سیستم عامل

Operating Systems
فرجیان

همزمانی: انحصار متقابل و همگام سازی

مباحث این فصل:

- اصول همزمانی
- انحصار متقابل : رویکرد های نرم افزاری
- انحصار متقابل : حمایت سخت افزار
- راهنماها
- ناظرها
- تبادل پیام
- مساله خوانندگان و نویسندگان
- ...

موضوعات محوری در طراحی سیستم عامل:

- **چند برنامه ای** : مدیریت فرایندهای متعدد در داخل یک سیستم تک پردازنده ای
- **چند پردازشی**: مدیریت فرایندهای متعدد در داخل یک سیستم چند پردازنده ای
- **پردازش توزیعی** :مدیریت فرایندهای متعدد که روی سیستم های کامپیوتری متعدد و توزیع شده اجرا میشوند.

براي هر سه زمينه فوق مسئله هم زماني است.

مشکلات سیستم تک پردازنده ای:

- اشتراک منابع سراسری پر مخاطره است(دو فرآیند از متغیر سراسری)
- مدیریت تخصیص بهینه منابع به فرایندها توسط سیستم عامل دشوار است.(اگر منبعی به فرآیند داده شود و معلق و بن بست)
- توزیع وقت پردازنده در بین فرایند ها
- ....

یک مثال ساده:

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

یک مثال ساده:

Process P1

```
.
chin = getchar();
.
chout = chin;
putchar(chout);
.
.
```

Process P2

```
.
.
chin = getchar();
chout = chin;
.
putchar(chout);
.
```

موارد مهم در سیستم عامل همزمانی:

- با در نظر گرفتن مساله همزمانی فرایند ها، در طراحی سیستم عامل باید موارد زیر را در نظر داشت.
  - سیستم عامل باید بتواند فرایند های فعال را دنبال کند. (بلوک کنترل فرآیند)
  - سیستم عامل باید بتواند منابع را به فرایندها تخصیص دهد و بگیرد.
  - سیستم عامل باید داده ها و منابع هر فرایند را از دسترسی سایر فرایندها محافظت کند.
  - نتایج یک فرایند باید مستقل از سرعت پیشرفت فرایندهای دیگر باشد.

محاوره فرایندها:

- **بی اطلاعی فرایند ها از یکدیگر:** اینها فرایند های مستقل از یکدیگرند خواستار همکاری با یکدیگر نیستند.

- **اطلاع غیر مستقیم فرایند ها از یکدیگر:** اینها فرایند هایی هستند که لزوما از شناسه یکدیگر آشنا نیستند، ولی در دسترسی به بعضی اشیاء مثل بافر ورودی خروجی با یکدیگر مشترکند.(متغیر مشترک ، پرونده های مشترک)

- **اطلاع مستقیم از یکدیگر:** اینها فرایندهایی هستند که قادرند با استفاده از شناسه، با یکدیگر ارتباط برقرار کنند و برای کار مشترک بر روی بعضی فعالیت ها ساخته شده اند.

میزان اطلاع	رابطه	تاثیری که یک فرایند روی فرایند دیگری میگذارد	مسئله بالقوه کنترل
بی اطلاعی فرایند ها از یکدیگر	رقابت	• استقلال نتایج یک فرایند از عملکرد فرایندهای دیگر • امکان تاثیر در زمانگیری فرایند	• انحصار متقابل • بن بست • گرسنگی
اطلاع غیر مستقیم فرایند ها از یکدیگر	همکاری بوسیله اشتراک	• امکان وابستگی نتایج یک فرایند به اطلاعات بدست آمده از فرایند های دیگر • امکان تاثیر در زمانگیری فرایند	• انحصار متقابل • بن بست • گرسنگی • وابستگی داده ها
اطلاع مستقیم از یکدیگر	همکاری توسط ارتباط	• امکان وابستگی نتایج یک فرایند به اطلاعات بدست آمده از فرایند های دیگر • امکان تاثیر در زمانگیری فرایند	• بن بست • گرسنگی

رقابت میان فرایند ها برای منابع:

- در مورد فرایند های رقیب با سه مساله کنترلی برخورد خواهیم داشت:
  - انحصار متقابل(بخش بحرانی)
    - در هر لحظه فقط یک برنامه اجازه دارد به بخش بحرانی خود وارد شود.
    - به عنوان مثال در هر لحظه تنها یک فرایند اجازه دارد پیامی را به چاپگر بفرستد.
  - بن بست
    - : هنگام اعمال انحصار متقابل، در صورتیکه یک فرایند کنترل منبعی را در اختیار بگیرد و در انتظار منبع دیگری برای اجرا یاشد ممکن است بن بست رخ دهد.
  - گرسنگی
    - ممکن است یکی از فرایندهای مجموعه برای مدتی نامحدود از دسترسی به منابع مورد نیازش محروم بماند، چرا که سایر فرایند ها منابع را به طور انحصاری بین یکدیگر مبادله میکنند. به این حالت گرسنگی می گویند.

IASBS
1992-2012

رقابت فرآیندها برای منابع

- انحصار متقابل (Mutual Exclusion)
  - در هر لحظه فقط یک فرآیند مجاز است درون ناحیهٔ بحرانی خود حضور داشته باشد. بعنوان مثال در یک لحظه از زمان فقط یک فرآیند مجاز است برای چاپگر فرمانی ارسال کند.

```
Const int n=/*number of processes */
Void P(int i)
{
        while(true)
        {
                EnterCritical(i)
                /* Critical Section */
                ExitCritical(i)
                /* Remainder */
        }
```

رقابت فرآیندها برای منابع

- نواحی بحرانی (Critical sections)
  - فرض کنید دو یا چند فرآیند نیاز به دسترسی به یک منبع غیراشتراکی دارند.
  - قسمتی از کد فرآیند که مسئول ارسال اطلاعات یا دریافت اطلاعات از این منبع میباشد را ناحیه بحرانی مینامیم. (در این ناحیه ممکن است بحران بوجود آید.)

```
Const int n=/*number of processes */
Void P(int i)
{
        while(true)
        {
                EnterCritical(i)
                /* Critical Section */
                ExitCritical(i)
                /* Remainder */
        }
}
```

رقابت فرآیندها برای منابع

- وابستگی داده ها (Data Dependency)
  - مثال: دو فرآیند زیر مقادیر متغیرها را بصورت همگام با هم تغییر میدهند.

```
P1:
        a = a + 1;
        b = b + 1;
```

```
P2:
        b = b * 2;
        a = a * 2;
```

  - حال ترتیب اجرای زیر و نتایج ممکن آنرا در نظر بگیرید:

```
a = a + 1;
b = b * 2;
b = b + 1;
a = a * 2;
```

  - پس در همزمانی ممکن است وابستگی داده ها تحت تأثیر قرار گیرد.

ملزومات انحصار متقابل:

- از میان فرایندهایی که برای منبع یکسان یا شیء مشترکی دارای بخش بحرانی هستند، تنها یک فرایند مجاز است در بخش بحرانی خود باشد.
- فرایندی که در بخش غیربحرانی خود متوقف میشود، باید طوری عمل کند که هیچ دخالتی در عملکرد فرایند های دیگر نداشته باشد.
- برای فرایندی که نیاز به دسترسی یه یک بخش بحرانی دارد نباید امکان به تاخیر انداختن نامحدود آن وجود داشته باشد، "بن بست یا گرسنگی نمی تواند مجاز باشد.

ملزومات انحصار متقابل:

- هنگامی که هیچ فرایندی در بخش بحرانی خود نیست، هر فرایندی که متقاضی ورود به بخش بحرانی خود باشد، باید بدون تأخیر مجاز به ورود باشد.
- هیچ فرضی در مورد سرعت نسبی فرایندها یا تعداد آنها نمیتوان نوشت.
- هر فرایندی فقط برای مدت زمان محدوددی در داخل بخش بحرانی خود می ماند.

انحصار متقابل: رويکرد نرم افزاري

رويکرد نرم افزاري

را ميتوان براي فرايند هاي همزماني که روي ماشينهاي تک پردازنده اي يا چند پردازنده اي که از حافظه مشترک استفاده مي کنند پياده سازي کرد.

الگوریتم دیجسترا : تلاش اول

- اولین تلاش :

```
Process P0

.

While (turn != 0)
      /* do nothing */ ;

/* Critical Section */

turn = 1

/* Remainder */
.
.
```

```
Process P1
.

While (turn != 1)
      /* do nothing */ ;

/* Critical Section */

turn = 0

/* Remainder */
.
.
```

الگوریتم دیجسترا : تلاش اول

- هر فرایند مقدار متغیر Turn را بررسی می کند، اگر برابر شماره آن فرایند بود به بخش بحرانی خود وارد میشود.
- انتظار برای مشغولی:
  - فرایند همواره در حال چک کردن است تا ببیند آیا میتواند به بخش بحرانی خود وارد شود یا نه.
  - اگر فرایندی چه در بخش بحرانی و چه در خارج آن با شکست مواجه شود، فرایند دیگر مسدود می ماند.

- دومین تلاش :

Process P0

```
.

While ( Flag[1] )
      /* do nothing */ ;
Flag[0] = true;

/* Critical Section */

Flag[0] = false;

/* Remainder */
.
```

Process P1

```
.

While ( Flag[0] )
      /* do nothing */ ;
Flag[1] = true;

/* Critical Section */

Flag[1] = false;

/* Remainder */
.
.
```

الگوریتم دیجسترا : تلاش دوم

- در این روش از یک بردار دودویی استفاده میشود که در آن Flag[i] مربوط به فرایند i است.
- هر فرایند میتواند مقدار مربوط به فرایند دیگر را بیازماید، ولی نمیتواند آنرا تغییر دهد.
- هنگامی که فرایند میخواهد وارد ناحیه بحرانی خود شود ابتدا مقدار سایر فرایند ها را بررسی میکند.
- اگر هیچ فرایندی در بخش بحرانی خود نبود (یا Flag برای همه فرایند ها False بود) فرایند بلافاصله مقدار Flag خود را True میکند و وارد بخش بحرانی خود میشود. هنگام خروج فرایند مقدار Flag را به False برمیگرداند.
- در این صورت اگر فرایندی در ناحیه بحرانی خود شکست بخورد، فرایند دیگر تا ابد مسدود است.
- این روش انحصار متقابل را تضمین نمی کند.

■ سومین تلاش :

```
Process P0

.

Flag[0] = true;
While ( Flag[1] )
      /* do nothing */ ;

/* Critical Section */

Flag[0] = false;

/* Remainder */
.
```

```
Process P1
.

Flag[1] = true;
While ( Flag[0] )
      /* do nothing */ ;

/* Critical Section */

Flag[1] = false;

/* Remainder */
.
.
```

الگوریتم دیجسترا : تلاش سوم

- فرایند P1 قبل از بررسی سایر فرایند ها مقدار پرچم خود را برای ورود به ناحیه بحرانی می نشاند.

- هنگامی که فرایند دیگری مثل P2 در ناحیه بحرانی است و پرچم فرایند P1 ، True است فرایند P1 تا زمانی که فرایند P2 از ناحیه بحرانی خارج شود در حالت مسدود میماند.

- امکان بن بست وجود دارد. هنگامی که دو فرایند Flag خود را برای ورود به ناحیه بحرانی True میکنند، در این صورت هر فرایند باید در انتظار فرایند دیگر برای رهایی ناحیه بحرانی باشد.

- چهارمین تلاش :

```
            Process P0
.
Flag[0] = true;
While ( Flag[1] )
{
        Flag[0] = false;
        /* delay */
        Flag[0] = true;
}

/* Critical Section */

Flag[0] = false;

/* Remainder */
.
```

```
            Process P1
.
Flag[1] = true;
While ( Flag[0] )
{
        Flag[1] = false;
        /* delay */
        Flag[1] = true;
}

/* Critical Section */

Flag[1] = false;

/* Remainder */
.
```

الگوریتم دیجسترا : تلاش چهارم

- هر فرایند Flag خود را مقدار دهی میکند تا تمایل خود را برای ورود به ناحیه بحرانی نشان دهد. اما آماده است Flag خود را برای احترام به سایر فرایند ها تغییر دهد.
- سایر فرایند ها بررسی میشوند، اگر یکی از آنها در بخش بحرانی بود مقدار Flag به False باز میگردد و بعدا دوباره مقدار دهی میشود تا تمایل خود را برای ورود به ناحیه بحرانی نشان دهد. این چرخه تا زمان ورود ادامه دارد.

الگوریتم دیجسترا : تلاش چهارم

- دقت کنید که چرخه تست Flag میتواند به طور نامحدود گسترش یابد، اما این یک بن بست نیست چرا که بن بست زمانی رخ میدهد که چند فرایند بخواهند به بخش بحرانی وارد شوند ولی هیچ کدام نتوانند. به این حالت بن باز گفته میشود، چرا که هر تغییری در سرعت نسبی دو فرایند چرخه را شکسته و موجب ورود به ناحیه بحرانی میشود.

■ راه حل صحیح:

Process P0

```
.
flag[0] = true;
While ( flag[1] )
    if ( turn == 1 )
    {
        flag[0] = false;
        while( turn == 1 )
            /*do nothing*/;
        flag[0] = true;
    }

/* Critical Section */
turn = 1;
flag[0] = false;

/* Remainder */
.
```

Process P1

```
.
flag[1] = true;
While ( flag[0] )
    if ( turn == 0 )
    {
        flag[1] = false;
        while( turn == 0 )
            /*do nothing*/;
        flag[1] = true;
    }

/* Critical Section */
turn = 0;
flag[1] = false;

/* Remainder */
.
```

الگوریتم دیجسترا : یک راه حل صحیح

- در این روش هم از آرایه برداری دودویی Flag و هم از متغیر Turn استفاده میشود.
- هر فرایند برای ورود به ناحیه بحرانی ابتدا مقدار Flag خود را True میکند و سپس در انتظار مقدار Turn میماند

انحصار متقابل: حمایت از سخت افزار

- از کار انداختن وقفه ها:
  - یک فرایند تا زمانی که خدمتی از سیستم عامل را احظار نکرده و یا تا زمانی که با وقفه مواجه نشده به اجرای خود ادامه میدهد.
  - از کار انداختن وقفه، انحصار متقابل را ضمانت میکند.
  - این رویکرد در معماری چند پردازشی کار نمیکند، چرا که در یک سیستم چند پردازشی در هر لحظه بیش از یک فرایند در حال اجراست.

انحصار متقابل: حمایت از سخت افزار

- دستورالعمل های ویژه ماشین:
  - این دستورالعمل ها در یک چرخه دستورالعمل واحد انجام میشوند، و در معرض دخالت دستورالعمل های دیگر نیستند.
  - در سطح سخت افزار دسترسی به یک محل از حافظه، از دسترسی به همان محل از حافظه جلوگیری میکند.

انحصار متقابل: حمایت از سخت افزار

- دستورالعمل آزمون و مقدار گذاری:

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

انحصار متقابل: حمایت از سخت افزار

- دستورالعمل معاوضه:

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

انحصار متقابل:

```
/* 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));

}
```

(a) Test and set instruction

```
/* 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));
}
```

(b) Exchange instruction

انحصار متقابل: دستورالعمل های ویژه ماشین

- **مزایا :**
  - برای هر تعداد از فرایندها، روی یک پردازنده و یا چند پردازنده، که از حافظه مشترک استفاده میکنند، قابل به کار گیری است.
  - ساده است و بنابراین وارسی آن آسان است.
  - از آن برای حمایت از بخش های بحرانی متعدد میتوان استفاده کرد که در آن هر بخش بحرانی میتواند با متغیر خاص خود تعریف شود.

انحصار متقابل: دستورالعمل های ویژه ماشین

- معایب:
  - انتظار مشغولی وجود دارد.
  - امکان گرسنگی وجود دارد: هنگامی که فرایندی بخش بحرانی خود را ترک میکند وبیش از یک فرایند در انتظار است.
  - امکان بن بست وجود دارد: اگر یک فرایند با اولویت پایین در بخش بحرانی خود باشد و به یک فرایند با اولویت بالاتر نیاز داشته باشد، و همینطور فرایند اولویت بالاتر در انتظار ورود به بخش بحرانی باشد بن بست رخ میدهد.

راهنما ها :(Semaphore)

- راهنما، مکانیسیمی است که از دسترسی دو یا چند فرایند به منابع مشترک به طور همزمان جلوگیری میکند. به عنوان مثال در یک راه آهن راهنما از برخورد قطار ها با هم در یک ریل مشترک جلوگیری میکند. در صورتیکه به راهنمها توجهی نشود تضمینی وجود ندارد که قطار ها با هم برخورد نکنند به طور مشابه درکامپیوتر در صورت عدم توجه به راهنما احتمال اغتشاش فرایندها وجود دارد.

راهنما ها :نرم افزار

- در سال ۱۹۶۵ دیجسترا راهنما ها را به عنوان راه حلی برای فرایند های همزمان در نظر گرفت.
- اصل بنیادی این بود: دو یا چند فرایند میتوانند با علامت های ساده با یکدیگر همکاری کنند. هنگامی که یک فرایند در انتظار یک علامت از طرف فرایند دیگر است، آن فرایند تا رسیدن آن علامت در حالت معلق است.
- برای علامت دهی از متغیر های ویژه ای به نام راهنما استفاده شد

عملیات روی راهنما:

- یک راهنما میتواند با یک مقدار غیر منفی صحیح مقدار دهی اولیه شود.
- عمل Wait مقدار راهنما را یک واحد کاهش میدهد. اگر مقدار منفی شود آنگاه فرایندی که دستور Wait را اجرا کرده مسدود میشود.
- عمل Signal مقدار راهنما را یک واحد افزایش میدهد. اگر مقدار مثبت نباشد آنگاه فرایندی که توسط دستور Wait مسدود شده بود آزاد میگردد.
  - غیر این ۳ عمل راه دیگری برای دستکاری سمافور وجود ندارد.

تعریف اولیه های راهنما:

```
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;
     }
}
```

تعریف اولیه های راهنماهای دودویی

```
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));
}
```

- نام های دیگر عملگر ها در سمافور

- P – proberen/wait/down
- V – verogen/signal/up

مثالی از مکانیسم سمافور:

Figure 5.5 Example of Semaphore Mechanism

I A S B S
1 9 9 2 - 2 0 1 2

دسترسی فرایند ها به داده های مشترکی که با یک راهنما محافظت شده اند.(شکل)

Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
1
semWait(lock)
0
semWait(lock)
B
D1
semWait(lock)
C B
D2
semSignal(lock)
C
D1
semSignal(lock)
0
semSignal(lock)
1

Critical region
Normal execution
Blocked on semaphore *lock*

Note that normal execution can proceed in parallel but that critical regions are serialized.

Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

مساله تولید کننده و مصرف کننده:

- یک تولید کننده یا بیشتر نوعی داده را تولید میکند و در میانگیری قرار میدهد.
- یک مصرف کننده این اقلام را یکی یکی از میانگیر برمیدارد.
- در هر زمان تنها یک تولید کننده یا مصرف کننده میتواند به میانگیر دسترسی داشته باشد.

مساله تولید کننده و مصرف کننده:نا محدود

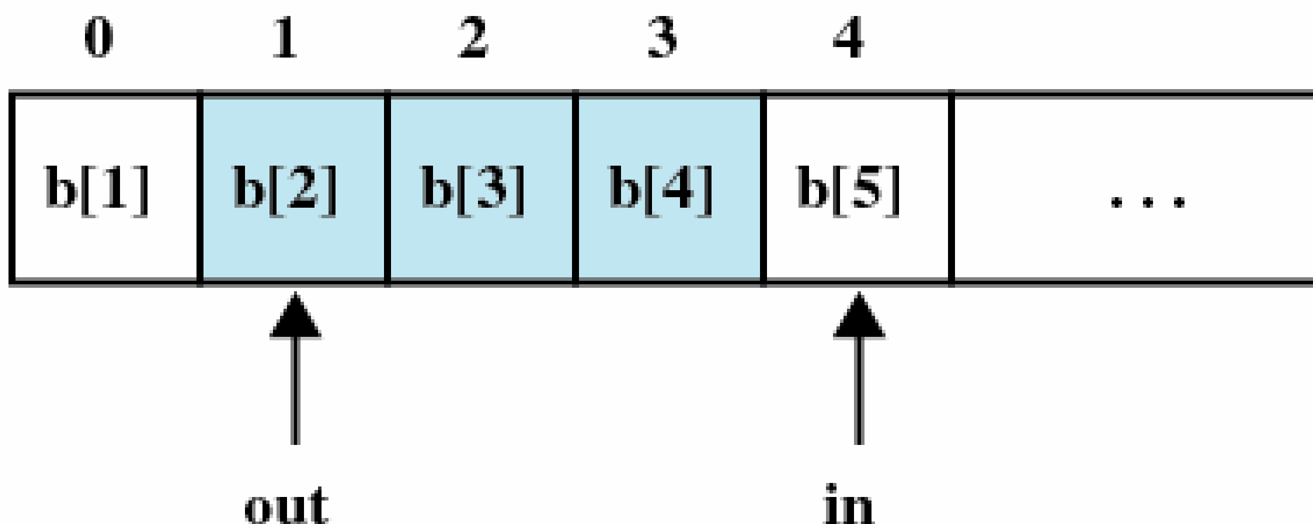
تولید کننده:

```
producer:
while (true) {
        /* produce item v */
        b[in] = v;
        in++;
}
```

مصرف کننده:

```
consumer:
while (true) {
    while (in <= out)
        /*do  nothing */;
    w = b[out];
    out++;
    /* consume item w */
}
```

مساله تولید کننده و مصرف کننده:



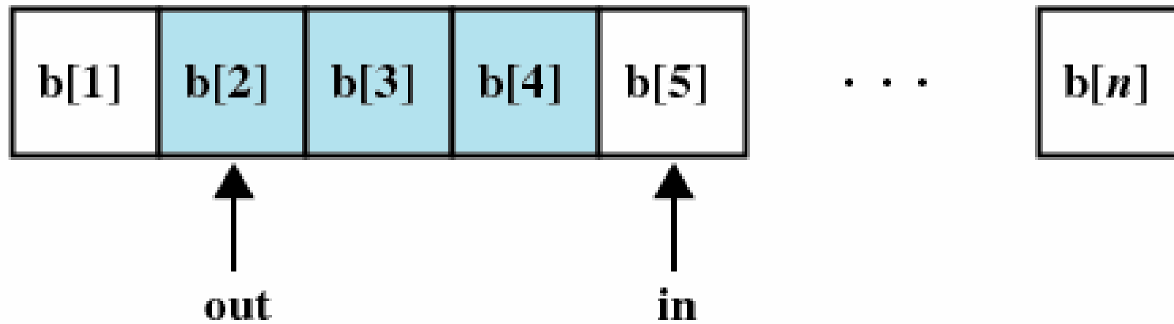
میانگیر نامحدود برای مساله تولید کننده و مصرف

مساله تولید کننده و مصرف کننده با میانگیر محدود:

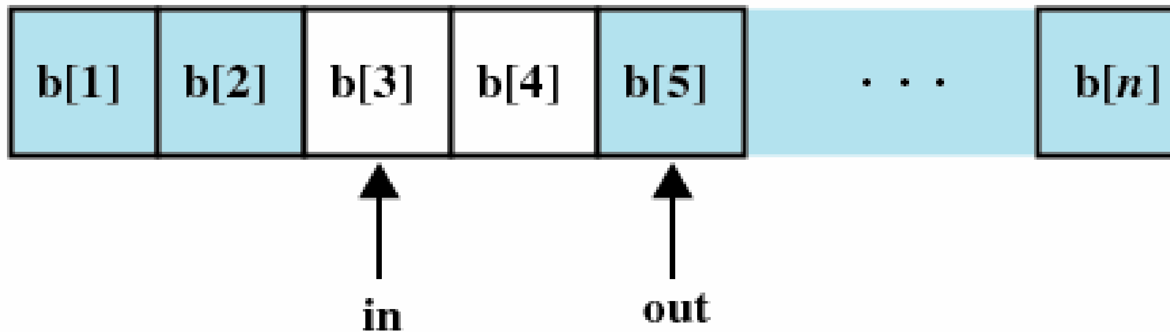
```
producer:
while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
        /* do nothing */;
    b[in] = v;
    in = (in + 1) % n
}
```

```
consumer:
while (true) {
    while (in == out)
        /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */
}
```

مساله تولید کننده و مصرف کننده با میانگیر محدود:



(a)



(b)

میانگیر محدود و مدور برای مساله تولید کننده و مصرف

حل مسئله با استفاده از راهنما

```
• #define N 100 /* Number of free slots
  */

•     semaphore mutex .count= 1;
•     semaphore empty.count= N;
•     semaphore full.count = 0;

•     void producer() {
•        item i;

•        while (1) {
•         i = produce_item();
•         P(empty);
•         P(mutex);
•         insert_new_item(i);
•         V(mutex);
•         V(full);
•        }
•     }
```

```
void consumer() {
    item i;

    while(1) {
     P(full);
     P(mutex);
     i = get_next_item();
     V(mutex);
     V(empty);
     consume_item(i);
    }
}
```

Producer consumer

```
/* program producerconsumer */
     int n;
     binary_semaphore s = 1, delay = 0;
     void producer()
     {
          while (true) {
               produce();
               semWaitB(s);
               append();
               n++;
               if (n==1) semSignalB(delay);
               semSignalB(s);
          }
     }
     void consumer()
     {
          semWaitB(delay);
          while (true) {
               semWaitB(s);
               take();
               n--;
               semSignalB(s);
               consume();
               if (n==0) semWaitB(delay);
          }
     }
     void main()
     {
          n = 0;
          parbegin (producer, consumer);
     }
```

Producer consumer Using Binary Semaphores

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
            while (true) {
                  produce();
                  semWaitB(s);
                  append();
                  n++;
                  if (n==1) semSignalB(delay);
                  semSignalB(s);
            }
      }
      void consumer()
      {
            int m; /* a local variable */
            semWaitB(delay);
            while (true)  {
                  semWaitB(s);
                  take();
                  n--;
                  m = n;
                  semSignalB(s);
                  consume();
                  if (m==0) semWaitB(delay);
            }
      }
      void main()
      {
            n = 0;
            parbegin (producer, consumer);
      }
```

A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

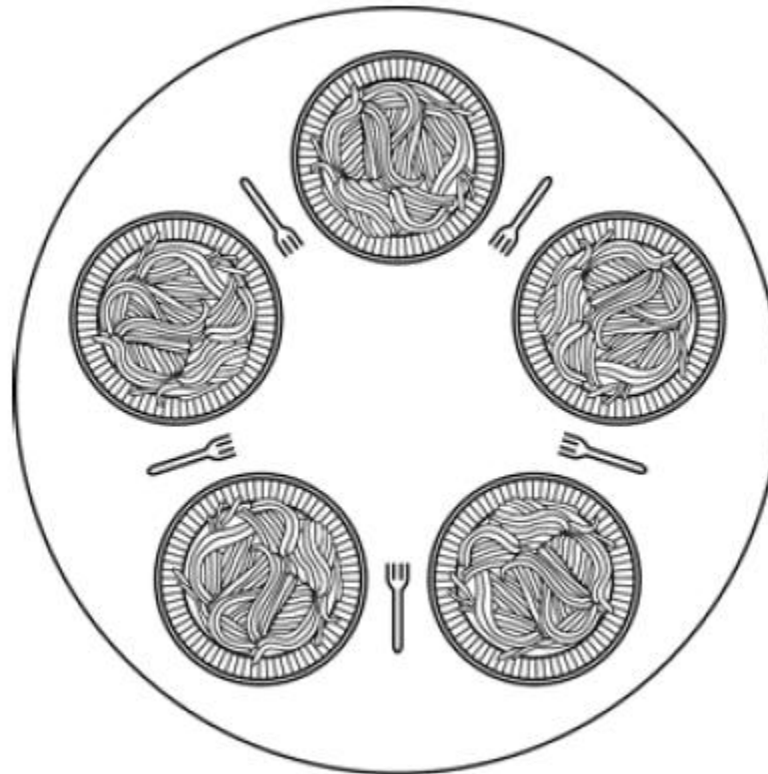
```
/* program  producerconsumer */
      semaphore n = 0, s = 1;
      void producer()
      {
           while (true) {
                produce();
                semWait(s);
                append();
                semSignal(s);
                semSignal(n);
           }
      }
      void consumer()
      {
           while (true) {
                semWait(n);
                semWait(s);
                take();
                semSignal(s);
                consume();
           }
      }
      void main()
      {
           parbegin (producer, consumer);
      }
```

Readers and writer

```
typedef int semaphore;/* use your imagination */
semaphore mutex = 1;/* controls access to 'rc' */
semaphore db = 1;/* controls access to the database */
int rc = 0;/* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {/* repeat forever */
        down(&mutex);    /* get exclusive access to 'rc' */
        rc = rc + 1;/* one reader more now */
        if (re == 1) down(&db);  /* if this is the first reader… */
        up{&mutex);/* release exclusive access to 'rc' */
        read_data_base();/* access the data */
        down(&mutex);/* get exclusive access to 'rc' */
        rc = rc - 1;        /* one reader fewer now */
        if (rc == 0) up(&db);/* if this is the last reader… */
        up(&mutex);/* release exclusive access to 'rc' */
        use_data_read();/* noncritical region */
    }
}
```

- void writer(void)
- {
- while (TRUE) {/* repeat forever */
- think_up_data(); /* noncritical region */
- down(&db);/* get exclusive access */
- write_data_base(); /* update the data */
- up(&db); /* release exclusive access */
- }

The Dining Philosophers Problem



Lunch time in the Philosophy Department.

The Dining Philosophers Problem

```
#define N 5/* number of philosophers */

void philosopher(int i)/* i: philosopher number, from 0 to 4 */
{
while (TRUE) {
 think( );     /* philosopher is thinking */
take_fork(i);     /* take left fork */
take_fork((i+1) % N);/* take right fork; % is modulo operator */
eat();     /* yum-yum, spaghetti */
put_fork(i);     /* Put left fork back on the table */
put_fork((i+1) % N);/* put right fork back on the table */
}
}
```

The Dining Philosophers Problem

```
#define N5/* number of philosophers */
#define LEFT(i+N-1)%N/* number of i's left neighbor */
#define RIGHT(i+1)%N/* number of i's right neighbor */
#define THINKING0/* philosopher is thinking */
#define HUNGRY1/* philosopher is trying to get forks */
#define EATING2/* philosopher is eating */
typedef int semaphore;/* semaphores are a special kind of int */
int state[N];/* array to keep track of everyone's state */
semaphore mutex = 1;/* mutual exclusion for critical regions */
semaphore s[N];/* one semaphore per philosopher */

void philosopher (int i)/* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {/* repeat forever */
        think();/* philosopher is thinking */
        take_forks(i);/* acquire two forks or block */
        eat();/* yum-yum, spaghetti */
        put_forks(i);/* put both forks back on table */
    }
```

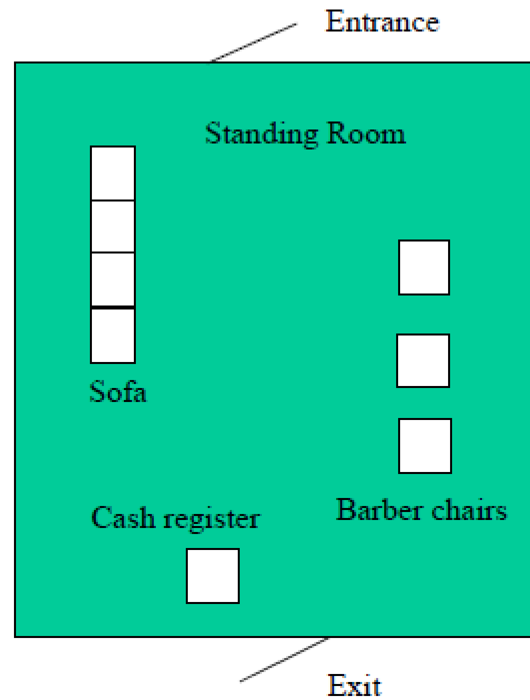
```
void take_forks(int i)/* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);     /* enter critical region */
    state[i] = HUNGRY;/* record fact that philosopher i is hungry */
    test(i);/* try to acquire 2 forks */
    up(&mutex);           /* exit critical region */
    down(&s[i]);          /* block if forks were not acquired */
}

void put_forks(i)/* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);           /* enter critical region */
    state[i] = THINKING;/* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT);/* see if right neighbor can now eat */
    up(&mutex);           /* exit critical region */
}

void test(i)/* i: philosopher number, from 0 to N-1 */
{
 if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    state[i] = EATING;
    up(&s[i]);
 }
```

The Dining Philosophers Problem

```
#define N5/* number of philosophers */
#define LEFT(i+N-1)%N/* number of i's left neighbor */
#define RIGHT(i+1)%N/* number of i's right neighbor */
#define THINKING0/* philosopher is thinking */
#define HUNGRY1/* philosopher is trying to get forks */
#define EATING2/* philosopher is eating */
typedef int semaphore;/* semaphores are a special kind of int */
int state[N];/* array to keep track of everyone's state */
semaphore mutex = 1;/* mutual exclusion for critical regions */
semaphore s[N];/* one semaphore per philosopher */

void philosopher (int i)/* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {/* repeat forever */
        think();/* philosopher is thinking */
        take_forks(i);/* acquire two forks or block */
        eat();/* yum-yum, spaghetti */
        put_forks(i);/* put both forks back on table */
    }
```

```
void take_forks(int i)/* i: philosopher number, from 0 to N-1 */
{
     down(&mutex);     /* enter critical region */
     state[i] = HUNGRY;/* record fact that philosopher i is hungry */
     test(i);/* try to acquire 2 forks */
     up(&mutex);             /* exit critical region */
     down(&s[i]);             /* block if forks were not acquired */
}

void put_forks(i)/* i: philosopher number, from 0 to N-1 */
{
     down(&mutex);            /* enter critical region */
     state[i] = THINKING;/* philosopher has finished eating */
     test(LEFT); /* see if left neighbor can now eat */
     test(RIGHT);/* see if right neighbor can now eat */
     up(&mutex);             /* exit critical region */
}

void test(i)/* i: philosopher number, from 0 to N-1 */
{
 if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
     state[i] = EATING;
     up(&s[i]);
 }
```

barbershop



barbershop

```
/* program barbershop1 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment= 0, receipt = 0;

void customer ()
{
wait(max_capacity);
enter_shop();
wait(sofa);
sit_on_sofa();
wait(barber_chair);
get_up_from_sofa();
signal(sofa);
sit_in_barber_chair;
signal(cust_ready);
wait(finished);
leave_barber_chair();
signal(leave_b_chair);
pay();
signal(payment);
wait(receipt);
exit_shop();
signal(max_capacity)
}

void barber()
{
while (true)
{
wait(cust_ready);
wait(coord);
cut_hair();
signal(coord);
signal(finished);
wait(leave_b_chair);
signal(barber_chair);
}
}

void cashier()
{
while (true)
{
wait(payment);
wait(coord);
accept_pay();
signal(coord);
signal(receipt);
}
}
```

- پایان جلسه 8