: الف) خط پنجم و ششم در تابع deposit() و ب) خط یازدهم تا شانزدهم در تابع withdraw().

وضعیت رقابتی برای مثال دوم

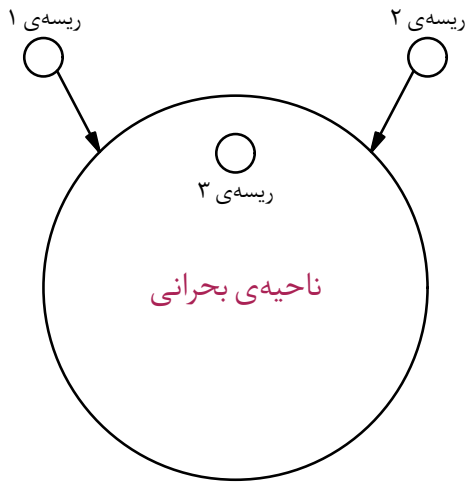
- فرض کنید دو ریسه به صورت زیر اجرا شوند و مقدار اولیه‌ی متغیر balance برابر ۵۰ باشد.
- ریسه‌ی اول تابع withdraw(10) و ریسه‌ی دوم تابع deposit(20) را فراخوانی می‌کند.

Time	Thread	Line	Balance
1	1	11	50
2	1	15	50
3	2	5	50
4	2	6	70
5	1	16	40

```
 1  int balance;
 2
 3  void deposit(int x)
 4  {
 5       int n = balance + x;
 6       balance = n;
 7  }
 8  void withdraw(int x)
 9  {
10       int n;
11       if (balance < x) {
12            error("Account balance too low\n");
13            return;
14       }
15       n = balance - x;
16       balance = n;
17  }
```

- پس از اینکه ریسه‌ی اول خط یازدهم و پانزدهم را اجرا کرد مقدار متغیر balance پنجاه و مقدار متغیر محلی n در تابع withdraw ۴۰ خواهد بود.
- در زمان سه، ریسه‌ی دوم اجرا می‌شود و موجودی را در زمان چهار به ۷۰ تغییر می‌دهد.
- در زمان پنج، ریسه‌ی اول کارش را ادامه می‌دهد و مقدار متغیر محلی n یعنی ۴۰ را روی balance می‌نویسد.
- این خروجی اشتباه است چون انتظار داشتیم مقدار balance پس از اجرای این دو ریسه ۶۰ می‌شد.
- تمرین: وضعیت رقابتی را در یک مثال فقط با فراخوانی تابع deposit توسط دو ریسه نشان دهید.

- هر راه حل مناسبی که برای حل مسئله‌ی ناحیه‌ی بحرانی ارائه می‌شود باید سه شرط زیر را داشته باشد.
- الف) انحصار متقابل (Mutual Exclusion): در هر لحظه فقط یک ریسه می‌تواند در داخل ناحیه‌ی بحرانی مرتبط باشد.



- ب) پیشرفت (Progress): اگر چند ریسه منتظر ورود به ناحیه‌ی بحرانی باشند، باید یکی از آنها برای ورود انتخاب شود. ورود یکی از ریسه‌های منتظر نباید وابسته به کاری باشد که ریسه‌ی دیگری که منتظر نیست انجام دهد.
- ج) انتظار محدود (Bounded Waiting): اگر ریسه‌ای منتظر ورود به ناحیه‌ی بحرانی باشد باید بعد از تعداد محدودی ورود و خروج سایر ریسه‌ها به ناحیه‌ی بحرانی، نوبتش بشود و وارد ناحیه‌ی بحرانی‌اش بشود. در انتظار محدود زمان اهمیت ندارد و تعداد ورود و خروج سایر ریسه‌ها مهم است.

شرایط راه حل‌های ناحیه‌ی بحرانی

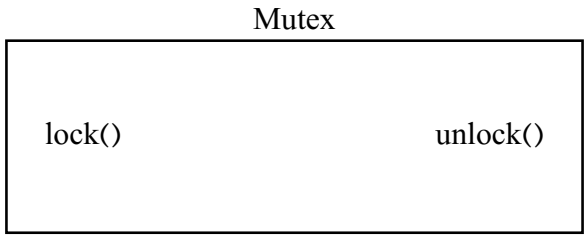
- هر راه حلی که برای ناحیه‌ی بحرانی ارائه می‌شود باید سه شرطی که در صفحه‌ی قبل بیان شد را داشته باشد.
- بعدا خواهیم دید که در عمل گاهی شرط سوم نادیده گرفته می‌شود اگر احتمال انتظار زیاد یک ریسه کم باشد.
- اهمیت شرط اول واضح است: برای حل مشکل لازم است از دسترسی همزمان ریسه‌ها به متغیرها جلوگیری کنیم و برای این کار اگر یکی از ریسه‌ها در حال دسترسی به متغیرهای مشترک باشد، سایر ریسه‌ها نباید این کار را انجام دهند.

می‌توانیم کد دارای ناحیه‌ی بحرانی را به صورت زیر تقسیم کنیم.

```
[remainder section: سایر قسمت‌های کد]

[entry section: ورود به ناحیه‌ی بحرانی]

[critical section: ناحیه‌ی بحرانی]

[exit section: خروج از ناحیه‌ی بحرانی]

[remainder section: سایر قسمت‌های کد]
```

- در یک راه حل ناحیه‌ی بحرانی دستوراتی را به دو قسمت ورود و خروج از ناحیه‌ی بحرانی اضافه می‌کنیم.
- به مدیریت دسترسی‌های همزمان ریسه‌ها (یا پردازه‌هایی با حافظه‌ی مشترک بین آنها) و مدیریت ترتیب اجرای آنها همگام‌سازی (Synchronization) گفته می‌شود.
- در ادامه چند ابزار برای این کار خواهیم دید.

- ابزار اولی که برای حل مسئله‌ی ناحیه‌ی بحرانی بررسی می‌کنیم قفل‌های Mutex هستند.
- کلمه‌ی Mutex مخفف Mutual Exclusion هست.



- قفل‌های Mutex دو تابع دارند: الف) تابع ()acquire یا ()lock: برای قفل کردن آن و ب) تابع ()release یا ()unlock: برای باز کردن آن.
- وقتی که یک ریسه تابع ()lock یک Mutex را فراخوانی می‌کند آن را در اختیار می‌گیرد (قفل می‌کند).
- پس از آن هر ریسه‌ی دیگری که تابع ()lock را فراخوانی کند Block (منتظر) می‌شود تا وقتی که ریسه‌ی در اختیار دارنده‌ی Mutex قفل را با فراخوانی تابع ()unlock آزاد کند.
- پس از آن اجرای یکی از ریسه‌ها منتظر ادامه می‌یابد و قفل را در اختیار می‌گیرد.

- فرض کنید یک متغیر Mutex دو مقدار دارد: صفر یعنی آزاد (باز) و یک یعنی بسته (قفل).
- شبه کد زیر عملکرد تابع acquire را نشان می‌دهد.
- اخطار: دقت کنید که این شبه کد فقط برای توضیح عملکرد قفل است و پیاده‌سازی نیست.

```
1  void acquire(Mutex *lock)
2  {
3       while (*lock == 1)
4            ;
5       *lock = 1;
6  }
```

- تابع زیر هم رفتار تابع release را نشان می‌دهد.

```
7  void release(Mutex *lock)
8  {
9       *lock = 0;
10 }
```

استفاده از قفل در مثال اول

- برای از بین بردن وضعیت رقابتی با کمک قفل‌های Mutex تغییرات زیر در مثال اول ایجاد شده است:

```
 1  int count;
 2  Mutex lock;
 3
 4  void buy(void)
 5  {
 6       acquire(&lock);
 7       if (count == 0) {
 8            error("None left!\n");
 9            release(&lock);
10            return;
11       }
12       count = count - 1;
13       release(&lock);
14  }
```

- یک قفل با نام lock اضافه شده است (خط دوم). دقت کنید که قفل باید در حافظه‌ی مشترک باشد.
- قبل از ورود به ناحیه‌ی بحرانی قفل گرفته می‌شود (خط ششم) و پس از خروج از ناحیه‌ی بحرانی قفل آزاد می‌شود (خط‌های نهم و سیزدهم).
- بنابراین ناحیه‌ی بحرانی به صورت انحصار متقابل اجرا می‌شود.

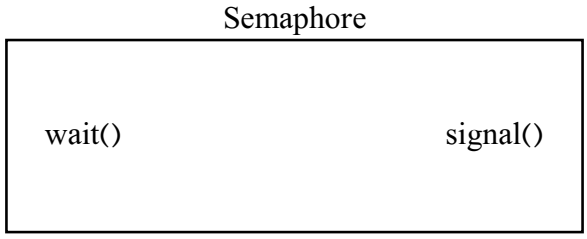
استفاده از قفل در مثال دوم

- برای از بین بردن وضعیت رقابتی با کمک قفل‌های Mutex تغییرات زیر در مثال دوم ایجاد شده است:

```
 1  int balance;
 2  Mutex balance_lock;
 3
 4  void deposit(int x)
 5  {
 6       acquire(&balance_lock);
 7       int n = balance + x;
 8       balance = n;
 9       release(&balance_lock);
10  }
11  void withdraw(int x)
12  {
13       int n;
14       acquire(&balance_lock);
15       if (balance < x) {
16            error("Account balance too low\n");
17            release(&balance_lock);
18            return;
19       }
20       n = balance - x;
21       balance = n;
22       release(&balance_lock);
23  }
```

- یک قفل با نام balance_lock در حافظه‌ی تعریف شده است (خط دوم).
- قبل از ورود به ناحیه‌ی مشترک قفل گرفته می‌شود (خط ششم و چهاردهم).
- پس از خروج از ناحیه‌ی بحرانی قفل آزاد می‌شود (خط نهم، هفدهم و بیست و دوم).

- سمافور Semaphore ابزار دومی است که برای همگام‌سازی معرفی می‌شود.
- یک سمافور یک مقدار عددی دارد که به صورت مستقیم نمی‌توان به آن دسترسی داشت.
- سمافور دو تابع دارد: الف) تابع signal() برای افزایش مقدار سمافور و ب) تابع wait() برای کاهش مقدار آن.



- ویژگی مهم سمافور: اگر مقدار عددی سمافور صفر باشد، تابع wait() منتظر می‌شود تا مقداری بیشتر از صفر بگیرد تا پس از کاهش مقدار سمافور، مقدارش منفی نشود.

- شبه کد زیر عملکرد تابع signal را نشان می‌دهد.
- اخطار: دقت کنید که این شبه کد فقط برای توضیح عملکرد سمافور است و پیاده‌سازی نیست.
- ریسه‌ای که تابع signal مقدار سمافور را افزایش می‌دهد.

```
1  void signal(Semphore sem)
2  {
3       sem = sem + 1;
4  }
```

- تابع زیر هم رفتار تابع wait را نشان می‌دهد.

```
5  void wait(Semaphore sem)
6  {
7       while (sem == 0)
8            ;
9       sem = sem - 1;
10 }
```

- فرض کنید تابع f() توسط ریسه‌ی اول و تابع g() توسط ریسه‌ی دوم فراخوانی می‌شود.

Thread #1	Thread #2
... f(); ...	... g(); ...

- به کمک سمافور می‌توان کاری کرد که g() وقتی فراخوانی شود که فراخوانی f() خاتمه یافته باشد.

Thread #1	Thread #2

...
f();
signal(sem);
...
	...
	wait(sem);
	g();
	...

- برای این کار مقدار اولیه‌ی سمافور sem چند باشد؟
- تمرین: اگر مقدار اولیه‌ی سمافور یک باشد چه اتفاقی می‌افتد؟ اگر دو باشد چطور؟

استفاده از سمافور در مثال اول

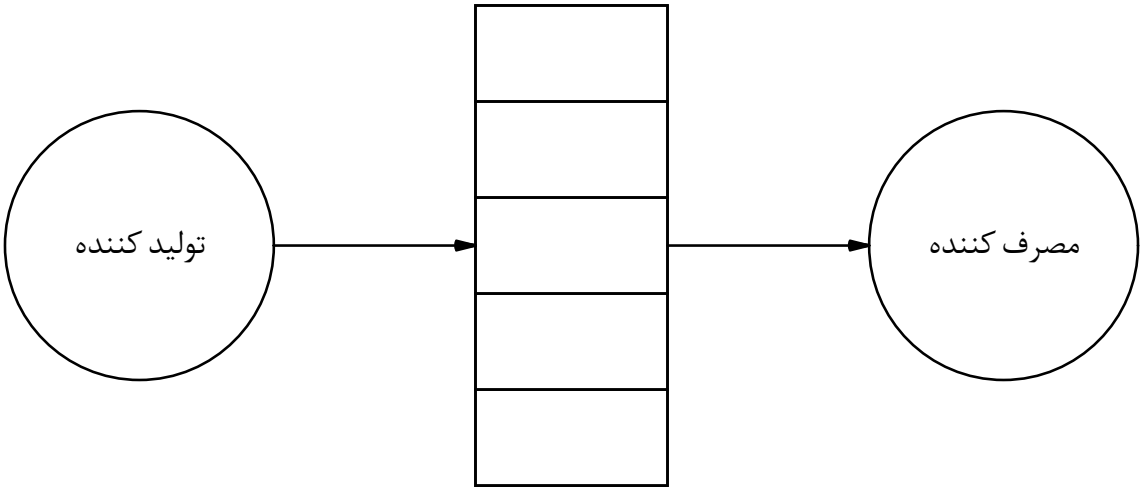
- برای از بین بردن وضعیت رقابتی با کمک قفل‌های Semaphore تغییرات زیر در مثال اول این بخش از درس ایجاد شده است:

```
 1  int count;
 2  Semaphore sem = 1;
 3
 4  void buy(void)
 5  {
 6       wait(sem);
 7       if (count == 0) {
 8            error("None left!\n");
 9            signal(sem);
10            return;
11       }
12       count = count - 1;
13       signal(sem);
14  }
```

- یک سمافور با نام sem در حافظه‌ی مشترک تعریف شده است (خط دوم).
- قبل از ورود به ناحیه‌ی بحرانی تابع wait(sem) فراخوانی می‌شود (خط ششم) و پس از خروج از ناحیه‌ی بحرانی تابع signal(sem) فراخوانی می‌شود (خط‌های نهم و سیزدهم).
- وقتی که یک ریسه، پس از فراخوانی wait(sem) وارد ناحیه‌ی بحرانی شود، ریسه‌های دیگر منتظر می‌مانند چون مقدار سمافور صفر می‌شود.
- وقتی ریسه‌ی داخل ناحیه‌ی بحرانی تابع signal(sem) را فراخوانی می‌کند، یکی از ریسه‌ها منتظر وارد می‌شوند.
- بنابراین ناحیه‌ی بحرانی به صورت انحصار متقابل اجرا می‌شود.
- تمرین: اگر مقدار اولیه‌ی سمافور sem برابر صفر باشد چه اتفاقی می‌افتد؟
- تمرین: اگر مقدار اولیه‌ی سمافور sem برابر دو باشد چه اتفاقی می‌افتد؟

- در ادامه چند مسئله‌ی کلاسیک در مورد همگام‌سازی معرفی می‌شوند.
- حالتی از این مسائل در کاربردهای زیادی مطرح می‌شود.

- مسئله‌ی تولید کننده و مصرف کننده را قبلا دیده‌ایم.
- فرض کنید از حافظه‌ی مشترک برای انتقال داده‌های تولید شده از تولید کننده به مصرف کننده استفاده شود.



Bounded Buffer

- اندازه‌ی حافظه‌ای که برای انتقال داده‌ها استفاده می‌شود (بافر) محدود است (N).

تولید کننده و مصرف کننده: راه حل

- فرض کنید از یک ساختمان داده‌ی صف برای بافر استفاده شود.

```
 1  void producer(void) {
 2       while (1) {
 3            Item x;
 4            # produce item x...
 5            queue.append(x);
 6       }
 7  }
 8  void consumer(void) {
 9       while (1) {
10            Item x;
11            x = queue.remove();
12            # consume item x...
13       }
14  }
```

- باید به سه نکته‌ی مهم زیر در یک راه حل درست برای این مسئله توجه کنیم.
- الف) چون هم تولید کننده و هم مصرف کننده به بافر دسترسی دارند باید این دسترسی به صورت انحصار متقابل انجام شود (رنگ در کد: قرمز).
- ب) اگر تولید کننده داده‌ها را با سرعت بیشتری نسبت به مصرف کننده تولید کند بافر پر می‌شود و تولید کننده باید منتظر شود (رنگ در کد: زرد).
- ج) اگر مصرف کننده داده‌ها را با سرعت بیشتری نسبت به تولید کننده مصرف کند بافر خالی می‌شود و مصرف کننده باید منتظر شود (رنگ در کد: سبز).

تولید کننده و مصرف کننده: راه حل

- از متغیرهای مشترک زیر برای مسئله‌ی تولید کننده و مصرف کننده استفاده می‌شود.

```
1  Queue queue;
2  Semaphore queue_lock = 1;
3  Semaphore full = 0;
4  Semaphore empty = N;
```

- متغیر queue_lock برای محافظت از دسترسی همزمان به بافر است.
- متغیر full برای انتظار مصرف کننده است: مقدار این سمافور تعداد خانه‌های پر بافر را نشان می‌دهد.
- متغیر empty برای انتظار تولید کننده است: مقدار این سمافور تعداد خانه‌های خالی بافر را نشان می‌دهد.
- در نتیجه مقدار اولیه‌ی سمافور full صفر و مقدار اولیه‌ی سمافور empty برابر N است.

تولید کننده و مصرف کننده: راه حل

- تولید کننده به صورت متناوب یک داده تولید می‌کند و به بافر اضافه می‌کند.

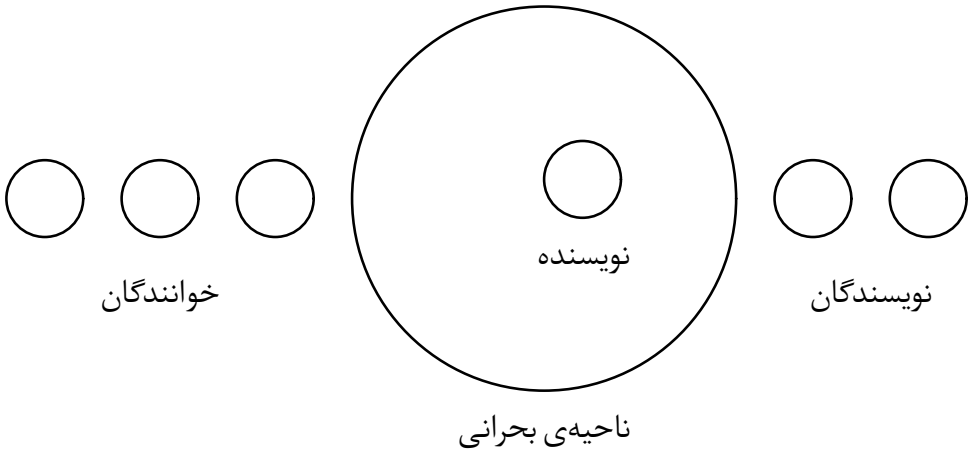
```
 5  void producer(void) {
 6       while (1) {
 7            Item x;
 8            # produce item
 9            ...
10            wait(empty);
11            wait(queue_lock);
12            queue.append(x);
13            signal(queue_lock);
14            signal(full);
15       }
16  }
```

- تولید کننده به صورت متناوب یک داده را از بافر بر می‌دارد و مصرف می‌کند.

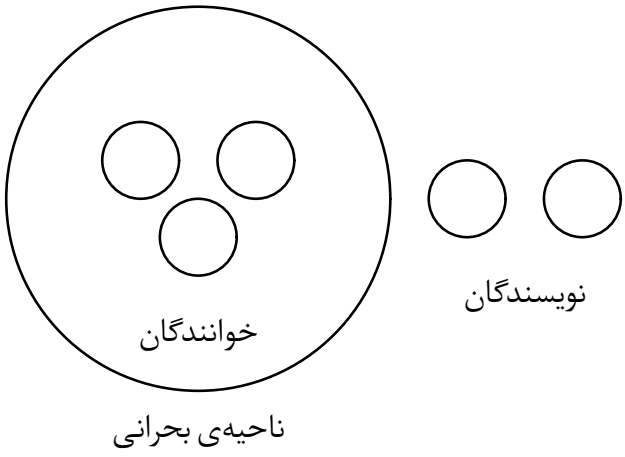
```
17  void consumer(void) {
18       while (1) {
19            Item x;
20            wait(full);
21            wait(queue_lock);
22            x = queue.remove();
23            signal(queue_lock);
24            signal(empty);
25            # consume item
26            ...
27       }
28  }
```

- برقراری شرط الف: از سمافور queue_lock برای ایجاد انحصار متقابل استفاده می‌شود (قرمز).
- ب) اگر بافر پر شود، مقدار سمافور empty به صفر تغییر می‌کند و تولید کننده با فراخوانی wait(empty) منتظر می‌شود (زرد).
- ج) اگر بافر خالی شود، مقدار سمافور full به صفر تغییر می‌کند و مصرف کننده با فراخوانی wait(full) منتظر می‌شود (سبز).

- خواندن یک قسمت از حافظه توسط چند ریسه به صورت همزمان مشکلی ایجاد نمی‌کند.
- وقتی دسترسی همزمان منجر به وضعیت رقابتی می‌گردد که حداقل یکی از ریسه‌ها حافظه را تغییر دهد.
- فرض کنید منبعی (مثل یک متغیر مشترک، یک پایگاه داده یا یک فایل) وجود دارد که به دو شکل استفاده می‌شود: الف) برخی از ریسه‌ها آن را می‌خوانند و ب) برخی به آن می‌نویسند.
- اگر یک نویسنده به منبع دسترسی دارد، هیچ ریسه‌ی دیگری نباید به آن دسترسی داشته باشد (شکل زیر).



- اما خوانندگان می‌توانند به صورت همزمان به منبع دسترسی داشته باشند (شکل زیر).



- در وقع برای بهبود عملکرد در این کاربردها، مطلوب است که خوانندگان بتوانند همزمان از منبع بخوانند.

- در راه حل پیشنهادی اجازه می‌دهیم چند خواننده با هم از منبع بخوانند ولی وقتی یک نویسنده از منبع می‌خواند هیچ خواننده یا نویسنده‌ای نباید از منبع بخواند.
- سه متغیر در حافظه‌ی مشترک تعریف می‌کنیم.

```
Semaphore lock;
int rcount = 0;
Semaphore rlock;
```

- سمافور lock برای محافظت از منبع است تا به صورت انحصار متقابل استفاده شود.
- متغیر rcount تعداد ریسه‌هایی که در حال خواندن از منبع هستند را نشان می‌دهد.
- متغیر rlock برای محافظت از دسترسی همزمان به متغیر rcount است.

- هر نویسنده برای هر بار دسترسی به منبع به صورت زیر عمل می‌کند.

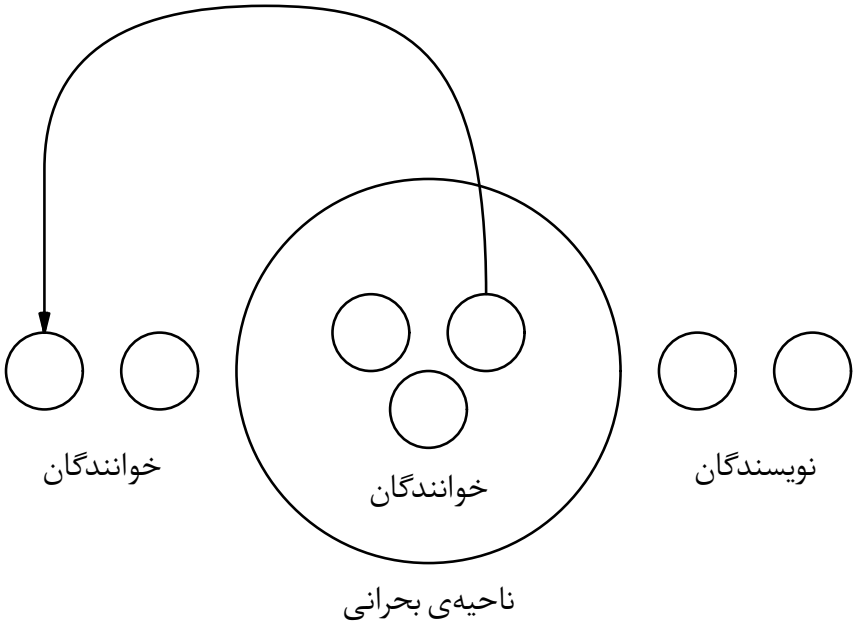
```
1       wait(lock);
2       ...       # write to resource
3       signal(lock);
```

- قبل از نوشتن به منبع قفل منبع را می‌گیرد و پس از نوشتن، قفل را آزاد می‌کند.
- ریسه‌های خواننده برای هر بار خواندن کد زیر را اجرا می‌کنند.

```
1       wait(rlock);
2       rcount = rcount + 1;
3       if (rcount == 1)
4            wait(lock);
5       signal(rlock);
6       ...       # read from resource
7       wait(rlock)
8       rcount = rcount - 1;
9       if (rcount == 0)
10           signal(lock);
11      signal(rlock);
```

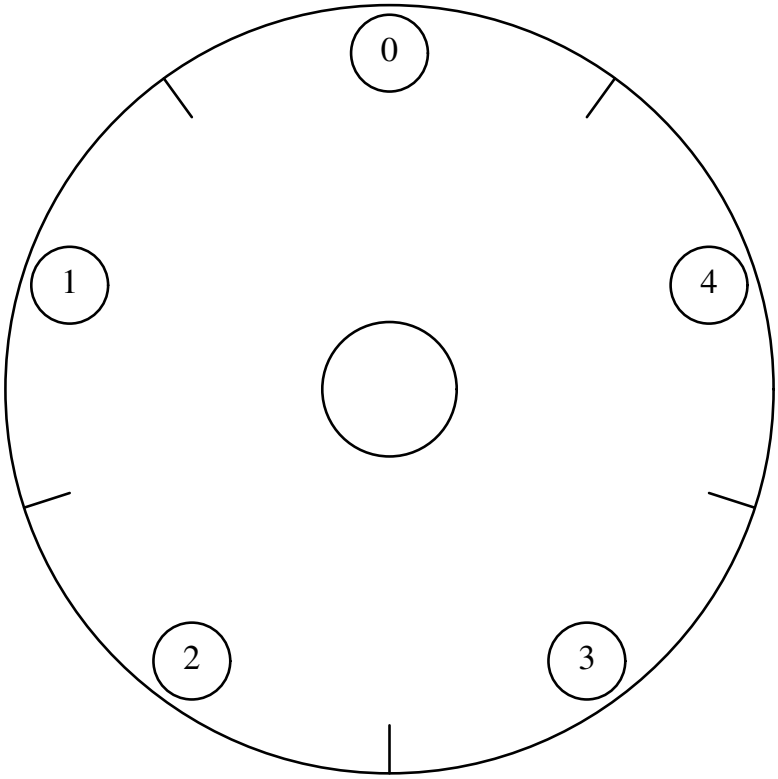
- ایده‌ی اصلی در این قطعه کد این است که اولین خواننده منبع را قفل می‌کند (خط سوم و چهارم).
- خوانندگان بعدی نیازی ندارند منبع را قفل کنند چون توسط اولین خواننده منبع قفل شده است و خوانندگان می‌توانند به صورت همزمان از منبع بخوانند.
- وقتی کار خواندن یک ریسه تمام شد، فقط آخرین خواننده منبع را آزاد می‌کند (خط دهم) تا نویسنده‌های منتظر یا خواننده‌های آینده بعدا منبع را در اختیار بگیرند.
- برای جلوگیری از وضعیت رقابتی برای متغیر rcount، خوانندگان برای دسترسی به آن قفل rlock را می‌گیرند.

- فرض کنید همواره قبل از اینکه آخرین خواننده کارش تمام شود، خوانندگان دیگری وارد شوند.



- در این صورت هیچگاه نوبت به نویسندگان نمی‌رسد چون آخرین خواننده قفل lock را آزاد می‌کند.
- به این اتفاق قحطی یا گرسنگی (Starvation) گفته می‌شود.
- به بیان دقیق‌تر قحطی وقتی رخ می‌دهد که: امکان داشته باشد یکی از استفاده کنندگان از منبعی منتظر بماند ولی استفاده کنندگان بیشمار دیگری بتوانند از منبع استفاده کنند و کارشان تمام شود.

- مسئله‌ی Dining Philosophers یکی از مثال‌های شناخته شده در مورد همروندی است.



- فرض کنید پنج فیلسوف دور یک میز گرد نشسته‌اند.
- فیلسوف‌ها با کمک دو چوب (Chopstick) غذا می‌خورند اما متأسفانه فقط پنج چوب روی میز وجود دارد: یک چوب بین دو فیلسوف کنار هم.
- بنابراین، اگر یک فیلسوف گرسنه باشد، چوب سمت چپ را بر می‌دارد، سپس چوب سمت راست را بر می‌دارد، غذا می‌خورد و سپس هر دو چوب را جای خودشان قرار می‌دهد.

- بنابراین برای هر منبع (چوب) یک قفل در حافظه‌ی مشترک در نظر می‌گیریم.

```
Mutex sticks[5];
```

- به ازای i از صفر تا چهار، هر فیلسوف (یک ریسه) الگوریتم زیر را انجام می‌دهد.

```
1       while (1) {
2           # think ...
3           acquire(&sticks[i]);
4           acquire(&sticks[(i + 1) % 5]);
5           # eat ...
6           release(&sticks[i]);
7           release(&sticks[(i + 1) % 5]);
8       }
```

- اگر فیلسوف i-ام بخواهد غذا بخورد ابتدا چوب سمت چپ را در اختیار می‌گیرد، سپس چوب سمت راست را در اختیار می‌گیرد، سپس غذا می‌خورد و در پایان چوب‌ها را آزاد می‌کند.
- استفاده از عملگر باقی‌مانده به ۵ (خط چهارم و هفتم) برای این است که فیلسوف چهارم باید چوب چهار و صفرم را بردارد.
- فکر کنید اگر هر پنج فیلسوف تقریبا همزمان گرسنه شوند چه اتفاقی می‌افتد؟
- فرض کنید هر ریسه خط سوم را اجرا کند. چه اتفاقی می‌افتد؟

- فرض کنید همه‌ی ریسه‌ها خط سوم را اجرا کنند؛ در نتیجه چوب i-ام در اختیار فیلسوف i-ام قرار می‌گیرد.
- سپس همه‌ی ریسه‌ها منتظر چوب بعدی ((i + 1) % 5) خواهند بود.
- اما چون بعدی در اختیار فیلسوف متناظرش است، هر فیلسوف منتظر آزاد شدن چون توسط فیلسوف بعدی می‌شود.
- بنابراین، هر پنج فیلسوف منتظر فیلسوف بعدی هستند و هیچ یک از آنها چوب خودش را آزاد نمی‌کند و تا ابد در این وضعیت می‌مانند.
- به این رخداد بن‌بست (Deadlock) می‌گویند: وضعیتی که تعدادی ریسه‌ی (یا پردازه) منتظر وجود دارند که هر یک از آنها منتظر یکی دیگر از همین ریسه‌های منتظر است.
- در آینده به صورت دقیق‌تری بن‌بست را بررسی می‌کنیم.