

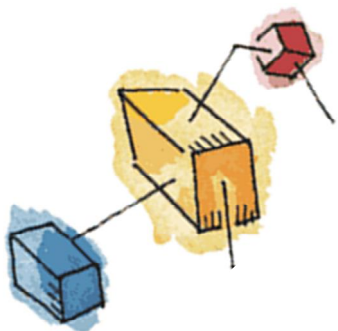
به نام خدا

فصل پنجم

همروندی:

انحصار متقابل و همگام سازی
(بخش دوم)

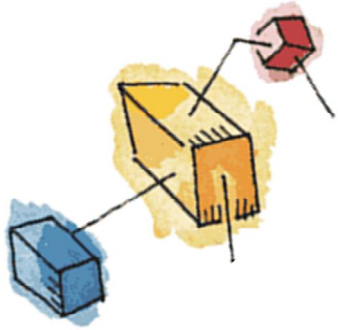
**Concurrency: Mutual Exclusion
and Synchronization**



سرفصل مطالب

- اصول همروندی (همزمانی)
- انحصار متقابل: حمایت سخت افزار
- راهنماها (سمافورها)
- ناظرها (مانیتورها)
- تبادل پیام
- مساله خوانندگان و نویسندگان



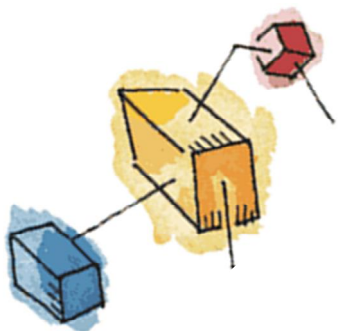


پشتیبانی سخت افزاری برای انحصار متقابل

• راه حل اول: از کار انداختن وقفه ها

- یک فرآیند تا زمانی که سرویسی از سیستم عامل را درخواست نکرده و یا با وقفه مواجه نشده است، به اجرای خود ادامه می دهد.
- در هنگام دسترسی یک فرآیند به ناحیه بحرانی، سرویس وقفه غیرفعال می شود.
- در سیستم های تک پردازنده، فرآیندها نمی توانند همزمان اجرا شوند، بلکه تنها می توانند در بین هم اجرا شوند. بنابراین، از کار انداختن وقفه، انحصار متقابل را تضمین می کند.





از کار انداختن وقفه

- ضعف های روش:

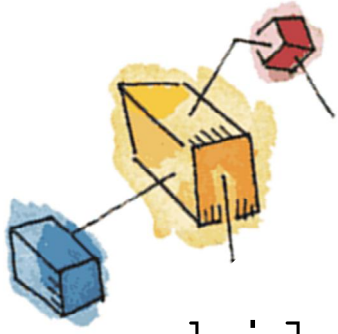
– توانایی پردازنده برای درهم سازی (interleaving) برنامه ها محدود می شود و کارایی اجرا پایین می آید.

– نامناسب برای سیستم های چند-پردازشی

- در یک سیستم چند پردازشی در هر لحظه بیش از یک فرآیند در حال اجرا است، بنابراین لازم است وقفه در تمامی پردازنده ها غیرفعال شود.

- موجب کاهش شدید کارایی سیستم می شود.

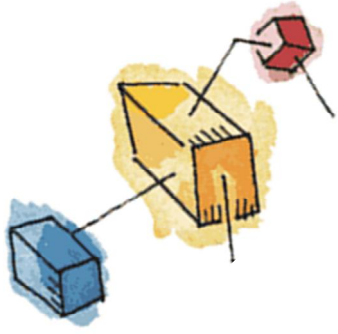




شبه کدی برای این کار

```
while (true) {  
    /* disable interrupts */;  
    /* critical section */;  
    /* enable interrupts */;  
    /* remainder */;  
}
```



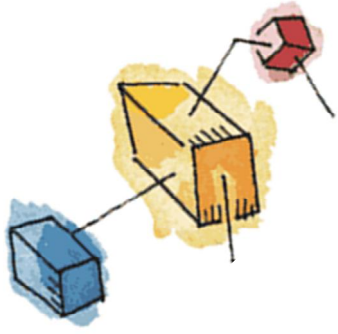


پشتیبانی سخت افزاری برای انحصار متقابل

• راه حل دوم: استفاده از دستورالعمل های ویژه سخت افزاری

- دستورالعمل های ویژه ای که به کمک آنها می توان محتوای کلمه ای از حافظه را بررسی کرد یا تغییر داد و یا با کلمه ای دیگر از حافظه تبادل نمود.
- چون این دستورالعمل ها در یک چرخه دستورالعمل واحد انجام می شوند، در معرض دخالت دستورالعمل های دیگر نیستند.
- امکان بروز وقفه بین آنها نبوده و اصطلاحاً بصورت اتمی (تفکیک ناپذیر) اجرا می شوند.
- دسترسی به محل مورد نظر از حافظه برای سایر دستورالعمل ها امکان پذیر نیست.





راه حل های سخت افزاری برای انحصار متقابل

- دستورالعمل بررسی و مقدارگذاری (Test & Set)

```
boolean testset (int i)
{
    if (i == 0)
    {
        i = 1;
        return true;
    }
    else
    {
        return false;
    }
}
```

عملکرد دستور Test & Set
به صورت این تابع است، ولی
در واقع همه این مراحل به
صورت اتمی و در یک سیکل
پردازنده انجام می شود.





راه حل های سخت افزاری برای انحصار متقابل

اعمال انحصار متقابل با استفاده از Test&Set

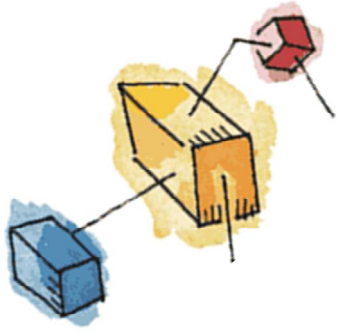
```
/* program mutualexclusion */  
const int n = /* number of processes */;  
int bolt;
```

```
void P(int i)  
{  
    while (true)  
    {  
        while (!testset (bolt))  
            /* do nothing */;  
        /* critical section */  
        bolt = 0;  
        /* remainder */  
    }  
}
```

```
void main()  
{  
    bolt = 0;  
    parbegin (P(1), P(2), . . . ,P(n));  
}
```

- صفر بودن **bolt** را به معنی باز بودن قفل تعبیر می کنیم.
- یک بودن **bolt** را به معنی بسته بودن قفل تعبیر می کنیم.





راه حل های سخت افزاری برای انحصار متقابل

- دستورالعمل تبادل یا معاوضه (Exchange)

```
void exchange(int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```





راه حل های سخت افزاری برای انحصار متقابل

اعمال انحصار متقابل با استفاده از Exchange

```
/* program mutualexclusion */  
int const n = /* number of processes*/;  
int bolt;
```

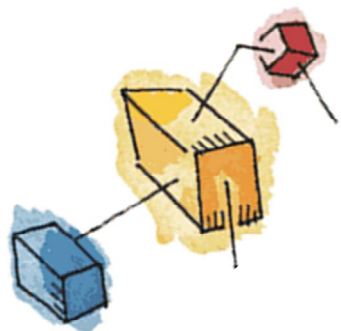
```
void P(int i)  
{  
    int keyi;  
    while (true)  
    {  
        keyi = 1;  
        while (keyi != 0)  
            exchange (keyi, bolt);  
        /* critical section */  
        exchange (keyi, bolt);  
        /* remainder */  
    }  
}
```

```
void main()  
{  
    bolt = 0;  
    parbegin (P(1), P(2), . . . , P(n));  
}
```

- صفر بودن **bolt** را به معنی باز بودن قفل تعبیر می کنیم.

- ۱ بودن **bolt** را به معنی بسته بودن قفل تعبیر می کنیم.





راه حل های سخت افزاری برای انحصار متقابل

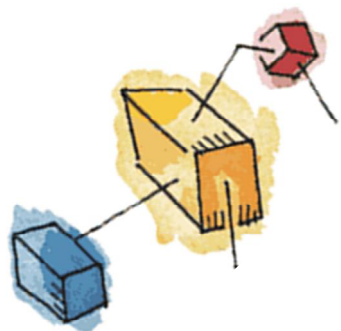
• مزایا :

– برای هر تعداد از فرآیندها، روی یک پردازنده و یا چند پردازنده، که از حافظه مشترک استفاده می کنند، قابل استفاده است.

– ساده است و بنابراین بررسی صحت آن آسان است.

– از آن برای پشتیبانی از چندین ناحیه بحرانی می توان استفاده نمود. هر ناحیه بحرانی از یک متغیر خاص خود استفاده می کند.





راه حل های سخت افزاری برای انحصار متقابل

• معایب:

– انتظار مشغولی (Busy-waiting) وقت پردازنده را هدر می دهد.

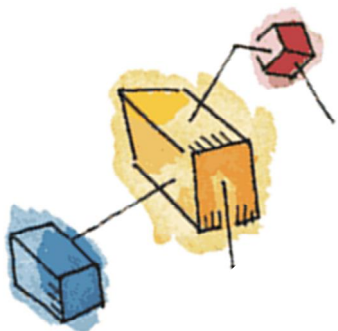
– امکان گرسنگی وجود دارد.

• هنگامی که فرآیندی بخش بحرانی خود را ترک می کند و بیش از یک فرآیند در انتظار است.

– امکان بن بست وجود دارد.

• اگر یک فرآیند با اولویت پایین در بخش بحرانی خود باشد و به یک فرآیند با اولویت بالاتر نیاز داشته باشد، و همچنین فرآیند اولویت بالاتر در انتظار ورود به بخش بحرانی باشد، بن بست رخ می دهد.

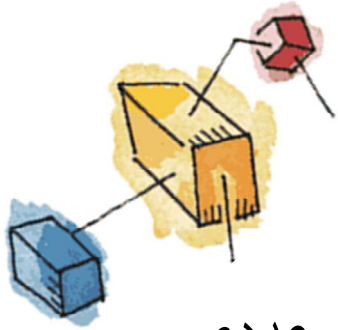




سرفصل مطالب

- اصول همروندی (همزمانی)
- انحصار متقابل: حمایت سخت افزار
- راهنماها (سمافورها)
- ناظرها (مانیتورها)
- تبادل پیام
- مساله خوانندگان و نویسندگان

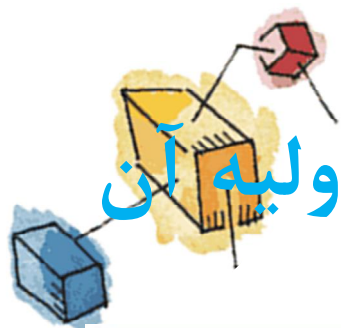




Semaphores (راهنماها)

- سمافور (راهنما) یک متغیر عددی است که می تواند با مقادیر عددی غیرمنفی مقدار دهی اولیه شود.
- تنها سه عمل را می توان بر روی سمافور انجام داد که هر سه عمل، اتمیک هستند:
 - مقداردهی اولیه با یک مقدار غیرمنفی
 - عمل Wait یا P موجب کاهش یک واحدی مقدار سمافور می شود.
 - عمل Signal یا V موجب افزایش یک واحدی مقدار سمافور می شود.
- سمافور (راهنما) متغیر ویژه ای است که برای سیگنال دهی استفاده می شود.
- اگر فرآیندی منتظر یک سیگنال باشد، تا زمان رسیدن آن سیگنال مسدود می شود.





تعریفی از سمافور شمارشی (عمومی) و عملیات اولیه آن

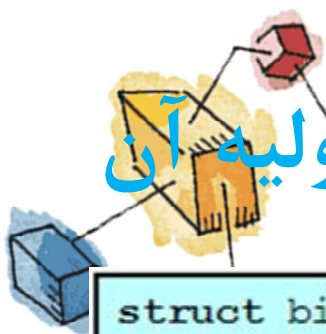
```
struct semaphore {  
    int count;  
    queueType queue;  
}
```

```
void semWait(semaphore s)  
{  
    s.count--;  
    if (s.count < 0)  
    {  
        place this process in s.queue;  
        block this process  
    }  
}
```

```
void semSignal(semaphore s)  
{  
    s.count++;  
    if (s.count <= 0)  
    {  
        remove a process P from s.queue;  
        place process P on ready list;  
    }  
}
```



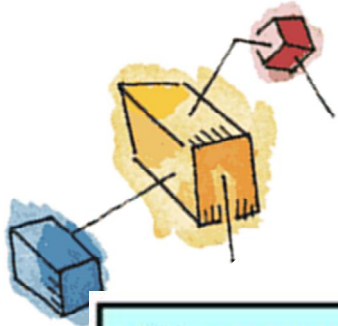
تعریفی از سمافور دودویی (Binary) و عملیات اولیه آن



```
struct binary_semaphore {  
    enum {zero, one} value;  
    queueType queue;  
};
```

```
void semWaitB(binary_semaphore s)  
{  
    if (s.value == 1)  
        s.value = 0;  
    else  
    {  
        place this process in s.queue;  
        block this process;  
    }  
}
```

```
void semSignalB(semaphore s)  
{  
    if (s.queue.is_empty())  
        s.value = 1;  
    else  
    {  
        remove a process P from s.queue;  
        place process P on ready list;  
    }  
}
```

انحصار متقابل با استفاده از سمافور

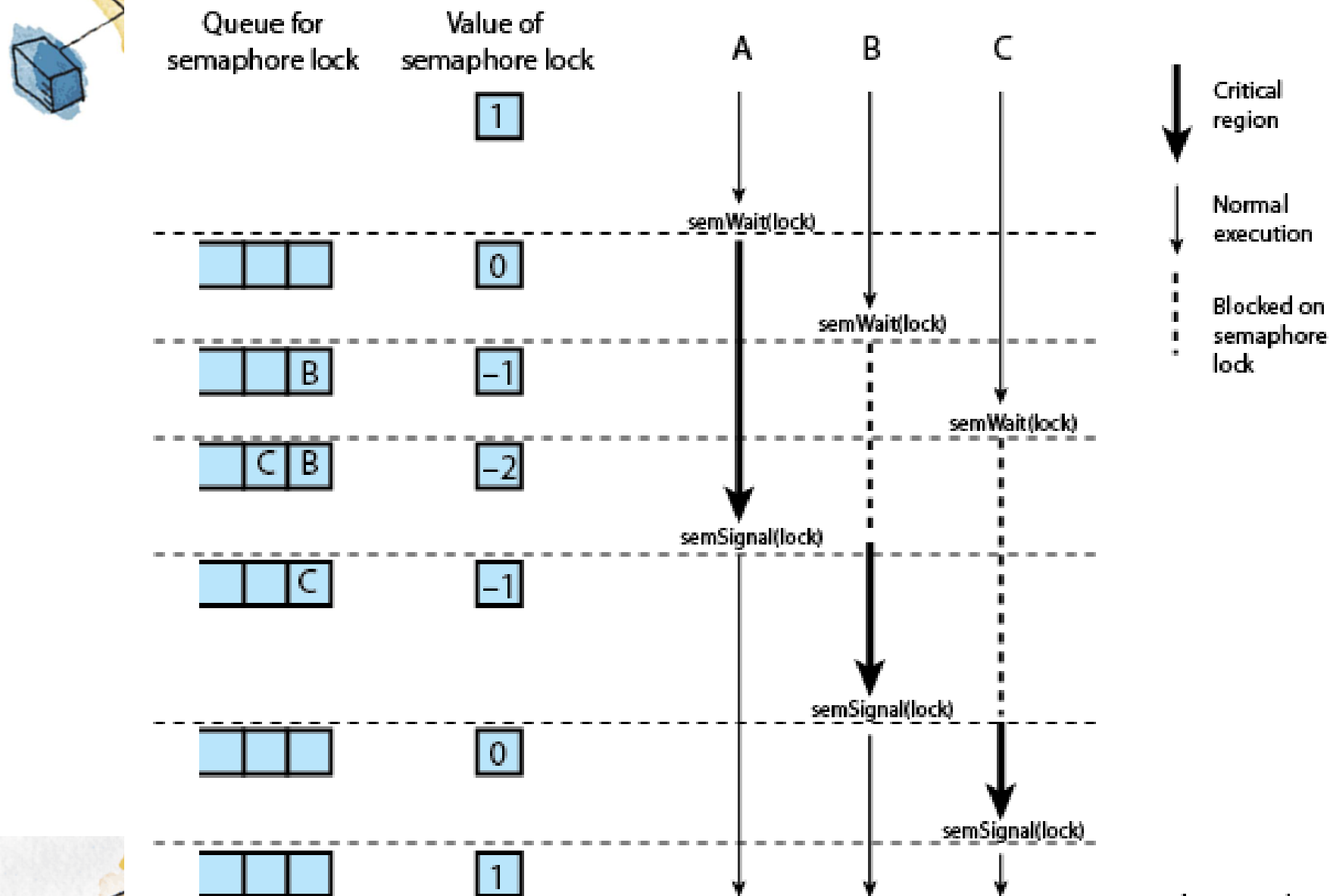
```
/* program mutualexclusion */  
const int n = /* number of processes */;  
semaphore s = 1;
```

```
void P(int i)  
{  
    while (true)  
    {  
        semWait(s);  
        /* critical section */;  
        semSignal(s);  
        /* remainder */;  
    }  
}
```

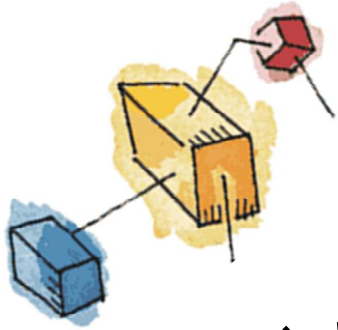
```
void main()  
{  
    parbegin (P(1), P(2), . . . , P(n));  
}
```



دسترسی فرآیند به داده های اشتراکی که توسط سمافور شمارشی محافظت شده اند



Note that normal execution can

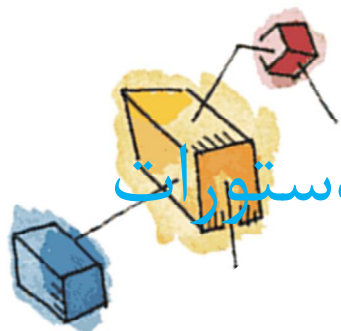


سمافورهای قوی و ضعیف

- چه در سمافورهای دودویی و چه در سمافورهای شمارشی، صفی برای نگهداری فرآیندهای منتظر آن سمافور منظور شده است.

- فرآیندها به چه ترتیبی از این صف خارج می شوند؟
 - در سمافورهای قوی (Strong Semaphores) از روش FIFO استفاده می شود.
 - در سمافورهای ضعیف (Weak Semaphores) ترتیب خروج فرآیندها مشخص نشده است.





مثالی از کاربرد سمافور برای کنترل ترتیب اجرای دستورات

- فرآیند P1 شامل دستور S1 است.
- فرآیند P2 شامل دستور S2 است.
- می خواهیم S2 پس از کامل شدن S1 اجرا شود.
- با استفاده از سمافور، این امر انجام می شود.
- مقدار اولیه سمافور synch صفر است.

S1;
Signal(synch);

Wait(synch);
S2;

