دانشگاه صنعتی شاهرود
دانشکده مهندسی کامپیوتر و فناوری اطلاعات

سیستم‌های عامل

مسأله ناحیه بحرانی

استاد: علیرضا تجری

وضعیت رقابتی

- فرض کنید متغیر **a** بین دو فرآیند (یا دو نخ) مشترک است.
- بخشی از کد این دو فرآیند به صورت زیر است. هر دو فرآیند می‌خواهند این خط را اجرا کنند.
- مقدار **a** قبل از اجرای این دو خط، 10 است.

```
P1:

a = a + 5
```

```
P2:

a = a - 5
```

- بعد از اجرای این دو خط، مقدار **a** چند می‌شود؟

15 10 5 -

وضعیت رقابتی

- بعد از اجرای این دو خط، مقدار **a** چند می‌شود؟
  - 5 10 15
  - هر سه مقدار امکان‌پذیر است!
- چند نکته
  - متغیرهای تعریف شده در زبان‌های برنامه نویسی، در حافظه قرار می‌گیرند.
  - عملیات در پردازنده و بر روی رجیسترها انجام می‌شود.
  - کامپایلر زبان سطح بالا را به زبان ماشین تبدیل می‌کند.

وضعیت رقابتی

- کد این فرآیندها پس از کامپایل
  - اینجا از دستورات شبه اسمبلی برای نمایش استفاده شده است.

```
P1:

a = a + 5
```

```
reg1 = a
reg1 = reg1 + 5
a = reg1
```

```
P2:

a = a - 5
```

```
reg2 = a
reg2 = reg2 - 5
a = reg2
```

وضعیت رقابتی

- آیا امکان دارد که ترتیب اجرای دستورات این دو فرآیند به صورت زیر باشد؟

```
P1: reg1 = a
P1: reg1 = reg1 + 5
P2: reg2 = a
P2: reg2 = reg2 - 5
P1: a = reg1
P2: a = reg2
```

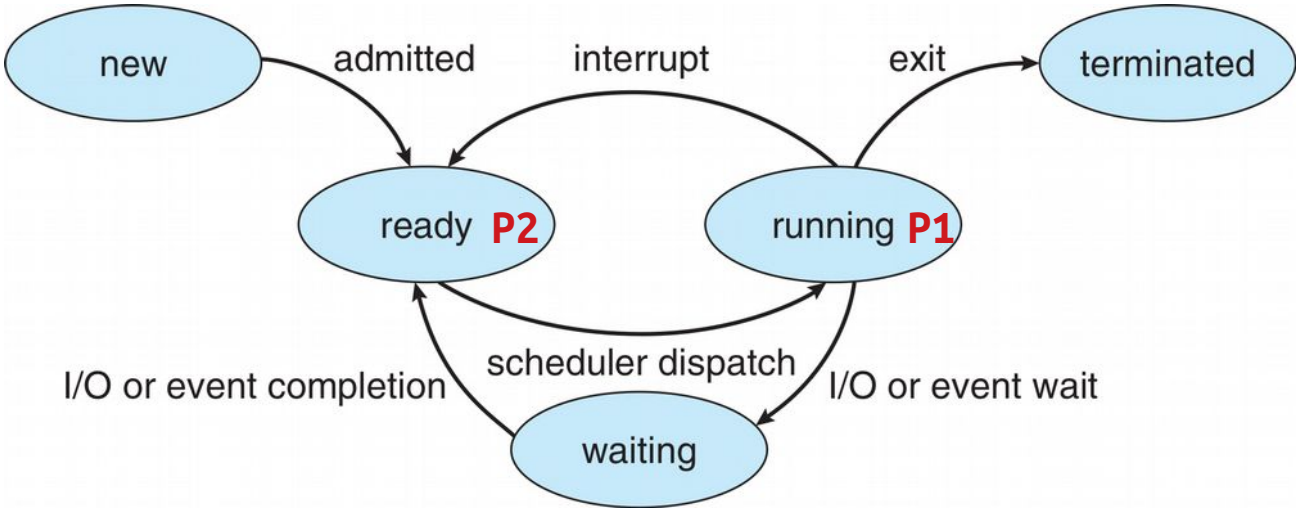
```
        P1
reg1 = a
reg1 = reg1 + 5
a = reg1
```

```
        P2
reg2 = a
reg2 = reg2 - 5
a = reg2
```

وضعیت رقابتی

- آیا امکان دارد که ترتیب اجرای دستورات این دو فرآیند به صورت زیر باشد؟

```
P1: reg1 = a

P1: reg1 = reg1 + 5

P2: reg2 = a

P2: reg2 = reg2 - 5

P1: a = reg1

P2: a = reg2
```



وضعیت رقابتی

- آیا امکان دارد که ترتیب اجرای دستورات این دو فرآیند به صورت زیر باشد؟

```
P1: reg1 = a
P1: reg1 = reg1 + 5
P2: reg2 = a
P2: reg2 = reg2 - 5
P1: a = reg1
P2: a = reg2
```

تغییر متن
Context Switch

وضعیت رقابتی

- مقدار متغیر **a** پس از اجرای این دستورات چند است؟

```
P1: reg1 = a

P1: reg1 = reg1 + 5

P2: reg2 = a

P2: reg2 = reg2 - 5

P1: a = reg1

P2: a = reg2
```

وضعیت رقابتی

- مقدار متغیر **a** پس از اجرای این دستورات چند است؟

```
P1: reg1 = a                    reg1 = 10
P1: reg1 = reg1 + 5             reg1 = 15
P2: reg2 = a                    reg2 = 10
P2: reg2 = reg2 - 5             reg2 = 5
P1: a = reg1                    a = 15
P2: a = reg2                    a = 5 !
```

وضعیت رقابتی

- آیا امکان دارد مقدار **a** برابر با 15 شود؟

```
P1: reg1 = a

P1: reg1 = reg1 + 5

P2: reg2 = a

P2: reg2 = reg2 - 5

P2: a = reg2

P1: a = reg1
```

وضعیت رقابتی

- آیا امکان دارد مقدار **a** برابر با 10 شود؟
  - مقدار مد نظر برنامه نویس

P1

```
P1: reg1 = a
P1: reg1 = reg1 + 5
P1: a = reg1
```

P2

```
P2: reg2 = a
P2: reg2 = reg2 - 5
P2: a = reg2
```

P2

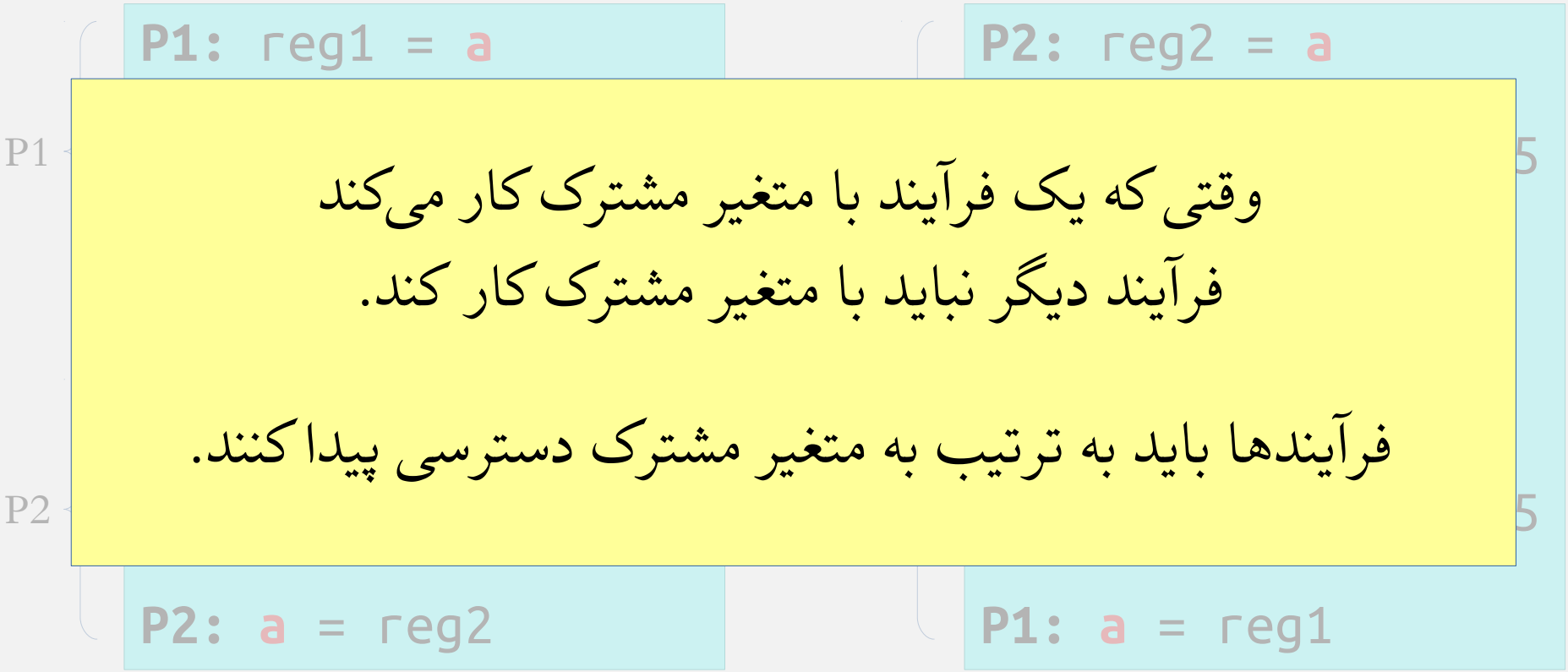
```
P2: reg2 = a
P2: reg2 = reg2 - 5
P2: a = reg2
```

P1

```
P1: reg1 = a
P1: reg1 = reg1 + 5
P1: a = reg1
```

وضعیت رقابتی

- آیا امکان دارد مقدار **a** برابر با 10 شود؟
  - مقدار مد نظر برنامه نویس



وقتی که یک فرآیند با متغیر مشترک کار می‌کند
فرآیند دیگر نباید با متغیر مشترک کار کند.

فرآیندها باید به ترتیب به متغیر مشترک دسترسی پیدا کنند.

وضعیت رقابتی Race Condition

- وضعیت رقابتی: وضعیتی که چند فرآیند (یا نخ) از یک منبع مشترک (در مثال قبل، داده مشترک) به صورت همزمان استفاده می‌کنند و نتیجه اجرا وابسته به ترتیب دسترسی به منبع مشترک است.
- وضعیت بسیار بدی است!
- راه حل جلوگیری از رخ دادن وضعیت رقابتی
  - در هر لحظه، فقط یک فرآیند با منبع مشترک کار کند.
- آیا ممکن است وضعیت رقابتی رخ دهد؟
  - بله، فرآیندهای همکار دارای منابع مشترک هستند.
  - بخش داده در نخ‌ها مشترک است.

مسأله ناحیه بحرانی

- فرض کنید سیستمی با **n** فرآیند داریم.
- $\{P_0, P_1, \cdots, P_{n-1}\}$
- هر فرآیند دارای قطعه کدی به نام ناحیه بحرانی است.
- فرآیندها در ناحیه بحرانی خود به منبع مشترک دسترسی پیدا می‌کنند.
  - در هر لحظه فقط یک فرآیند باید در ناحیه بحرانی خود باشد. چرا؟
- مسأله ناحیه بحرانی، طراحی قراردادی است که فرآیندها برای همکاری با هم آن را اجرا می‌کنند تا وضعیت رقابتی پیش نیاید.

مسأله ناحیه بحرانی

- به طور کلی، هر فرآیند را می‌توان به صورت زیر نشان داد.

بخش ورود

ناحیه بحرانی

بخش خروج

بخش باقیمانده

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```

مسأله ناحیه بحرانی

- مثال

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

```
P0
do{
   entry section
      critical section
   exit section
   remainder section
}while(true)
```

```
P1
while(true){
   entry section
      critical section
   exit section
   remainder section
}
```

ویژگی‌های راه‌حل مسأله ناحیه بحرانی

- هر راه‌حل مسأله بحرانی سه ویژگی زیر را داشته باشد:
  - انحصار متقابل Mutual Exclusion
  - پیشروی Progress
  - انتظار محدود Bounded Waiting

ویژگی‌های راه‌حل مسأله ناحیه بحرانی

- انحصار متقابل **Mutual Exclusion**
  - اگر فرآیند Pi در ناحیه بحرانی خود قرار دارد، هیچ فرآیند دیگری نباید به ناحیه بحرانی خود وارد شود.
    - در هر لحظه فقط یک فرآیند می‌تواند وارد ناحیه بحرانی خود شود.

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

ویژگی‌های راه‌حل مسأله ناحیه بحرانی

- **پیشروی Progress**
  - اگر هیچ فرآیندی در ناحیه بحرانی خود قرار ندارد، فقط فرآیندهایی که در بخش ورود و یا بخش خروج خود هستند، می‌توانند تعیین کنند که کدام فرآیند وارد ناحیه بحرانی شود.
  - انتخاب این فرآیند نباید به صورت نامعین طول بکشد.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

ویژگی‌های راه‌حل مسأله ناحیه بحرانی

- انتظار محدود **Bounded Waiting**

  - از زمانی که یک فرآیند درخواست ورود به ناحیه بحرانی می‌دهد تا زمانی که وارد ناحیه بحرانی می‌شود، تعداد محدودی از فرآیندها می‌توانند وارد ناحیه بحرانی شوند.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

راه‌حل پیترسون برای مسأله ناحیه بحرانی

- سؤال: هر یک از قطعه کدهای زیر چه کاری انجام می‌دهد؟

```
while(a>b){

    print a;

}
```

```
while(a>b);

print a;
```

```
while(true){

    print a;

}
```

```
while(true);

print a;
```

راه‌حل پیترسون برای مسأله ناحیه بحرانی

- الگوریتم پیترسون **Peterson’s Algorithm**
  - یک راه‌حل نرم‌افزاری است.
  - برای سیستمی با دو فرآیند طراحی شده است.
    - امکان توسعه به n فرآیند وجود دارد.
  - از دو متغیر مشترک بین دو فرآیند استفاده می‌کند.
  - امروزه در پردازنده‌های چند هسته‌ای قابل استفاده نیست.
  - مثال خوبی از یک راه‌حل است.

راه‌حل پیترسون برای مسأله ناحیه بحرانی

- ساختار فرایند $\mathbf{P_i}$ در الگوریتم پیترسون

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);

                critical section

        flag[i] = false;

                remainder section

} while (true);
```

بخش ورود

ناحیه بحرانی

بخش خروج

بخش باقیمانده

راه‌حل پیترسون برای مسأله ناحیه بحرانی

- متغیرهای مشترک
  - int turn
    - تعیین می‌کند که الان نوبت کدام فرآیند است که وارد ناحیه بحرانی شود.
    - اگر turn == 0 باشد، نوبت P0 وارد ناحیه بحرانی بشود.
    - اگر turn == 1 باشد، نوبت P1 وارد ناحیه بحرانی بشود.
  - boolean flags[2]
    - تعیین می‌کند که آیا یک فرآیند می‌خواهد وارد ناحیه بحرانی بشود یا خیر.
    - اگر flags[0] == true باشد، P0 می‌خواهد وارد ناحیه بحرانی بشود.
    - اگر flags[1] == true باشد، P1 می‌خواهد وارد ناحیه بحرانی بشود.
    - مقدار اولیه flags ها false است.

راه‌حل پیترسون برای مسأله ناحیه بحرانی

- ایده کلی
  - در ابتدا هر فرآیند اعلام می‌کند که می‌خواهد وارد ناحیه بحرانی شود.
    - مقدار flags مربوط به خود را true می‌کند.
  - اما نوبت را به فرآیند دیگر می‌دهد.
    - مقدار turn را برابر با شماره فرآیند دیگر می‌کند.
    - تعارف می‌کند!
  - چک می‌کند که اگر فرآیند دیگر وارد ناحیه بحرانی نشد، خودش وارد ناحیه بحرانی بشود.

راه‌حل پیترسون برای مسأله ناحیه بحرانی

```
do{
   flags[0] = true;
   turn = 1;
   while (flags[1] && turn == 1);

   ناحیه بحرانی

   flags[0] = false;

   بخش باقیمانده

}while(true)
```

- کد فرآیند **P0**

```
flags[0] = false;
flags[1] = false;
```

راه‌حل پیترسون برای مسأله ناحیه بحرانی

```
do{
   flags[1] = true;
   turn = 0;
   while (flags[0] && turn == 0);

   ناحیه بحرانی

   flags[1] = false;

   بخش باقیمانده

}while(true)
```

- کد فرآیند **P1**

```
flags[0] = false;
flags[1] = false;
```

راه‌حل پیترسون برای مسأله ناحیه بحرانی

P1

```
do{
    flags[1] = true;
    turn = 0;
    while (flags[0] && turn == 0);

    ناحیه بحرانی

    flags[1] = false;

    بخش باقیمانده

}while(true)
```

P0

```
do{
    flags[0] = true;
    turn = 1;
    while (flags[1] && turn == 1);

    ناحیه بحرانی

    flags[0] = false;

    بخش باقیمانده

}while(true)
```

راه‌حل پیترسون برای مسأله ناحیه بحرانی

- آیا سه ویژگی مربوط به راه‌حل ناحیه بحرانی در راه‌حل پیترسون وجود دارد؟
  - انحصار متقابل Mutual Exclusion
  - پیشروی Progress
  - انتظار محدود Bounded Waiting

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- منظور از سخت‌افزار، پردازنده است.
- دستورات اتمیک: یک سری دستورالعمل جدید در پردازنده‌های جدید
  - Atomic Instructions
- این دستورات برای پشتیبانی از همگام‌سازی فرآیندها ایجاد شده‌اند.
  - راه‌حل برای مسأله ناحیه بحرانی
- دستورالعمل شامل دسترسی به حافظه و انجام عملیات است.
  - کل دستورالعمل یا اجرا می‌شود و یا اجرا نمی‌شود.

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- یک نمونه از این دستورات

  test_and_set –

  - عملیاتی که این دستورالعمل انجام می‌دهد.

```
boolean test_and_set(boolean target) {
   boolean rv = target;
   target = true;

   return rv;
}
```

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- **test_and_set**
  - مقدار متغیر target را true می‌کند و مقدار قبلی آن را بر می‌گرداند.

```
boolean test_and_set(boolean  target) {
   boolean rv =  target;
   target = true;

   return rv;
}
```

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- **test_and_set**

```
a = false;

b = test_and_set(a);
```

```
a = true;

b = test_and_set(a);
```

```
a = false;

b = test_and_set(a);

c = test_and_set(a);

d = test_and_set(a);
```

```
a = true;

b = test_and_set(a);

c = test_and_set(a);

d = test_and_set(a);
```

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- محافظت از ناحیه بحرانی با دستور **test_and_set**
  - یک متغیر مشترک از نوع boolean به نام lock تعریف می‌شود.
  - مقدار اولیه این متغیر، false است.

```
lock = false;
```

P0

```
do{
   while(test_and_set(lock));

   ناحیه بحرانی P0

   lock = false;
}while(true)
```

P1

```
do{
   while(test_and_set(lock));

   ناحیه بحرانی P1

   lock = false;
}while(true)
```

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- محافظت از ناحیه بحرانی با دستور **test_and_set**

```
lock = false;
```

- چطور کار می‌کند؟

```
do{
  while(test_and_set(lock));

  ناحیه بحرانی P0

  lock = false;
}while(true)
```

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- نکات مربوط به این راه‌حل
  - انتظار مشغولی busy waiting
  - spinlock
    - lock چرخشی

```
do{
   while(test_and_set(lock));

   ناحیه بحرانی P0

   lock = false;
}while(true)
```

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- آیا سه ویژگی مربوط به راه‌حل ناحیه بحرانی در این راه‌حل وجود دارد؟
  - انحصار متقابل
  - پیشروی
  - انتظار محدود

```
do{
   while(test_and_set(lock));

   ناحیه بحرانی P0

   lock = false;
}while(true)
```

راه‌حل مسأله ناحیه بحرانی با کمک سخت‌افزار

- یک مشکل!
  - دستورات test_and_set از نوع دستورات ویژه است!
    - Privileged Instructions
      - فقط در مود هسته اجرا می‌شوند.
  - نمی‌توان از این دستورات در مود کاربر استفاده کرد.
- راه‌حل؟
  - سیستم‌عامل برای حل مسأله ناحیه بحرانی به ما کمک کند!
    - با یک سری فراخوانی سیستمی

راه‌حل مسأله ناحیه بحرانی با کمک سیستم‌عامل

- یک سری فراخوانی سیستمی (**system call**) برای اینکار وجود دارد.
- معمولاً این فراخوانی‌های سیستمی به صورت جفت هستند.
  - یک فراخوانی سیستمی در بخش ورود به ناحیه بحرانی استفاده می‌شود.
  - یک فراخوانی سیستمی در بخش خروج از ناحیه بحرانی استفاده می‌شود.
- از این فراخوانی‌های سیستمی برای کاربردهای دیگر هم می‌توان استفاده کرد.
  - تعیین ترتیب اجرای کد در چند فرآیند (یا نخ) همکار.

راه‌حل مسأله ناحیه بحرانی با کمک سیستم‌عامل

- **mutex lock**

  - mutex: MUTual Exclusion
    - انحصار متقابل
  - هدف از آن ایجاد انحصار متقابل است.
  - یک متغیر مشترک از نوع boolean با مقدار اولیه true وجود دارد.
    - معمولاً نام آن، available است.
  - دو تابع برای دسترسی به این متغیر وجود دارد.
    - تابع acquire برای گرفتن متغیر
    - تابع release برای آزاد سازی متغیر

راه‌حل مسأله ناحیه بحرانی با کمک سیستم‌عامل

- **mutex lock**

- عملیاتی که این دو تابع انجام می‌دهند به صورت زیر است.
  - تابع acquire برای گرفتن متغیر
    - اگر مقدار available برابر با false باشد، صبر می‌کند تا مقدار آن true شود.
    - سپس مقدار available را false می‌کند و از تابع خارج می‌شود.
  - تابع release برای آزاد سازی متغیر
    - مقدار available را true می‌کند.
- برای پیاده‌سازی این دو تابع، از دستورات test_and_set استفاده می‌شود.

راه‌حل مسأله ناحیه بحرانی با کمک سیستم‌عامل

- **mutex lock**

```
acquire(){
   while(!available);
   available = false;
}
```

```
release(){
   available = true;
}
```

راه‌حل مسأله ناحیه بحرانی با کمک سیستم‌عامل

- نحوه محافظت از ناحیه بحرانی با کمک **mutex**

```
do{
   acquire();

   ناحیه بحرانی

   release();

}while(true)
```

راه‌حل مسأله ناحیه بحرانی با کمک سیستم‌عامل

- نحوه محافظت از ناحیه بحرانی با کمک **mutex**

```
boolean available = true;
```

```
do{
   acquire();

   ناحیه بحرانی

   release();

}while(true)
```

```
acquire(){
   while(!available);
   available = false;
}
```

```
release(){
   available = true;
}
```