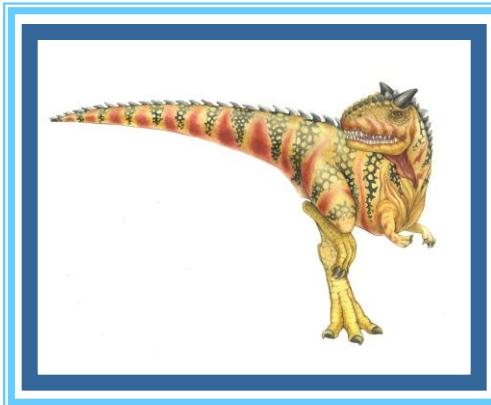


# فصل ۶: ابزار همگام سازی





# سرفصل

- پیش زمینه
- مشکل بخش بحرانی
- راه حل پترسون
- پشتیبانی سخت افزار برای همگام سازی
- قفل های درگیری متقابل (Mutex Locks)
- سیمافورها (Semaphores)
- مانیتورها (Monitors)
- سرزندگی (Liveness)
- ارزیابی



## اهداف

- در این بخش موارد زیر شرح داده می شود:
- مشکل بخش بحرانی و نمایش یک وضعیت مسابقه (Race Condition)
- راه حل های سخت افزاری برای مشکل بخش بحرانی با استفاده از موانع حافظه (Memory Barriers)، عملیات مقایسه و تعویض (Compare-and-Swap) و متغیرهای اتمی (Atomic Variables)
- چگونگی استفاده از قفل های انحصار متقابل، سمافورها، مانیتورها و متغیرهای شرطی برای حل مشکل بخش بحرانی
- ارزیابی ابزارهایی که مشکل بخش بحرانی را در سناریوهای رقابت کم، متوسط و زیاد حل می کنند.



## پیش زمینه

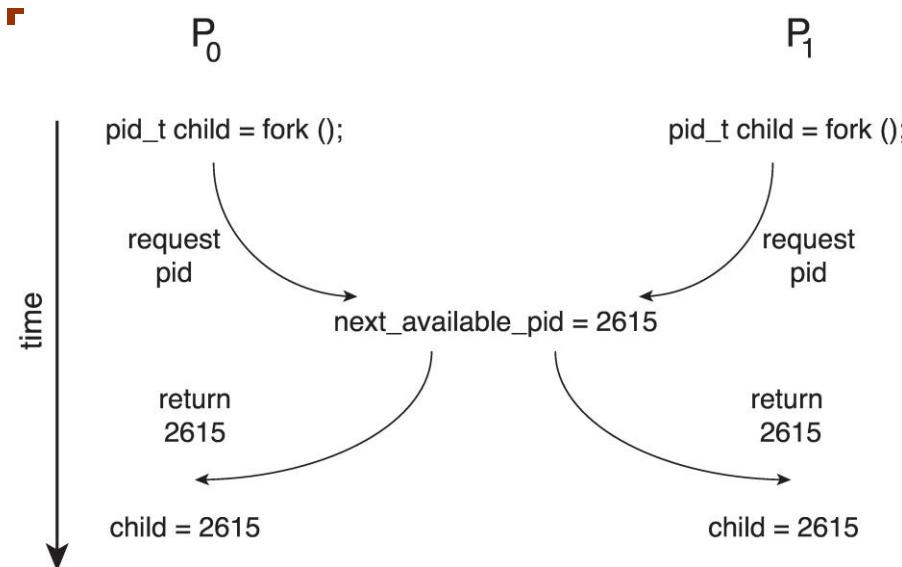
- فرایندها می توانند به طور همزمان اجرا شوند.
- ممکن است در هر زمانی قطع شوند و اجرای آنها به طور کامل به پایان نرسد.
- دسترسی همزمان به داده های مشترک ممکن است منجر به ناهمگونی داده شود
- حفظ انسجام داده به مکانیزم هایی برای اطمینان از اجرای منظم فرآیندهای همکاری نیاز دارد.
- در فصل ۴، مشکل را زمانی نشان دادیم که مسئله بافر محدود را با استفاده از یک شمارنده در نظر گرفتیم که به طور همزمان توسط تولید کننده و مصرف کننده به روز می شود. این امر منجر به وضعیت مسابقه (Race Condition) شد.



# شرط مسابقه

■ فرآیندهای  $P_0$  و  $P_1$  با استفاده از فراخوانی سیستمی (`fork()`) فرآیندهای فرزند ایجاد می‌کنند.

■ وضعیت مسابقه بر روی متغیر `next_available_pid` که نشان دهنده شناسه فرآیند (`pid`) در دسترس بعدی است، رخ می‌دهد.



■ اگر مکانیزمی برای جلوگیری از دسترسی همزمان فرآیندهای  $P_0$  و  $P_1$  به متغیر `next_available_pid` وجود نداشته باشد، یک شناسه فرآیند (`pid`) یکسان می‌تواند به دو فرآیند مختلف اختصاص یابد!



## مسئله ناحیه بحرانی

- سیستم با  $n$  فرآیند  $\{P_0, P_1, \dots, P_{n-1}\}$  را در نظر بگیرید.
- هر فرآیند دارای ناحیه بحرانی از کد است که شامل دسترسی و تغییر متغیرهای مشترک، بهروزرسانی جدول، نوشتن فایل و غیره می‌باشد.
- هنگامی که یک فرآیند در ناحیه بحرانی است، هیچ فرآیند دیگری نمی‌تواند در ناحیه بحرانی خود باشد.
- مسئله ناحیه بحرانی طراحی یک پروتکل برای حل این موضوع است.
- هر فرآیند باید در بخش ورود، مجوز ورود به بخش بحرانی را درخواست کند، پس از ناحیه بحرانی ممکن است بخش خروج را دنبال کند و سپس ناحیه باقیمانده را اجرا نماید.



# بخش بحرانی

- General structure of process  $P_i$

```
while (true) {
```

```
    entry section
```

ناحیه ورودی

```
        critical section
```

ناحیه بحرانی

```
    exit section
```

ناحیه خروجی

```
        remainder section
```

```
}
```



# بخش بحرانی (ادامه)

■ راه کار مسئله ناحیه بحرانی باید این نیازمندهای را برآورده کند.

## ۱- قاعده انحصار متقابل (Mutual Exclusion)

▶ اگر فرآیند  $P_i$  در حال اجرای بخش بحرانی خود است، پس هیچ فرآیند دیگری نمی تواند بخش بحرانی خود را اجرا کند.

## ۲- پیشرفت (Progress)

▶ اگر پردازشی در قسمت باقی مانده خود باشد در تصمیم گیری اینکه کدام پردازش وارد بخش بحرانی شود شرکت نمی کند. یعنی هیچ پردازشی نباید خارج از ناحیه بحرانی خود امکان بلوکه کردن پردازش‌های دیگر را داشته باشد.

## ۳- انتظار محدود (Bounded Waiting)

▶ یک برنامه منتظر برای ورود به ناحیه بحرانی نباید به طور نامحدود منتظر بماند.

۴- هر پردازشی با سرعت غیر صفر اجرا می شود ولی هیچ فرضی در مورد سرعت نسبی  $n$  فرآیند وجود ندارد.



## راه کار مبتنی بر وقفه

- بخش ورود: قطع کردن وقفه ها (Disable Interrupts)
- بخش خروج: فعال کردن وقفه ها (Enable Interrupts)
  - آیا این مشکل را حل خواهد کرد؟
  - چه اتفاقی می افتد اگر بخش بحرانی کدی باشد که برای یک ساعت اجرا شود؟
  - آیا برخی فرآیندها ممکن است گرسنه بمانند - هرگز وارد بخش بحرانی خود نشوند؟
  - اگر دو پردازنده وجود داشته باشد چه؟



# راه کار نرم افزاری ۱

- راه حل دو فرآیند
- فرض کنید دستورالعمل‌های بارگذاری و ذخیره‌سازی زبان ماشین اتمیک هستند؛ یعنی قابل وقفه نیستند. دو فرآیند یک متغیر مشترک دارند:
  - **int turn;**
- تغییر **turn** نشان می‌دهد که نوبت کدام فرآیند است تا وارد بخش بحرانی شود.
- مقدار اولیه **turn** برابر با ۰ تنظیم می‌شود.



# Algorithm for Process $P_i$

```
while (true){
```

```
    while (turn == j);
```

```
    /* critical section */
```

```
    turn = j;
```

```
    /* remainder section */
```

```
}
```



Process P0:  
while (true){

    while (turn == 1);

    /\* critical section \*/

    turn = 1;

    /\* remainder section \*/

}

Process P1:  
while (true){

    while (turn == 0);

    /\* critical section \*/

    turn = 0;

    /\* remainder section \*/

}

مقدار اولیه turn برابر با یک بوده و p1 به ناحیه بحرانی می‌رود در حالی پردازش دیگر در حلقه وایل منتظر هست. پس از خروج پردازش p1 متغیر turn برابر 0 می‌شود. حال p0 وارد ناحیه بحرانی شده و خارج می‌شود. مقدار 1 turn شده و پردازش p1 وارد ناحیه بحرانی می‌شود و در حین خروج turn=0 می‌شود. اگر فرض کنیم پردازش p0 در ناحیه باقیمانده به صورت طولانی باشد. شرط پیشرفت برای p1 نقض خواهد شد. زیرا p1 پس از تغییر turn=0 دیگر نمی‌تواند وارد ناحیه بحرانی خود شود.



# صحبت راه کار نرم افزاری

## ■ انحصار متقابل حفظ می شود:

فرآیند  $Pi$  فقط در صورتی وارد بخش بحرانی می شود که :

$$\text{turn} = i \quad \circ$$

و  $\text{turn}$  نمی تواند همزمان  $0$  و  $1$  باشد.

■ اما در مورد الزام پیشرفت (Progress) و انتظار محدود (Bounded-Waiting) چه؟



## راه حل بهبود یافته پترسون

- راه حل بهبود یافته پترسون (راه کار نرم افزاری)
- فرض کنید دستورالعمل‌های بارگذاری و ذخیره‌سازی زبان ماشین اتمی هستند؛ یعنی قابل وقفه نیستند. دو فرآیند دو متغیر مشترک دارند：
  - `int turn;`
  - `boolean flag[2]`
- متغیر `turn` نشان می‌دهد که نوبت کدام فرآیند است تا وارد بخش بحرانی شود. آرایه `flag` برای نشان دادن آمادگی یک فرآیند برای ورود به بخش بحرانی استفاده می‌شود `flag[i] = true` نشان می‌دهد که فرآیند `Pi` آماده است.



# Algorithm for Process $P_i$

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



Process P0:  
**while** (true){

```
flag[0] = true;  
turn = 1;  
while (flag[1] && turn == 1)  
    ;
```

/\* critical section \*/

```
flag[0] = false;  
  
/* remainder section */
```

}

Process P1:  
**while** (true){

```
flag[1] = true;  
turn = 0;  
while (flag[0] && turn == 0)  
    ;
```

/\* critical section \*/

```
flag[1] = false;  
  
/* remainder section */
```

}



# صحت راه کار پترسون

■ قابل اثبات است که سه الزام بخش بحرانی رعایت می شوند:

**1. درگیری متقابل حفظ می شود:**

. فرآیند  $P_i$  فقط در صورتی وارد بخش بحرانی می شود که :

turn = i یا flag[j] = false یا اینکه

**2. الزام پیشرفت برآورده می شود.**

**3. الزام انتظار محدود رعایت می شود.**



## راه کار پترسون و معماری مدرن

- اگرچه راه حل پترسون برای نشان دادن یک الگوریتم مفید است، اما تضمینی برای کار کرد آن روی معماری های مدرن وجود ندارد.
- برای بهبود عملکرد، پردازنده ها و/یا کامپایلرها ممکن است عملیات هایی که وابستگی ندارند را مرتب سازی مجدد (ترتیب دستورات جابجا شود) کنند.
- در ک اینکه چرا این راه حل کار نمی کند برای درک بهتر شرایط مسابقه (Race Conditions)
- برای تک-رشته ای (Single-Threaded) این موضوع مشکلی ندارد زیرا نتیجه همیشه یکسان خواهد بود. برای چند رشته ای (Multithreaded) مرتب سازی مجدد ممکن است نتایج ناسازگار یا غیرمنتظره ایجاد کند.



# مثال معماري مدرن

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

```
while (!flag) ;  
print x
```

- Thread 2 performs

```
x = 100;  
flag = true
```

- What is the expected output?

100



## مثال معماری مدرن (ادامه)

- با این حال، از آنجایی که متغیرهای `flag` و `x` مستقل از یکدیگر هستند، دستورالعمل‌های زیر:

```
flag = true;  
x = 100;
```

- ممکن است برای رشته ۲ مرتب‌سازی مجدد شوند. اگر این اتفاق بیفتد، خروجی ممکن است • شود!

Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

Thread 1 performs

```
while (!flag)  
;  
print x
```

Thread 2 performs

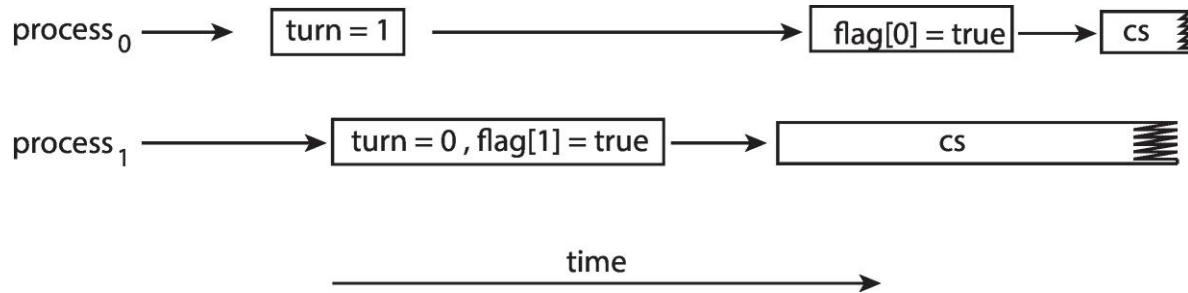
```
flag = true  
x = 100;
```

What is the expected output?



# Peterson's Solution Revisited

## اثر مرتب‌سازی مجدد دستورالعمل در راه حل پترسون



- این امر به هر دو فرآیند اجازه می‌دهد تا همزمان در بخش بحرانی خود باشند!
- برای اطمینان از اینکه راه حل پترسون به درستی روی معماری کامپیوتر مدرن کار می‌کند، باید از سد حافظه (Memory Barrier) استفاده کنیم.



## سد حافظه

- مدل‌های حافظه می‌توانند به دو دسته تقسیم شوند:
  - **مرتب‌شده قوی** - (Strongly Ordered) جایی که یک تغییر در حافظه توسط یک پردازنده بلافصله برای تمام پردازنده‌های دیگر قابل مشاهده است.
  - **مرتب‌شده ضعیف** - (Weakly Ordered) جایی که یک تغییر حافظه توسط یک پردازنده ممکن است بلافصله برای تمام پردازنده‌های دیگر قابل مشاهده نباشد.
- **سد حافظه** (Memory Barrier) دستورالعملی است که هرگونه تغییر در حافظه را مجبور می‌کند به تمام پردازنده‌های دیگر منتقل (قابل مشاهده) شود.



# دستورالعمل سد حافظه

- هنگامی که دستورالعمل سد حافظه اجرا می‌شود، سیستم اطمینان می‌دهد که همه بارگذاری‌ها و ذخیره‌سازی‌ها (مثلایک متغیر) قبل از هر بارگذاری یا ذخیره‌سازی بعدی انجام شده و قابل مشاهده توسط پردازنده‌های دیگر هستند.
- بنابراین، حتی اگر دستورالعمل‌ها مرتب‌سازی مجدد شوند، سد حافظه تضمین می‌کند که عملیات ذخیره‌سازی در حافظه تکمیل شده و برای پردازنده‌های دیگر قبل از انجام عملیات بارگذاری یا ذخیره‌سازی بعدی قابل مشاهده است.



## مثال سد حافظه

- بازگشت به مثال اسلایدهای ۱۷، ۱۸ تا ۲۰
- برای اطمینان از اینکه خروجی رشته ۱ برابر با ۱۰۰ باشد، می‌توانیم یک سد حافظه به دستورالعمل‌های زیر اضافه کنیم:
  - اکنون رشته ۱ موارد زیر را انجام می‌دهد:
- ```
while (!flag)
memory_barrier();
print x
```
- رشته ۲ موارد زیر را انجام می‌دهد:
  - ```
x = 100;
memory_barrier();
flag = true
```
  - برای رشته ۱ تضمین می‌شود که مقدار **flag** قبل از مقدار **x** بارگذاری شود.
  - برای رشته ۲ اطمینان حاصل می‌کنیم که اختصاص به **x** قبل از اختصاص **flag** رخ دهد.



# همگام‌سازی سخت‌افزار

- بسیاری از سیستم‌ها برای پیاده‌سازی کد بخش بحرانی، پشتیبانی سخت‌افزاری ارائه می‌دهند.
- پردازنده‌های تک (Uniprocessors) می‌توانند وقفه‌ها را غیرفعال کنند.
  - کد در حال اجرا بدون تصاحب از پیش اجرا می‌شود.
  - به طور کلی در سیستم‌های چند پردازنده بسیار ناکارآمد است.
- ▶ سیستم‌عامل‌هایی که از این روش استفاده می‌کنند، قابلیت‌شان به طور گسترده قابل مقیاس نیست.
- به چند شکل از پشتیبانی سخت‌افزاری نگاه می‌کنیم:
  1. دستورالعمل‌های سخت‌افزاری
  2. متغیرهای اتمی (Atomic Variables)



# دستورات سخت افزاری

■ دستورالعمل‌های سخت افزاری ویژه‌ای که به ما اجازه می‌دهند محتویات یک کلمه را آزمایش و اصلاح کنیم، یا محتویات دو کلمه را به صورت اتمی (بدون وقفه) جابجا کنیم.

■ دستورالعمل Test-and-Set  
■ دستورالعمل مقایسه و تعویض-Swap



# دستور آزمایش و تنظیم

## ■ Definition

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

## ■ خصوصیات:

- به صورت اتمی اجرا می‌شود. یعنی اگر دو دستور testandset در هسته‌های مختلف اجرا شوند، اجرای آن‌ها متوالی خواهد نه همزمان.
- مقدار اصلی پارامتر ارسالی را برمی‌گرداند.
- مقدار جدید پارامتر ارسالی را روی **True** تنظیم می‌کند.



# Solution Using test\_and\_set()

- متغیر اشتراکی Boolean lock به نام `lock` که مقدار اولیه آن `false` است.
- راه حل:

```
▪ do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
    /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```

- آیا این مشکل بخش بحرانی را حل می‌کند؟



# دستور مقایسه و تعویض

## ■ Definition

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

### ■ خصوصیات:

- به صورت اتمی اجرا می‌شود.
- مقدار اصلی پارامتر ارسالی را برمی‌گرداند.
- مقدار متغیر را روی مقدار پارامتر جدید new\_value تنظیم می‌کند، اما تنها در صورتی که \*value == expected درست باشد. یعنی تبادل تنها تحت این شرط انجام می‌شود.



# راه کار با الگوریتم تعویض و جابجایی

قفل عدد صحیح اشتراکی lock که مقدار اولیه آن ۰ است.

راه حل:

- Does it solve the critical-section problem?

```
while I(true){  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
}
```

```
int compare_and_swap(int *value, int  
expected, int new_value)  
{  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

آیا این کد مسئله بخش بحرانی را حل می‌کند؟

بله، از نظر انحصار متقابل و پیشرفت. 

خیر، از نظر انتظار محدود و عدالت. 



# انتظار محدود با الگوریتم تعویض و جابجایی

```
boolean waiting[n]=false; int lock=0;
```

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}
```

- ◆ حلقه‌ای بینهایت: هر نخ (یا فرآیند) بارها تلاش می‌کند به بخش بحرانی وارد شود.
- ◆ نخ شماره اعلامت می‌زند که می‌خواهد وارد بخش بحرانی شود. متغیر محلی key را برابر ۱ قرار می‌دهد. این متغیر برای تلاش در گرفتن قفل استفاده می‌شود.
- ◆ این حلقه تا زمانی که نخ منتظر است و قفل آزاد نشده ادامه دارد.
- ◆ تابع compare\_and\_swap تلاش می‌کند قفل را از ۰ به ۱ تغییر دهد. اگر موفق نشد (یعنی قفل در اختیار نخ دیگری است)، مقدار key برابر ۱ باقی می‌ماند و حلقه ادامه می‌یابد.
- ◆ وقتی نخ موفق به گرفتن قفل شد، وضعیت "در حال انتظار" بودن خود را به false تغییر می‌دهد.
- ◆ این بخش بررسی می‌کند که آیا نخ دیگری منتظر ورود به بخش بحرانی هست یا نه ◆ از نخ بعدی (i+1) شروع می‌کند و حلقه می‌چرخد تا نخ منتظری پیدا کند یا به خود ابرسد.
- ◆ اگر هیچ نخ منتظری پیدا نشد (یعنی فقط نخ فعلی در حال فعالیت بوده)، قفل را آزاد می‌کند.
- ◆ در غیر این صورت، نخ بعدی را از حالت انتظار خارج می‌کند تا بتواند وارد بخش بحرانی شود.



# انتظار محدود با الگوریتم تعویض و جابجایی

برای اثبات اینکه الزام انحصار متقابل برآورده شده است، توجه می کنیم که فرآیند  $Pi$  فقط در صورتی می تواند وارد بخش بحرانی خود شود که  $key == 0$  یا  $waiting[i] == false$  باشد.

- مقدار  $key$  تنها زمانی می تواند به ۰ برسد که دستور «مقایسه و تعویض» (CAS) اجرا شود. اولین فرآیندی که دستور «مقایسه و تعویض» را اجرا می کند،  $key == 0$  را پیدا خواهد کرد. همه فرآیندهای دیگر باید منتظر بمانند. متغیر  $[i] waiting$  تنها زمانی می تواند به  $false$  تبدیل شود که فرآیند دیگری بخش بحرانی خود را ترک کند.

- برای اثبات اینکه الزام پیشرفت برآورده شده است، فرآیندی که بخش بحرانی را ترک می کند،  $lock$  را روی  $[j]$  تنظیم می کند یا  $waiting[j] == false$  را روی قرار می دهد. هر دو مورد به فرآیندی که در انتظار ورود به بخش بحرانی خود است، اجازه ادامه می دهند.

- برای اثبات اینکه الزام انتظار محدود برآورده شده است، توجه می کنیم که وقتی یک فرآیند بخش بحرانی خود را ترک می کند، آرایه  $waiting$  را در ترتیب چرخشی  $i, ..., n - 1, 0, ..., i + 2, i + 1$  (اسکن می کند. اولین فرآیندی را در این ترتیب که در بخش ورودی قرار دارد ( $waiting[j] == true$ ) به عنوان فرآیند بعدی برای ورود به بخش بحرانی تعیین می کند. بنابراین، هر فرآیندی که در انتظار ورود به بخش بحرانی خود باشد، ظرف  $1 - n$  دور این کار را انجام خواهد داد.



# متغیرهای اتمی

■ افزایش یا کاهش مقدار یک عدد صحیح ممکن است شرایط مسابقه (race condition) ایجاد کند

■ متغیرهای اتمی (atomic variables) را می توان برای تضمین خروج متقابل (mutual exclusion) در موقعیت هایی که ممکن است رقابت داده ای (data race) روی یک متغیر واحد در حین به روز رسانی آن رخ دهد، مانند زمانی که یک شمارنده افزایش می یابد، استفاده کرد.

■ یکی از این ابزارها، متغیر اتمی است که به روزرسانی های اتمی (بدون وقفه) را روی انواع داده های اساسی مانند اعداد صحیح و بولی فراهم می کند.

■ برای مثال:

■ فرض کنید **sequence** یک متغیر اتمی باشد.

■ فرض کنید (**increment()**) عملیاتی روی متغیر اتمی **sequence** باشد.

■ دستور:

■ تضمین می کند که **sequence** بدون وقفه افزایش می یابد:



# متغیرهای اتمی

▪ پیاده سازی تابع increment()

```
void increment	atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v, temp, temp+1)));
}
```

▪ این تابع با نام increment، یک عملیات افزایش اتمی (atomic increment) را روی یک متغیر عدد صحیح انجام می‌دهد که به صورت atomic\_int تعریف شده است. این روش از الگوریتم Compare-and-Swap (CAS) برای اطمینان از عدم تداخل همزمانی (race condition) استفاده می‌کند.



# قفل های انجصار متقابل Mutex

راه حل های قبلی پیچیده و به طور کلی برای برنامه نویسان کاربردی در دسترس نیستند.

طراحان سیستم عامل ابزارهای نرم افزاری برای حل مشکل بخش بحرانی می سازند.

ساده ترین قفل انجصار متقابل (Mutex Lock) است.

- متغیر بولی که نشان می دهد قفل در دسترس است یا خیر.

- برای محافظت از یک بخش بحرانی با موارد زیر عمل کنید:

- ابتدا یک قفل را با `acquire()` به دست آورید.

- سپس قفل را با `release()` رها کنید.

- صدورهای `acquire()` و `release()` باید اتمی باشند.

- معمولًا از طریق دستور العمل های اتمی سخت افزاری مانند مقایسه و تعویض پیاده سازی می شوند.

- اما این راه حل نیازمند انتظار فعال (`busy waiting`) است.

- بنابراین این قفل یک چرخش قفل (`spinlock`) نامیده می شود.



# راه کار برای مسئله بخش بحرانی با قفل Mutex

```
while (true) {  
    acquire lock  
  
    critical section  
  
    release lock  
  
    remainder section  
}
```



# سِمافور

- سِمافور ابزاری برای همگام‌سازی است که روش‌های پیچیده‌تری (نسبت به قفل‌های انحصار متقابل) را برای فرآیندها برای همگام‌سازی فعالیت‌هایشان ارائه می‌دهد.
- سِمافور :  $S$  یک متغیر عدد صحیح است.
- فقط از طریق دو عملیات غیرقابل تقسیم (اتمی) قابل دسترسی است.
- دستورات  $V$ signal() یا  $P$  یا  $wait()$  تعريف عملیات انتظار

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

تعريف عملیات سیگنال



# سِمافور (ادامه)

- • سِمافور شمارش : مقدار عدد صحیح می‌تواند روی یک دامنه بدون محدودیت قرار گیرد.
- • سِمافور دودویی : مقدار عدد صحیح می‌تواند تنها بین ۰ و ۱ قرار گیرد.
- • مشابه قفل انحصار متقابل (Mutex Lock)
- • می‌توان یک سِمافور شمارش S را به عنوان یک سِمافور دودویی پیاده‌سازی کرد.
- با استفاده از سِمافورها می‌توانیم مشکلات مختلف همگام‌سازی را حل کنیم.



# مثال کاربرد سمافور

راهکار برای مسئله بخش بحرانی

- یک سمافور **mutex** با مقدار اولیه ۱

```
wait(mutex);
```

CS

```
signal(mutex);
```

■ فرآیندهای P1 و P2 را در نظر بگیرید که دارای دو دستورالعمل S1 و S2 هستند و الزام این است که S1 قبل از S2 اتفاق بیفتد.

■ یک سمافور ”synch“ ایجاد کنید که مقدار اولیه آن ۰ است.

P1:

```
S1;
```

```
signal(synch);
```

P2:

```
wait(synch);
```

```
S2;
```



# پیاده سازی سمافور

- باید تضمین شود که هیچ دو فرآیندی نمی‌توانند `() wait` و `signal()` را روی یک سمافور مشابه در یک زمان اجرا کنند.
- بنابراین، پیاده‌سازی به مشکل بخش بحرانی تبدیل می‌شود که در آن کدهای `wait` و `signal` در بخش بحرانی قرار می‌گیرند.
- حالا ممکن است انتظار فعال (`busy waiting`) در پیاده‌سازی بخش بحرانی وجود داشته باشد.
  - کد پیاده‌سازی کوتاه است.
  - انتظار فعال کمی وجود دارد اگر بخش بحرانی به ندرت اشغال شود.
- توجه داشته باشید که برنامه‌ها ممکن است زمان زیادی را در بخش‌های بحرانی صرف کنند و بنابراین این راه حل خوبی نیست.



## پیاده سازی سمافور بدون انتظار فعال

- همراه با هر سمافور یک صف انتظار مرتبط وجود دارد.
- هر ورودی در یک صف انتظار دارای دو داده است:
  - مقدار (از نوع عدد صحیح)
  - اشاره‌گر به رکورد بعدی در لیست
- دو عملیات وجود دارد:
  - مسدود کردن : (**Block**) فرآیندی که این عملیات را فراخوانی می‌کند در صف انتظار مناسب قرار می‌دهد.
  - بیدار کردن : (**Wakeup**) یکی از فرآیندهای موجود در صف انتظار را حذف کرده و آن را در صف آماده قرار می‌دهد.



# پیاده سازی سمافور بدون انتظار فعال(ادامه)

صف انتظار

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```



# پیاده سازی سمافور بدون انتظار فعال (ادامه)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        sleep();  
    }  
}
```

---

```
-----  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```



# Problems with Semaphores

- استفاده نادرست از عملیات‌های سمافور
  - **signal (mutex) . . . wait (mutex)**
    - ▶ چندین فرایند در ناحیه بحرانی خود هستند.
  - **wait (mutex) ... wait (mutex)**
    - فرآیند تا ابد به خواب می‌رود.
  - **Omitting of wait (mutex) and/or signal (mutex)**
- این موارد - و موارد دیگر - نمونه‌هایی از اتفاقاتی هستند که می‌توانند در صورت استفاده نادرست از سمافورها و سایر ابزارهای همگام‌سازی رخ دهند.



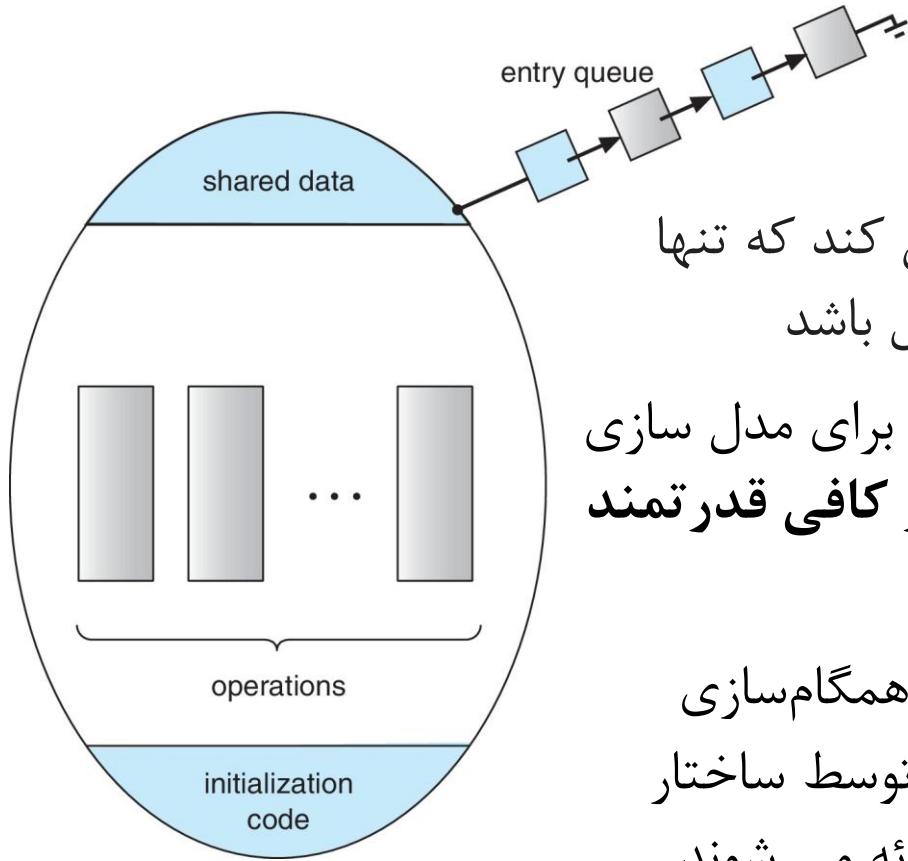
# Monitors

- یک نوع داده انتزاعی (Abstract Data Type) ADT که یک مکانیزم راحت و مؤثر برای همگام‌سازی فرآیند را فراهم می‌کند.
- نوع داده انتزاعی، متغیرهای داخلی فقط توسط کد درون رویه (مانیتور) قابل دسترسی هستند.
- فقط یک فرآیند می‌تواند در یک زمان در داخل مانیتور فعال باشد.
- نحوه نگارش شبه کد یک مانیتور:

```
monitor monitor-name
{
    // shared variable declarations
    function P1 (...) { .... }
    function P2 (...) { .... }
    function Pn (...) {.....}
    initialization code (...) { ... }
}
```



# Schematic view of a Monitor



- ❖ عبارت "مانیتور" (monitor) تضمین می کند که تنها یک فرآیند در یک زمان درون مانیتور فعال باشد
- ❖ ساختار مانیتور (monitor construct) برای مدل سازی برخی از طرح های همگام سازی به قدر کافی قدرتمند نیست .
- ❖ برای این منظور، ما باید مکانیزم های همگام سازی اضافی تعریف کنیم. این مکانیزم ها توسط ساختار شرط (condition construct) ارائه می شوند.
- ❖ برنامه نویسی که نیاز به نوشتن یک طرح همگام سازی سفارشی دارد، می تواند یک یا چند متغیر از نوع شرط (condition) تعریف کند.



## پیاده سازی مانیتور با استفاده از سیمافور

### متغیرها

```
semaphore mutex  
mutex = 1
```

■ هر تابع P با موارد زیر جایگزین می‌شود:

```
■ Function P() {  
    wait(mutex);  
    ...  
    body of P;  
    ...  
    signal(mutex);  
}
```

■ انحصار متقابل درون یک مانیتور تضمین می‌شود.



# Condition Variables

- **condition x, y;**

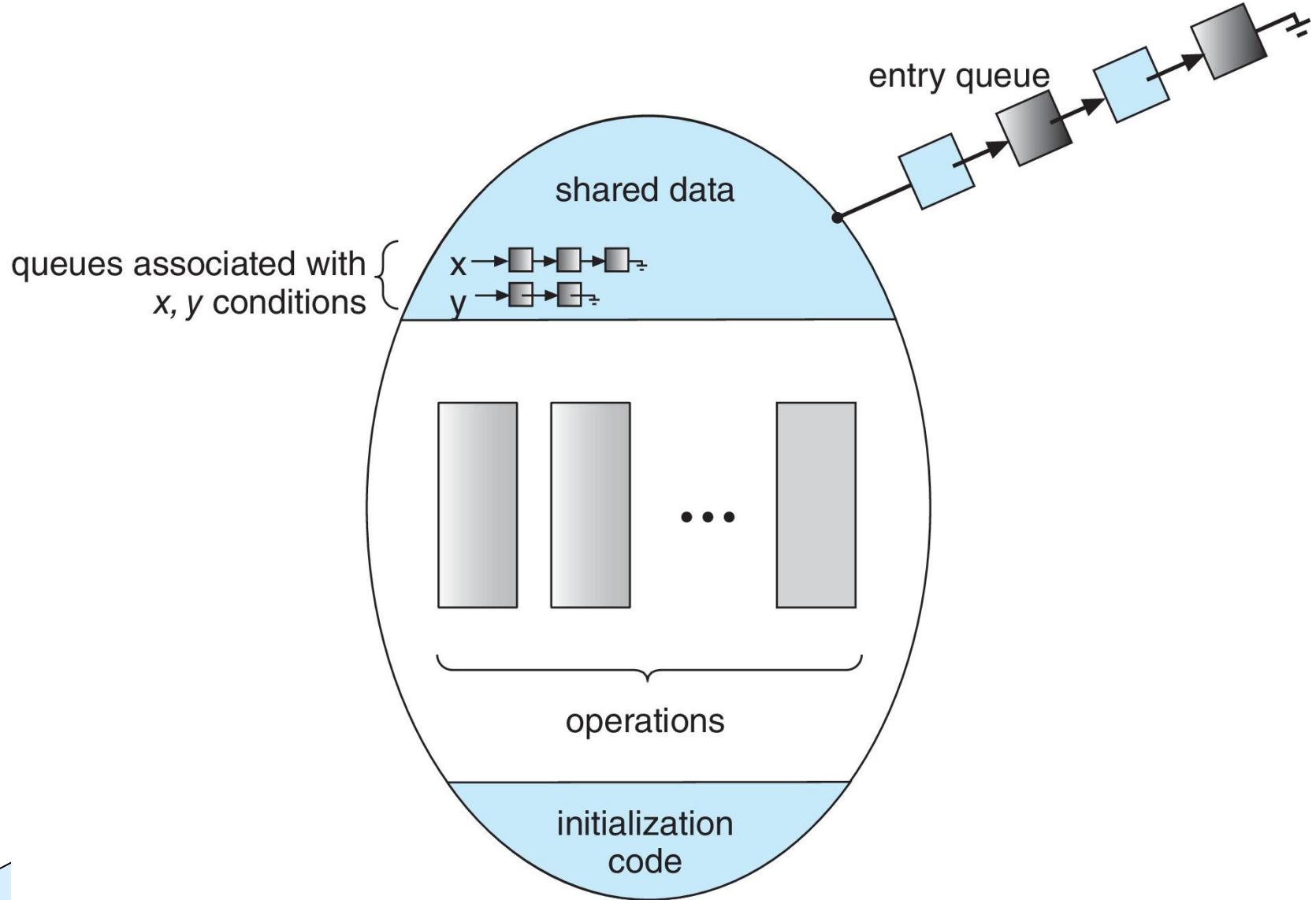
- دو عملیات روی یک متغیر شرط مجاز است:

- فرآیندی که این عملیات را فراخوانی می‌کند تا زمانی که () **x.wait()**: داده شود، معلق می‌شود.

- **x.signal():** یکی از فرآیندهایی را که () **x.wait()** را فراخوانی کرده‌اند (در صورت وجود) از سر می‌گیرد.



# Monitor with Condition Variables





# Usage of Condition Variable Example

- فرآیندهای P1 و P2 را در نظر بگیرید که نیاز به اجرای دو دستورالعمل S1 و S2 دارند و الزام این است که S1 قبل از S2 اتفاق بیفت.
- یک مانیتور با دو تابع F1 و F2 ایجاد کنید که به ترتیب F1 توسط P1 و F2 توسط P2 فراخوانی می‌شوند.
- یک متغیر شرطی "x" که مقدار اولیه آن ۰ است.
- یک متغیر بولی "done"

F1 called by P1:

```
S1;  
done = true;  
x.signal();
```

F2 called by P2:

```
if done = false  
x.wait()  
S2;
```

مجاز به اجرای S2 نیست، مگر اینکه P1 قبل از S1 را اجرا کرده باشد



# پیاده سازی مانیتور با سمافور

## ■ پیاده سازی مانیتور با سمافور

- برای هر مانیتور، یک سیمافور دودویی به نام **mutex** (که در ابتدا مقدار آن ۱ است) برای تضمین دسترسی متقابل در نظر گرفته می‌شود.
- یک فرایند باید قبل از ورود به مانیتور دستور **wait(mutex)** را اجرا کند و بعد از خروج از مانیتور دستور **signal(mutex)** را اجرا کند.
- ما در پیاده‌سازی خود از طرح **signal-and-wait** استفاده خواهیم کرد. از آنجایی که فرایند سیگنال دهنده باید منتظر بماند تا فرایند از سر گرفته شده، مانیتور را ترک کند یا منتظر بماند، یک سیمافور دودویی اضافی به نام **next** معرفی می‌شود که در ابتدا مقدار آن ۰ است.
- فرایندهای سیگنال دهنده می‌توانند از **next** برای تعليق خود استفاده کنند. همچنان یک متغیر صحیح به نام **next\_count** برای شمارش تعداد فرایندهای معلق بر روی **next** در نظر گرفته می‌شود.



# Single Resource allocation

- تخصیص یک منبع واحد بین فرآیندهای رقیب با استفاده از اعداد اولویت که حداکثر زمانی را که یک فرآیند برای استفاده از منبع برنامه ریزی کرده است، مشخص می کند.
- در اینجا R نمونه ای از نوع ResourceAllocator است.
- دسترسی به تخصیص دهنده منابع (ResourceAllocator) از طریق موارد زیر انجام می شود:
  - که در آن t حداکثر زمانی است که یک فرآیند برای استفاده از منبع برنامه ریزی کرده است.

**R.acquire(t);**

...

**access the resource;**

...

**R.release;**



# A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        while (busy)
            x.wait(time);
        busy = true;
    }
    void release() {
        busy = false;
        x.signal();
    }
    initialization code() {
        busy = false;
    }
}
```

- **busy** یک متغیر بولی که مشخص می‌کند آیا منبع در حال استفاده است یا خیر.
- **x** متغیر شرطی که نخ‌هایی که منتظر آزاد شدن منبع هستند روی آن منتظر می‌مانند.
- ◆ این متده برای درخواست دسترسی به منبع استفاده می‌شود.
- اگر منبع مشغول (**busy = true**) باشد، نخ با استفاده از **x.wait(time)** برای حداکثر مدت زمان مشخص شده منتظر می‌ماند.



# Single Resource Monitor (Cont.)

کاربرد:

`acquire`

...

`release`

استفاده نادرست از عملیات‌های مانیتور

- `release()` ... `acquire()`
- `acquire()` ... `acquire()`)
- Omitting of `acquire()` and/or `release()`



## زنده بودن

- ممکن است فرآیندها در هنگام تلاش برای به دست آوردن یک ابزار همگام‌سازی مانند قفل انحصار متقابل (Mutex Lock) یا سیمافور، به طور نامحدود منتظر بمانند.
- انتظار نامحدود، معیارهای پیشرفت و انتظار محدود که در ابتدای این فصل مورد بحث قرار گرفت را نقض می‌کند.
- زنده بودن : (Liveness) به مجموعه‌ای از ویژگی‌هایی اشاره دارد که یک سیستم برای اطمینان از پیشرفت فرآیندها باید از آن‌ها پیروی کند.
- انتظار نامحدود نمونه‌ای از شکست زنده بودن (Liveness Failure) است.



## زنده‌بودن

بن‌بست: (Deadlock) دو یا چند فرآیند به طور نامحدود منتظر رویدادی هستند که فقط می‌توانند توسط یکی از فرآیندهای در حال انتظار ایجاد شود.

فرض کنید  $S$  و  $Q$  دو سمافور هستند که مقدار اولیه آن‌ها ۱ است.

$P_0$

**wait(S) ;**

**wait(Q) ;**

...

**signal(S) ;**

**signal(Q) ;**

$P_1$

**wait(Q) ;**

**wait(S) ;**

...

**signal(Q) ;**

**signal(S) ;**

در نظر بگیرید که اگر  $P_0$  دستور  $\text{wait}(S)$  و  $P_1$  دستور  $\text{wait}(Q)$  را اجرا کند. هنگامی که  $P_0$  دستور  $\text{wait}(Q)$  را اجرا می‌کند، باید تا زمانی که  $P_1$  دستور  $\text{signal}(Q)$  را اجرا کند، منتظر بماند.

با این حال،  $P_1$  تا زمانی که  $P_0$  دستور  $\text{signal}(S)$  را اجرا نکند، در حال انتظار است. از آنجایی که این عملیات  $\text{signal}()$  هرگز اجرا نمی‌شوند،  $P_0$  و  $P_1$  دچار بن‌بست شده‌اند



## زنده‌بودن

- سایر اشکال بن‌بست:
- گرسنگی (**Starvation**): بلاک شدن نامحدود
- فرآیندی ممکن است هرگز از صف سِمافوری که در آن معلق (**sleep**) است، حذف نشود.



# مثال‌های از کاربرد مفاهیم همگام‌سازی



# Classical Problems of Synchronization

- مسائل کلاسیک برای تست رویکردهای همگام‌سازی جدید
- مسئله بافر محدود (Bounded-Buffer Problem):
  - مسئله نویسندها و خوانندها
  - مسئله فیلسوفان گرسنه



## مسئله بافر محدود

- این مسئله شامل  $n$  بافر است که هر کدام می‌تواند فقط یک آیتم را در خود جای دهد.
- یک سیمافور به نام **mutex** وجود دارد که مقدار اولیه آن برابر با **1** است.
- یک سیمافور دیگر به نام **full** با مقدار اولیه **0** وجود دارد.
- و در نهایت یک سیمافور به نام **empty** با مقدار اولیه **n** داریم.
- همچنین لازم است تا ساختار فرایند تولیدکننده (producer process) و ساختار فرایند مصرف‌کننده (consumer process) تعریف شود.
- توجه: در این بخش، جزئیات مربوط به ساختار فرایندها ذکر نشده است.



# مسئله بافر محدود (ادامه)

- ساختار فرآیند تولیدکننده

```
while (true) {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
}
```



# Bounded Buffer Problem (Cont.)

- ساختار فرآیند مصرف‌کننده

```
while (true) {  
    wait(full);  
    wait(mutex);  
  
    ...  
    /* remove an item from buffer to next_consumed */  
  
    ...  
    signal(mutex);  
    signal(empty);  
  
    ...  
    /* consume the item in next consumed */  
  
    ...  
}
```



# مسئله خوانندگان و نویسندها

## ■ مسئله خوانندگان و نویسندها (Readers-Writers Problem)

■ در این مسئله، یک مجموعه مشترک بین چندین فرایند همزمان به اشتراک گذاشته شده است.

- **خوانندگان Reader:** این فرایندها فقط مجاز به خواندن داده‌ها هستند و هیچ گونه تغییری در آن‌ها ایجاد نمی‌کنند.

- **نویسندها Writers:** این فرایندها می‌توانند هم داده‌ها را بخوانند و هم بنویسند.

### ■ مسئله:

- اجازه دهیم چندین خواننده بتوانند همزمان داده‌ها را بخوانند.

- فقط یک نویسنده می‌تواند به طور همزمان به داده‌های مشترک دسترسی پیدا کند.

- چندین رویکرد مختلف برای نحوه مدیریت خوانندگان و نویسندها وجود دارد که همه آن‌ها شامل نوعی اولویت‌بندی هستند.



# (ادامه) مسئله خوانندگان و نویسندها

داده‌های مشترک:

- مجموعه داده
- سیمافور `rw_mutex` که در ابتدا مقدار آن ۱ است.
- سیمافور `mutex` که در ابتدا مقدار آن ۱ است.
- یک عدد صحیح `read_count` به نام `integer` که در ابتدا مقدار آن ۰ است.



# (ادامه) مسئله خوانندگان و نویسندها

```
while (true) {  
    wait(rw_mutex);  
    ...  
    /* writing is performed */  
    ...  
    signal(rw_mutex);  
}
```

## ■ ساختار فرآیند نویسنده

نویسنده باید دسترسی انحصاری داشته باشد: وقتی در حال نوشتن است، نه خواننده و نه نویسنده دیگری نمی‌تواند به منبع دسترسی داشته باشد.

`rw_mutex` یک قفل دوچاره است که هم خواننده‌ها و هم نویسنده‌گان برای ورود به ناحیه بحرانی از آن استفاده می‌کنند.

نویسنده ابتدا با `wait(rw_mutex)` وارد ناحیه بحرانی می‌شود.

نوشتن انجام می‌شود.

با `signal(rw_mutex)` قفل آزاد می‌شود تا سایر خواننده‌گان یا نویسنده‌گان وارد شوند.



## (ادامه) مسئله خوانندگان و نویسندها

ساختار فرآیند خواننده: چند خواننده می‌توانند به‌طور همزمان داده را بخوانند.

```
while (true){  
    wait(mutex);  
    read_count++;  
    if (read_count == 1) /* first reader */  
        wait(rw_mutex);  
    signal(mutex);  
    /* reading is performed */  
    wait(mutex);  
    read_count--;  
  
    if (read_count == 0) /* last reader */  
        signal(rw_mutex);  
    signal(mutex);  
}
```

- قفل محافظ برای به‌روزرسانی ایمن متغیر `read_count` تعداد خوانندگان فعال.
- شمارنده‌ای که تعداد خوانندگان فعال را نگه می‌دارد.
- `rw_mutex` همان قفلی است که نویسنده نیز از آن استفاده می‌کند.



# Readers-Writers Problem Variations

- مشکل قفل شدگی خواننده اول (First Reader-Writer Problem):
  - ▶ راه حلی که در اسلاید قبلی ارائه شد، می‌تواند منجر به وضعیتی شود که یک فرایند نویسنده هرگز نتواند عمل نوشتن را انجام دهد. این مشکل به عنوان "قفل شدگی خواننده اول" شناخته می‌شود.
- مشکل قفل شدگی خواننده دوم (Second Reader-Writer Problem):
  - ▶ "قفل شدگی خواننده دوم" نسخه‌ای از مشکل "قفل شدگی خواننده اول" است که در آن بیان می‌شود:
    - ▶ به محض اینکه یک نویسنده برای نوشتن آماده شود، دیگر به هیچ خواننده‌ای که تازه وارد شده "اجازه خواندن داده نمی‌شود.
  - ▶ هر دو مشکل قفل شدگی خواننده اول و دوم می‌توانند منجر به گرسنگی (یعنی انتظار بی‌محدود) برای برخی فرایندها شوند. همین موضوع باعث ایجاد رویکردهای متنوع‌تری برای حل این مسئله شده است.



# Readers-Writers Problem Variations

- مشکل قفل شدگی خواننده اول زمانی رخ می دهد که خوانندگان همیشه اولویت داشته باشند و نویسنده ها هیچ گاه نتوانند وارد ناحیه بحرانی شده و بنویسنده (گرسنگی نویسنده) .
- مشکل قفل شدگی خواننده دوم نسخه ای از این مشکل است که وقتی نویسنده ای آماده نوشتن شود، هیچ خواننده جدیدی اجازه ورود نمی یابد تا نویسنده بتواند سریع تر وارد شود.
- هر دو مشکل باعث انتظار بی پایان (گرسنگی) برای برخی فرآیندها می شوند و نیاز به راهکارهای عادلانه تر دارند.

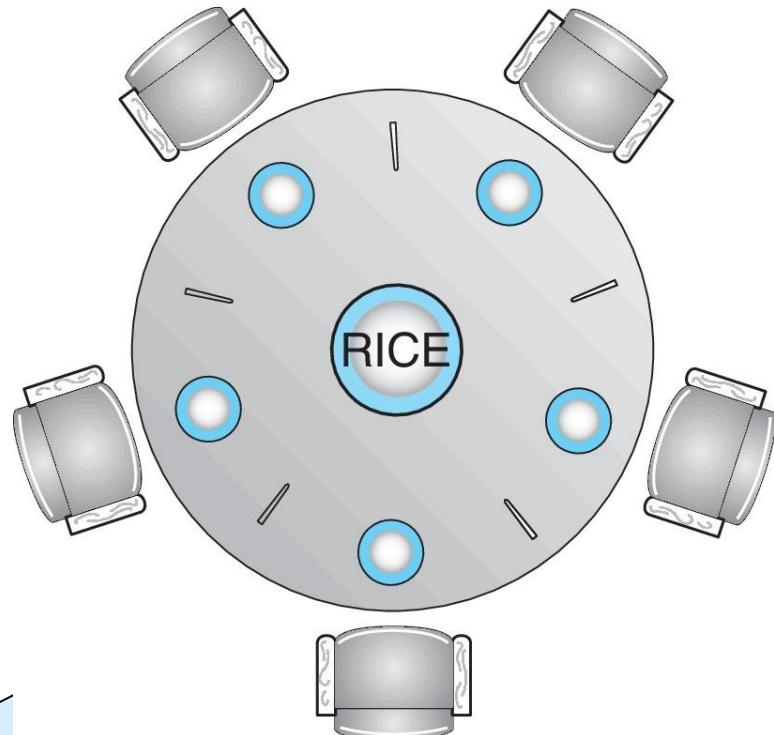


# Dining-Philosophers Problem

- فرض می‌کنیم ۵ فیلسوف وجود دارد
- داده‌های مشترک:

- کاسه برنج (مجموعه داده)

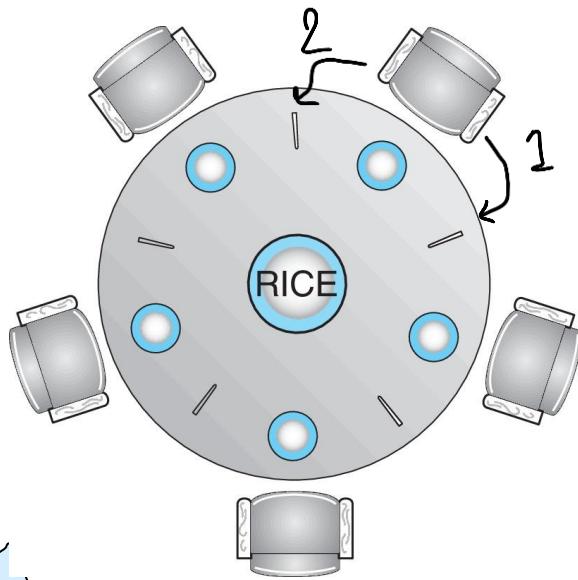
یک آرایه از ۵ سیمافور به نام چنگال که هر کدام در ابتدا مقدار ۱ دارند (هر سیمافور نشان‌دهنده در دسترس بودن یک چپستیک است).





# Dining-Philosophers Problem

- هر فیلسوف  $i$  به ترتیب زیر عمل می‌کند:
  - ابتدا تلاش می‌کند تا چنگال سمت چپ و سپس چنگال سمت راست را بردارد.
  - فیلسوف غذا می‌خورد.
  - سپس چپستیک‌ها را روی میز قرار می‌دهد تا فیلسفه‌دان دیگر بتوانند از آن‌ها استفاده کنند.





# Dining-Philosophers Problem Algorithm

- با استفاده از راه کار سمافور
- : اساختار یک فیلسوف

```
while (true){  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
    /* برای مدتی غذا بخور */  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] )  
    /* برای مدتی فکر کن */  
}
```

■ این الگوریتم چه مشکلی دارد؟



# Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{   enum {THINKING; HUNGRY, EATING} state [5];
    condition self [5];
    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }
    void putdown (int i) {state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5); }

    void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) && (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
    }
    initialization_code() {
        for (int i = 0; i < 5; i++){
            state[i] = THINKING;}
```



## Solution to Dining Philosophers (Cont.)

■ هر فیلسوف ا عمليات putdown و pickup را به ترتيب مقابل فرآخوانی می کند.

DiningPhilosophers.pickup(i);

/\*\* EAT \*\*/

DiningPhilosophers.putdown(i);

■ توجه : در اين سناريو، بن بست وجود ندارد اما احتمال گرسنگي (Starvation) برای برخى فيلسوفان وجود دارد (يعنى ممکن است يك فيلسوف برای مدت زمان نامحدودی منتظر بماند تا بتواند دو چنگال مورد نياز خود را بردارد).

# پایان فصل

