فصل پنجم

همزمانی:

انحصار متقابل و همگام سازی

مباحث این فصل:

- اصول همزمانی
- انحصار متقابل: رویکردهای 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 کامپیوتر تک پردازنده ای.
- **چند پردازشی:** مدیریت فرایندهای متعدد در داخل یک کامپیوتر چند پردازنده ای.
- **پردازش توزیعی:** مدیریت فرایند فرایندهای متعدد در روی سیستم های کامپیوتری متعدد.

برای هر سه زمینه فوق مسئله هم زمانی است.

همزمانی

در سه زمینه متفاوت طراحی می گردد:

- **کاربردهای متعدد**: زمان پردازش کامپیوتر به طور پویا بین تعدادی کار یا کاربرد بتواند تقسیم گردد.

- **کاربرد ساخت یافته**: بعضی از کاربردها را می‌توان به صورت مجموعه ای از فرایندهای همزمان،به طورکارامد برنامه سازی کرد.

- **ساختار سیستم عامل**: همین امتیاز ساختاردهی ،برای برنامه ساز سیستم نیز قابل اعمال است.

اصول همزمانی

در یک سیستم تک پردازنده‌ای و چند برنامه‌ای، فرآیندها در طول زمان در بین یکدیگر اجرا می‌شوند تا اجرای هحزمان را نشان دهند.

در سیستم های چند پردازنده ای ،نه تنها ممکن است فرایند ها در بین یکدیگر اجرا شوند،بلکه می توانند واقعا به موازات هم و با همپوشانی اجرا گردند.

در این روش مشکلات زیر پیش می آید:

۱- اشتراک منابع سراسری پر مخاطره است.

۲- مدیریت تخصیص بهینه منابع برای سیستم عامل است

۳- تعیین محل خطای برنامه سازی مشکل می شود.

همزمانی:

- همزمانی گروهی از موضوعات طراحی را در بر میگیرد:
  - ارتباط بین فرایندها
  - اشتراک منابع و رقابت برای آنها
  - همگام سازی فعالیتهای فرایند های متعدد
  - توزیع وقت پردازنده در بین فرایند ها

یک مثال ساده:

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

یک مثال ساده: (سیستم چند پردازنده‌ای)

Process P1

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

Process P2

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

نیاز اصلی برای حمایت از فرآیند های همزمان، توان اعمال انحصار متقابل است.

یعنی وقتی به یک فرآیند قدرت انجام عملی داده شد. بتوان تمام فرایندها دیگررا از این قدرت بازداشت.

ملاحظات سیستم عامل در همزمانی:

- سیستم عامل باید به تواند فرایندهای فعال مختلف را دنبال کند که این کار توسط بلوک های کنترل فرایند انجام می شود.
- سیستم عامل باید منابع را به هر یک از فرایند ها تخصیص دهد و یا باز پس بگیرد از جمله:

(وقت پردازنده، حافظه، پرونده ها، دستگاههای ورودی و خروجی)

ملاحظات سیستم عامل در همزمانی:

- سیستم عامل باید داده ها و منایع فیزیکی هر فرایند را در مقابل دخالت نا خواسته فرایند دیگر محافظت کند.

- نتایج یک فرایند باید مستقل از سرعت پیشرفت اجرای فرایند های همزمان دیگر باشد.

محاوره فرایندها:

- **بی اطلاعی فرایندها از یکدیگر:** اینها فرایند های مستقل از یکدیگرند و 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.

- **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.

همکاری فرایند ها توسط اشتراک:

- فرایندهایی را پوشش می دهند که با یکدیگر محاوره می کنند، بدون اینکه صراحتا از یکدیگر مطلع باشند.

- مسئله کنترل انحصار متقابل ، بن بست ،گرسنگی باز وجود دارد و داده در دو حالت ممکن خواندن و نوشتن مورد دسترسی قرار می گیرند و تنها مورد نوشتن باید در انحصار متقابل قرار گیرد.

همکاری فرایند ها توسط ارتباط:

- هر فرایند محیط مجزای خود را دارد و فرایند های دیگر را در بر ندارد.
- محاوره بین فرایند ها غیر مستقیم است و فرایند ها صراحتا از محل یکدیگر اطلاع ندارند.
- ارتباط را با نوعی پیام مشخص می کنند.

- مسئله بن بست و گرسنگی وجود دارد.

ملزومات انحصار متقابل:

- انحصار متقابل باید اعمال گردد: 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: رویکرد نرم افزاری

رویکرد نرم افزاری: را میتوان برای فرایند های همزمانی که روی ماشینهای تک پردازنده ای یا چند پردازنده ای که از حافظه مشترک استفاده می کنند پیاده سازی کرد.

```
while(TRUE)
{
دستورات عادی

دستورات ورود به ناحیه بحرانی

دستورات ناحیه بحرانی

دستورات خروج از ناحیه بحرانی

دستورات عادی
}
```

تلاش صفر

- یک متغیر lock برای ناحیه بخرانی وجود داشته باشد و هر فرایند قبل از ورود به ناحیه بحرانی آن را بررسی می کند، اگر برابر صفر بود آن فرایند به بخش بحرانی خود وارد میشود.
  - انحصار مقابل؟
  - بن بست؟
  - گرسنگی؟

P0

```
while (true)
{
……..
while(lock==true);
lock=true;
Critical Section;
lock=false;
……..
}
```

P1

```
while (true)
{
……..
while(lock==true);
lock=true;
Critical Section;
lock=false;
……..
}
```

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.

P0

```
while (true)
{
……..
while(turn!=0);
Critical Section;
turn=1;
……..
}
```

P1

```
while (true)
{
……..
while(turn!=1);
Critical Section;
turn=0;
……..
}
```

الگوریتم دیجسترا : تلاش اول

- ساختاری که در بالا گفته شد ساختار **همروال** است
  - هم روال ها برای این طراحی شدند تا بتوانند کنترل اجرا را بین یکدیگر عقب و جلو ببرند.
  - این فن تنها برای ساخت دادن به یک فرایند واحد است، و **برای حمایت از پردازش همزمان کافی نیست**

- **اشکالات:**
  - فرایندها باید دقیقا متناوب عمل کنند
  - اگر یکی از فرآیندها دچار خطا شود دیگری تا ابد مسدود خواهد بود

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.

الگوریتم دیجسترا : تلاش دوم

- این روش انحصار متقابل را تضمین نمی کند.

P0

```
while (true)
{
……..
while(flag[1]!=false);
flag[0]=true;
Critical Section;
flag[0]=false;
……..
}
```

P0

```
while (true)
{
……..
while(flag[0]!=false);
flag[1]=true;
Critical Section;
flag[1]=false;
……..
}
```

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.

P0

```
while (true)
{
……..
flag[0]=true;
while(flag[1]!=false);
Critical Section;
flag[0]=false;
……..
}
```

P1

```
while (true)
{
……..
flag[1]=true;
while(flag[0]!=false);
Critical Section;
flag[1]=false;
……..
}
```

الگوریتم دیجسترا : تلاش سوم

- 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 انتظاز فرایند دیگر برای رهایی ناحیه بحرانی باشد.

P0

```
while (true)
{
……..
flag[0]=true;
while(flag[1]!=false);
Critical Section;
flag[0]=false;
……..
}
```

P1

```
while (true)
{
……..
flag[1]=true;
while(flag[0]!=false);
Critical Section;
flag[1]=false;
……..
}
```

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.

```
/* PROCESS 0 */
.
.
flag[0] = true;
while (flag[1])
{
   flag[0] = false;
   /*delay */;
   flag[0] = true;
}
/*critical section*/;
flag[0] = false;
.
```

```
/* PROCESS 1 */
.
.
flag[1] = true;
while (flag[0])
{
   flag[1] = false;
   /*delay */;
   flag[1] = true;
}
/* critical section*/;
flag[1] = false;
.
```

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.

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

```
void P0()
{
    while (true)
    {
        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 */;
    }
}
```

```
void P1( )
{
    while (true)
    {
        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 Peterson :

آرایه سراسری flag نمایانگر وضع هر فرایند نسبت به انحصار متقابل است و متغییر سراسری turn در گیریهای همزمانی را حل می کند.

P0

```
while (true)
{
……..
flag[0]=true;
turn=0;
while(flag[1]==true  && turn==0);
Critical Section;
flag[0]=false;
……..
}
```

P0

```
while (true)
{
……..
flag[1]=true;
turn=1;
while(flag[0]==true  && turn==1);
Critical Section;
flag[1]=false;
……..
}
```

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 چون بخش بحرانی نمی تواند وقفه داده شود.
  - پردازنده محدود به قابلیت در همگذاری برنامه هاست.
  - 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.
  - کارایی اجرایی به طور متقابل کم می شود.

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

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 توسط دیگر فرآیندها 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 و بیش 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 (Semaphore):

- 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

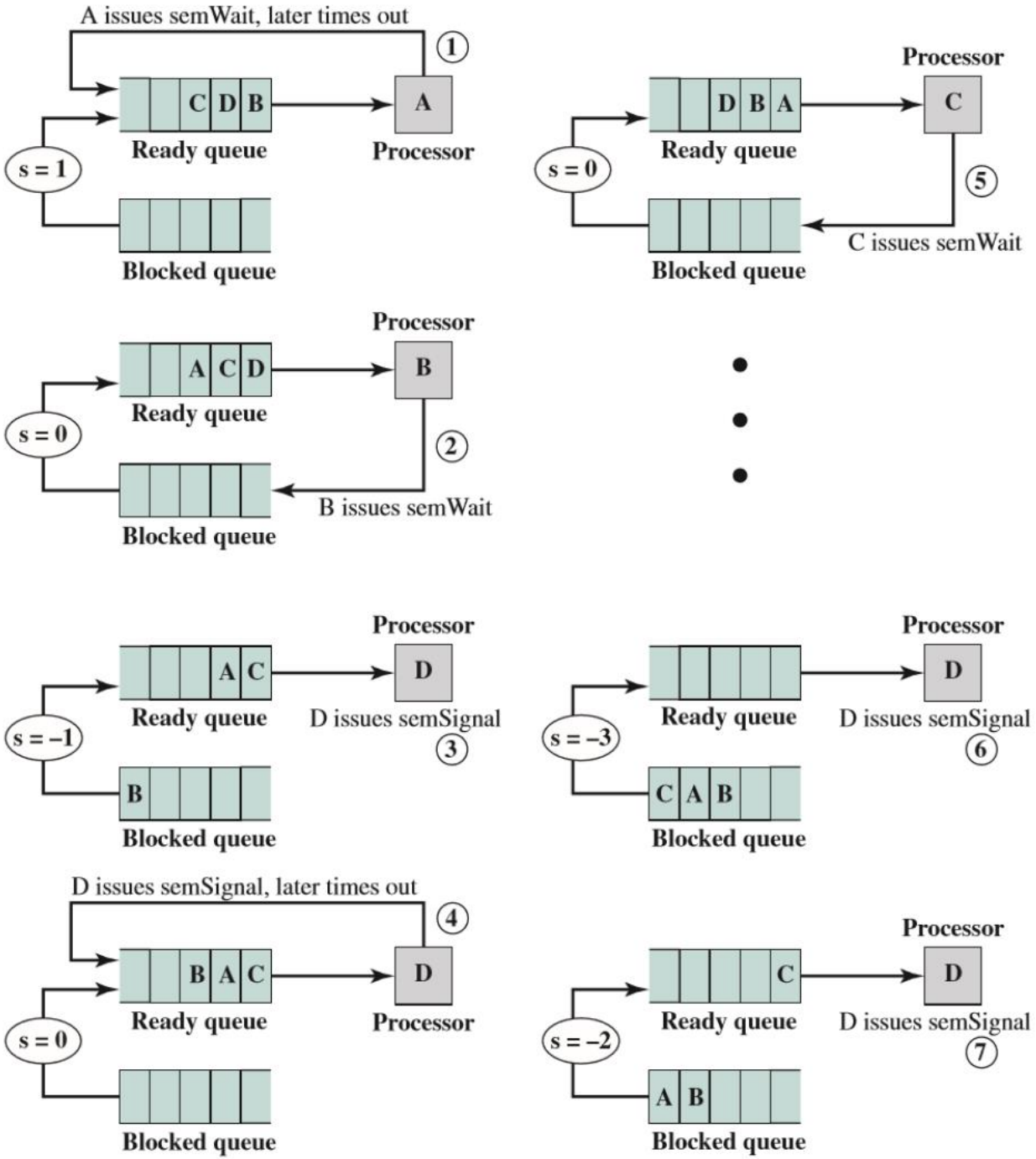
```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};

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));
}
```

مثالی از مکانیسم سمافور:



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

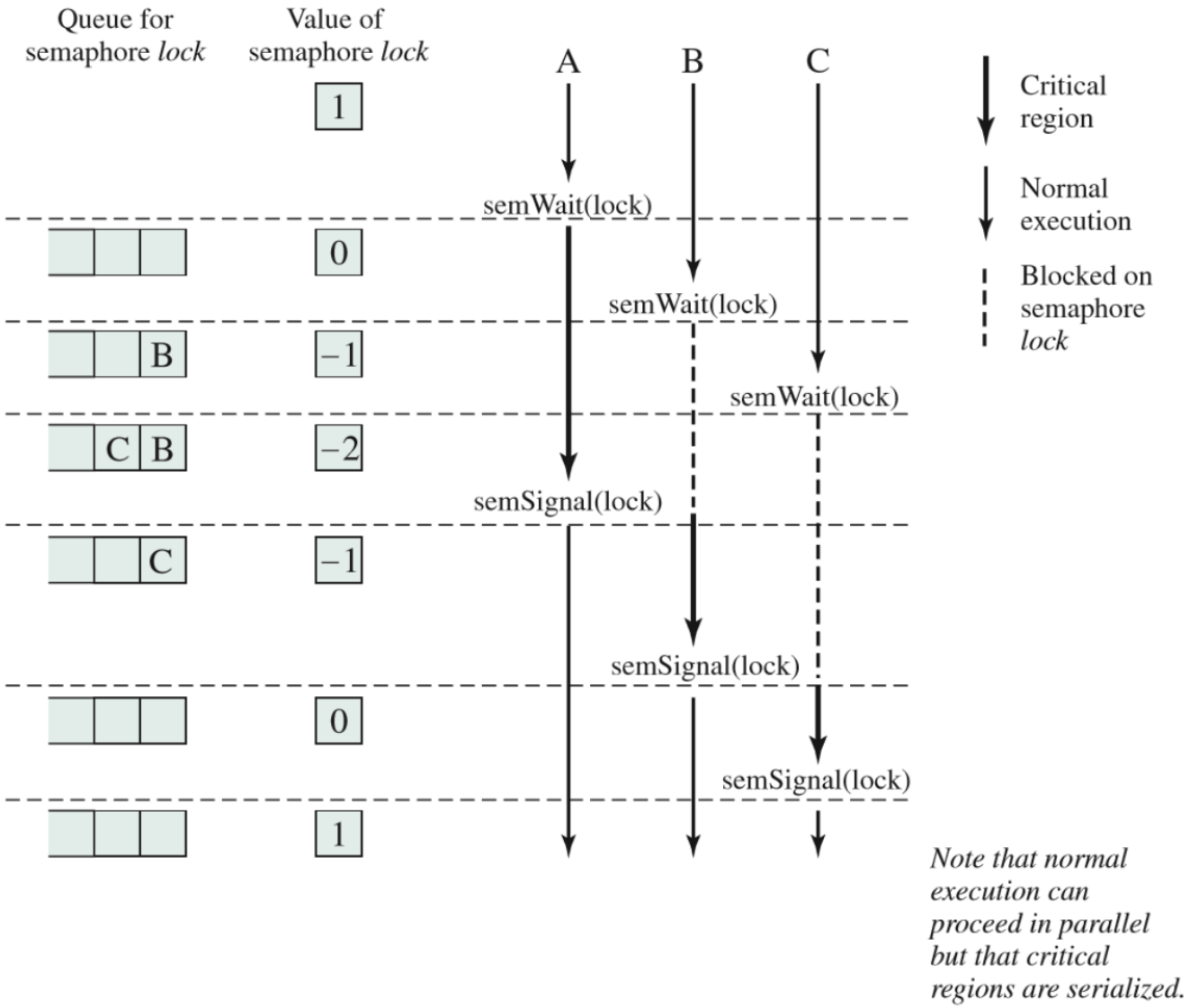


Figure 5.7 **Processes Accessing Shared Data Protected by a Semaphore**

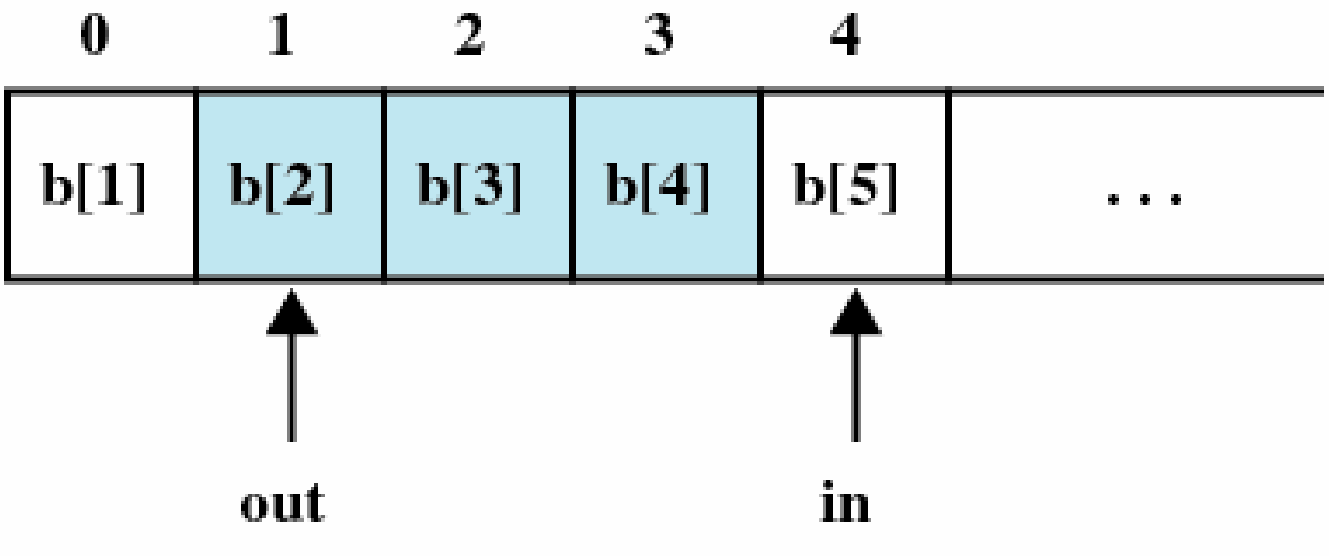
پیاده‌سازی راهنما

- اعمال wait و signal باید به صورت تجزیه ناپذیر پیاده سازی شوند.
- در هر لحظه فقط یک فرآیند حق دستکاری یک راهنما را دارد (انحصار متقابل)
- استفاده از راه حل‌های سخت افزاری و نرم افزاری

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:



Note: shaded area indicates portion of buffer that is occupied

میانگیر نامحدود برای مساله تولید کننده و مصرف

مساله تولید کننده و مصرف کننده:

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 */
}
```

پیاده‌سازی با راهنمای دودویی(راه حل نادرست)

```
/* 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);
      }
```

پیاده‌سازی با راهنمای دودویی(راه حل نادرست)

	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		semSignalB(s)	1	–1	0

پیاده‌سازی با راهنمای دودویی (راه حل درست)

```
/* 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);
      }
```

پیاده‌سازی با راهنمای معمولی (راه حل درست)

```
/* 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);
      }
```

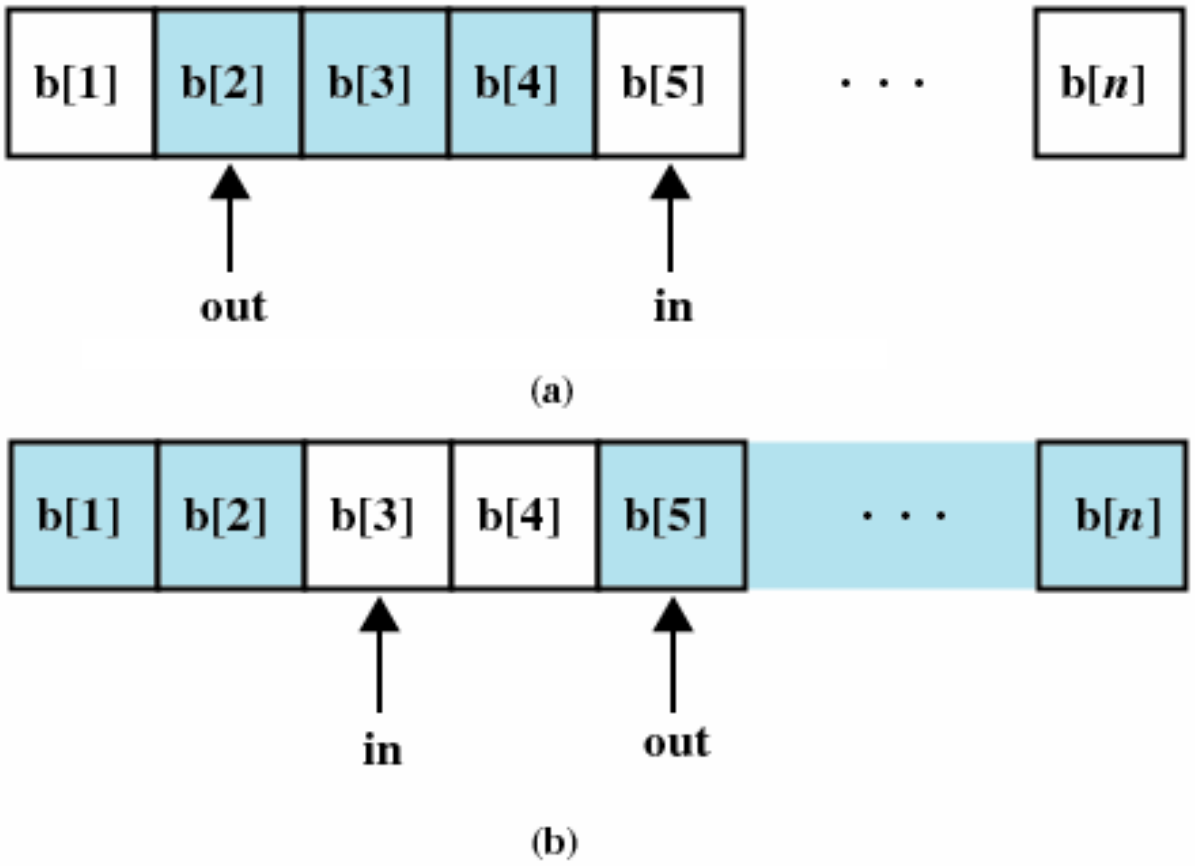
مساله تولید کننده و مصرف کننده با میانگیر نامحدود:

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

```
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 نامحدود:



میانگیر محدود و مدور برای مساله تولید کننده و مصرف

پیاده‌سازی با راهنمای معمولی

```
/* program boundedbuffer */
      const int sizeofbuffer = /* buffer size */;
      semaphore s = 1, n = 0, e = sizeofbuffer;
      void producer()
      {
            while (true) {
                  produce();
                  semWait(e);
                  semWait(s);
                  append();
                  semSignal(s);
                  semSignal(n);
            }
      }
      void consumer()
      {
            while (true) {
                  semWait(n);
                  semWait(s);
                  take();
                  semSignal(s);
                  semSignal(e);
                  consume();
            }
      }
      void main()
      {
                  parbegin (producer, consumer);
      }
```

ناظر (Monitor)**:**

- مجموعه ای از رویه ها، متغیر ها، و ساختمان داده ها میباشد.که همگی در یک ماژول نرم افزاری خاص قرار گرفته اند.
- پردازش ها در هر زمان میتوانند زیربرنامه های داخل مانیتور را فراخوانی کرده و اجرا کنند.

ویژگیهای مهم ناظر :

- متغیر ها و داده های محلی ناظر تنها برای رویه های خود ناظر قابل دسترس است و هیچ رویه دیگری به آنها دسترسی ندارد.
- یک فرایند با احضار یکی از رویه های ناظر وارد آن میشود.
- در هر زمان تنها یک فرایند میتوان در ناظر در حال اجرا باشد، فرایند های دیگری که ناظر را احضار کرده اند تا فراهم شدن ناظر معلق خواهند ماند.

توابع ناظر:

ناظر بدون علامت

Cwait(c) : اجرای یک فرایند صدا کننده را روی شرط c معلق می کند.

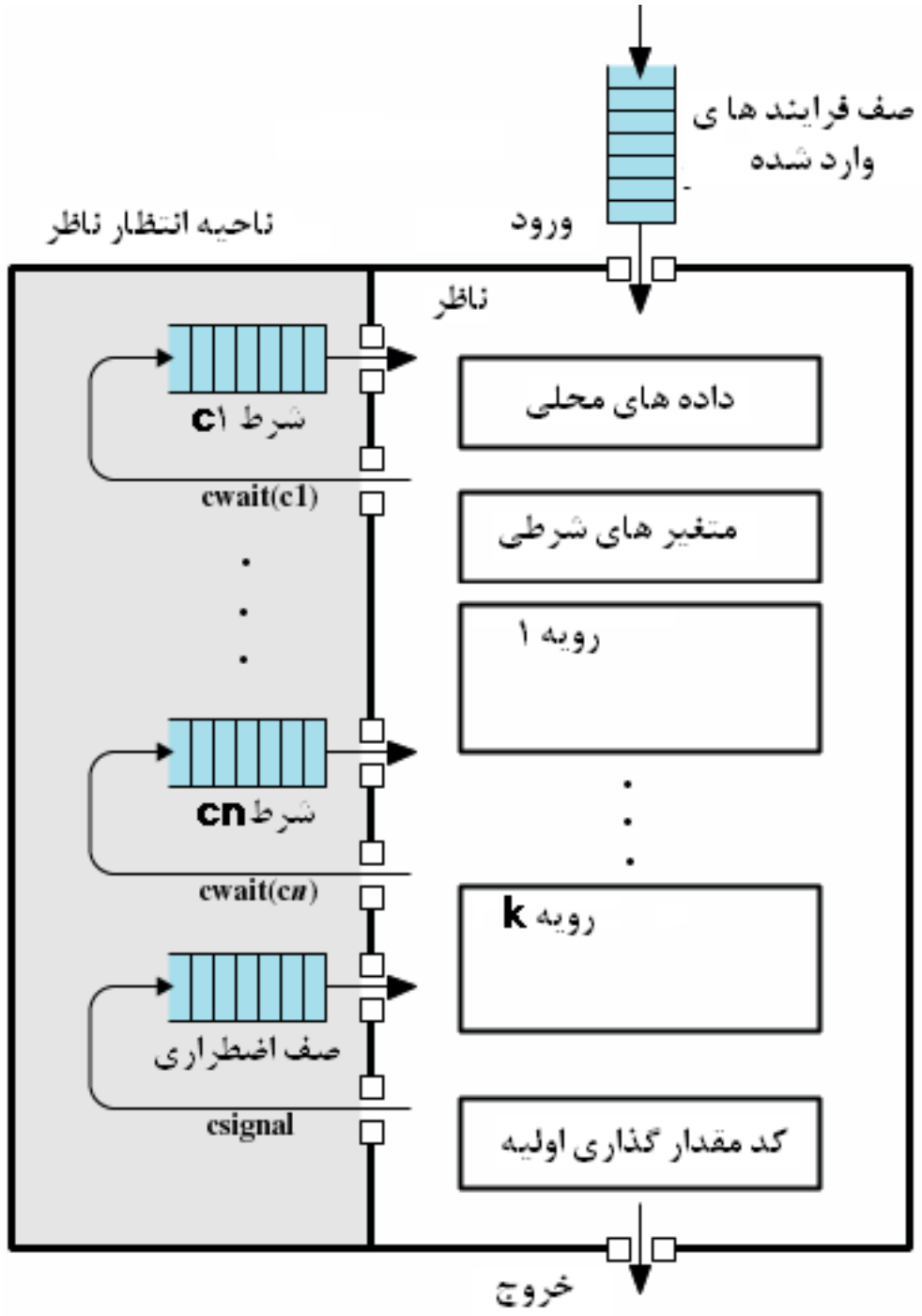
Csignsl(c) : اجرای یک فرایندی که بعد از یک عمل cwait ، روی همان شرط معلق بوده است را از سر می گیرد.

ناظر با اعلام :

Cnotify(c) : یک فرایندی که بعد از یک عمل cwait مسدود بوده است را آزاد میکند.

Cbroadcast(c) : تمام فرایندهایی که بعد از یک عمل cwait بر روی شرط مسدود بوده است را آزاد میکند.

ساختار ناظر:


صف فرایند ها ی
وارد شده
ناحیه انتظار ناظر
ورود
ناظر
داده های محلی
شرط c1
cwait(c1)
متغیر های شرطی
رویه ۱
شرط cn
cwait(cn)
رویه k
صف اضطراری
csignal
کد مقدار گذاری اولیه
خروج

تولیدکننده/مصرف کننده با ناظر

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
      take(x);
      consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

ناظر بدون اعلام

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                        /* space for N items */
int nextin, nextout;                                    /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;              /* condition variables for synchronization */
void append (char x)

{
      if (count == N) cwait(notfull);        /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (notempty);                          /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);     /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                        /* one fewer item in buffer */
      csignal (notfull);                           /* resume any waiting producer */
}
{                                                                 /* monitor body */
      nextin = 0; nextout = 0; count = 0;             /* buffer initially empty */
}
```

ناظر با اعلام

```
void append (char x)
{
      while (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;                                    /* one more item in buffer */
      cnotify(notempty);                        /* notify any waiting consumer */
}

void take (char x)
{
      while (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                    /* one fewer item in buffer */
      cnotify(notfull);                         /* notify any waiting producer */
}
```

تبادل پیام:

- در یک نتیجه گیری کلی میتوان گفت سمافور ها سطح پایین هستند، و کار کردن با آنها مشکل است.مانیتور ها نیز به جز چند زبان برنامه نویسی غیر معروف غیر قابل استفاده اند.علاوه بر این هیچ کدام از این دو، امکانات لازم برای تبادل اطلاعات بین کامپیوتر ها در کامپیوتر های توزیع شده را ندارند بنابرین تکنیک دیگری به نام تبادل پیام ابداع شد.
- پیام ها یک مکانیزم ساده و مناسب جهت همگام سازی و ارتباط دهی بین فرایندها در یم محیط غیرمتمرکز و توزیع شده اند.
- بسیاری از سیستم عامل های چند برنامگی از نوعی پیام های بین فرایند ها پشتیبانی میکنند.

تبادل پیام:

- پیام مجموعه ای از اطلاعات است که بین فرایندهای ارسال کننده و دریافت کننده مبادله میشود.
- قالب پیام قابل انعطاف و قابل تبادل توسط هر زوج گیرنده فرستنده است.

```
send (destination, message)
receive (source, message)
```

خلاصه ای از مسائل مربوط به پیام

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queueing Discipline
- FIFO
- Priority

همگام سازی:

- فرستنده و گیرنده میتوانند مسدود باشند یا نباشند(منتظر برای پیام)
- مسدود شدن فرستنده، مسدود شدن گیرنده:
  - هم فرستنده و هم گیرنده تا زمانی که پیام تحویل داده شود مسدودند.
  - گاهی به آن قرار ملاقات هم گفته میشود.
  - این ترکیب همگام سازی محکم بین فرایندها را میسر میکند.

همگام سازی:

- مسدود نشدن فرستنده، مسدود شدن گیرنده:
  - فرستنده به کار خود ادامه میدهد.
  - گیرنده تا زمانی که پیام تحویل داده شود، مسدود است.
  - این مفید ترین ترکیب است، چرا که اجازه میدهد یک پیام یا بیشتر در اسرع وقت به مقصد های متنوع ارسال شود.
- مسدود نشدن فرستنده، مسدود نشدن گیرنده
  - انتظار هیچ یک از دو طرف ضروری نیست

آدرس دهی:

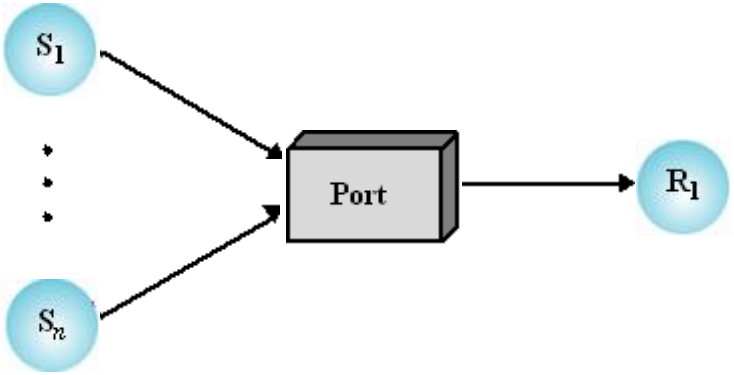
- آدرس دهی مستقیم:
  - اولیه Send شامل شناسه مشخص فرایند مقصد است
  - اولیه Receive میتوان به یکی از دو صورت زیر داده شود:
    - فرایند گیرنده صراحتاً فرایند فرستنده را تعیین کند بنابراین فرایند باید از قبل بداند از کدام فرایند باید انتظار پیام داشته باشد.
    - مشخص کردن فرایند مبدأ مورد انتظار غیر ممکن است در این حالت پارامتر مبدأ از اولیه Receive دارای مقداریست که با انجام عمل دریافت برگشت داده میشود.

آدرس دهی:

- آدرس دهی غیر مستقیم:
  - پیامها مستقیماً از فرستنده به گیرنده ارسال نمیشوند.بلکه به یک ساختمان داده مشترک که شامل صفهایی است که میتوانند پیامها را به طور دائمی نگه دارند ارسال میشود.
  - صف ها معمولاً صندوقهای پستی (Mail box) نامیده میشوند.
  - یک فرایند پیام را به صندوق پستی ارسال میکند و فرایند دیگر آن را از صندوق پستی بر میدارد.

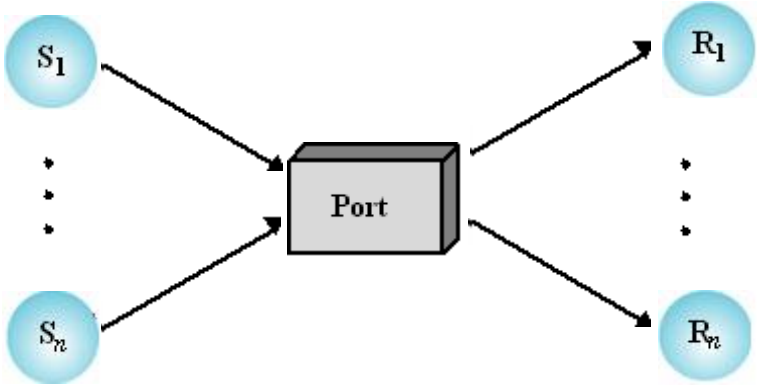
ارتباط غیر مستقیم فرایند ها:

چند به یک

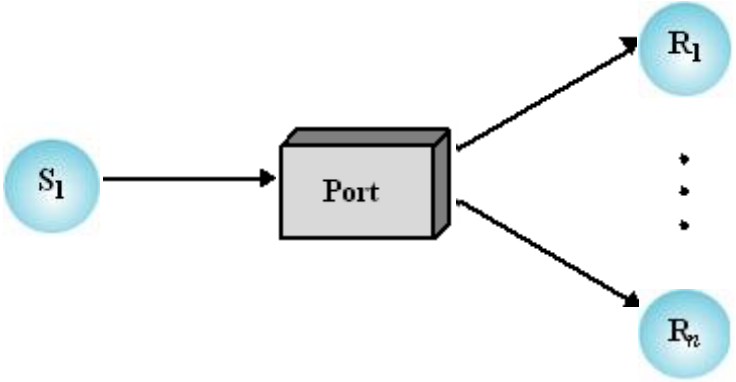


یک به یک

چند به چند



یک به چند

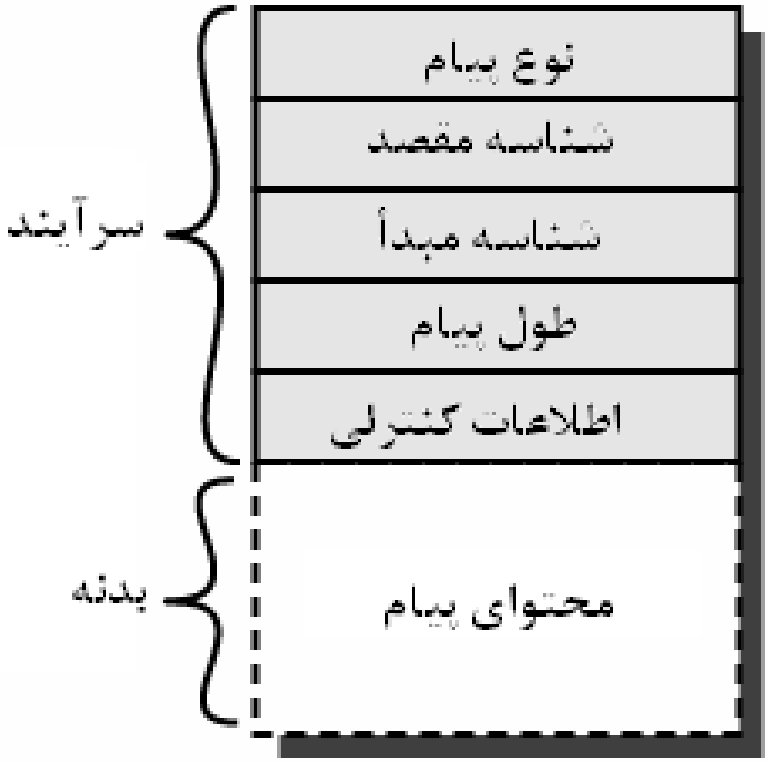


قالب پیام:

- قالب پیام به اهداف امکانات پیام دهی و اینکه این امکانات روی یک کامپیوتر یا یک سیستم توزیعی اجرا میشوند بستگی دارد.
- بعضی سیستم عاملها برای حداقل کردن سربار پردازش و حافظه پیامهای کوچک با طول ثابت را ترجیح داده اند.
- قالب کلی پیام های طول متغیربه دو بخش تقسیم میشود:
  - سرآمد: حاوی اطلاعات مربوط به پیام
  - بدنه: حاوی خود پیام

قالب پیام:

- سرآیند:
  - اطلاعات مبدأ و مقصد مورد نظر پیام
  - طول پیام
  - اطلاعات کنترلی
    - اشاره گر برای ایجاد لیستی از پیام ها
    - شماره ترتیب برای پیگیری تعداد و ترتیب انتقال پیامها
- بدنه:



مسأله خوانندگان و نویسندگان:

- در این مسأله شرایط زیر همواره برقرار است:
  - هر تعدادی از خوانندگان میتوانند از پرونده بخوانند.
  - در هر زمان فقط یک نویسنده میتواند در پرونده بنویسد.
- هر گاه یک نویسنده در حال نوشتن بر روی پرونده است، هیچ خواننده ای امکان خواندن را ندارد.

پیاده‌سازی با راهنما اولویت خوانندگان

```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
     while (true){
      semWait (x);
      readcount++;
      if(readcount == 1)
          semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if(readcount == 0)
          semSignal (wsem);
      semSignal (x);
     }
}
void writer()
{
     while (true){
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
     }
}

void main()
{
     readcount = 0;
     parbegin (reader,writer);
}
```

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority

```
/* program readersandwriters */
int readcount,writecount; semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true){
      semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
                    semSignal (x);
               semSignal (rsem);
          semSignal (z);
          READUNIT();
          semWait (x);
               readcount--;
               if (readcount == 0) semSignal (wsem);
          semSignal (x);
    }
}
void writer ()
{
    while (true){
      semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
      semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

پیاده‌سازی با راهنما
اولویت نویسندگان

پیاده‌سازی با پیام

```
void reader(int i)
{
    message rmsg;
        while (true) {
            rmsg = i;
            send (readrequest, rmsg);
            receive (mbox[i], rmsg);
            READUNIT ();
            rmsg = i;
            send (finished, rmsg);
        }
}
void writer(int j)
{
    message rmsg;
    while(true) {
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}
```

```
void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (writerequest)) {
                receive (writerequest, msg);
                writer_id = msg.id;
                count = count - 100;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                send (msg.id, "OK");
            }
        }
        if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}
```

Figure 5.24 A Solution to the Readers/Writers Problem Using Message Passing