سازمان سما

وابسته به دانشگاه آزاد اسلامی

دانشگاه سما واحد حاجی آباد

سیستم عامل

منبع : سیستم عامل دکتر شیر افکن

حمیدرضا رضاپور

WWW.HREZAPOUR.IR

فصل پنجم
(قسمت دوم)

سمافور

سمافور

راه حل های نرم افزاری و سخت افزاری که بررسی کردیم ، دارای نقاط ضعف زیادی بودند.

سمافور قدرت زیادی در برقراری انحصار متقابل دارد و از عهده انواع مختلف مسایل همگام سازی بر می آید.

سمافور یک ساختار شامل فیلدهای زیر است:

۱- شمارنده صحیح (count)

شمارش تعداد wakeup هایی که می خواهند هدر بروند.

(ذخیره سیگنال ها برای استفاده های بعدی)

۲- صف (queue)

نگهداری فرایندهای بلوکه شده بر روی سمافور .

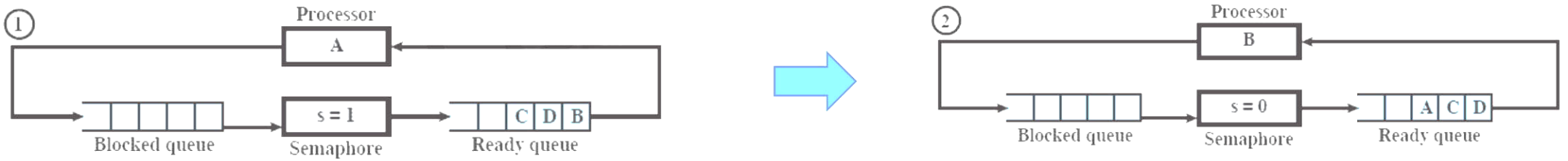
تعريف سمافور

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

تابع wait

تابع wait، یک واحد از شمارنده کم کرده و اگر منفی شود، فرایند در صف، مسدود می شود.

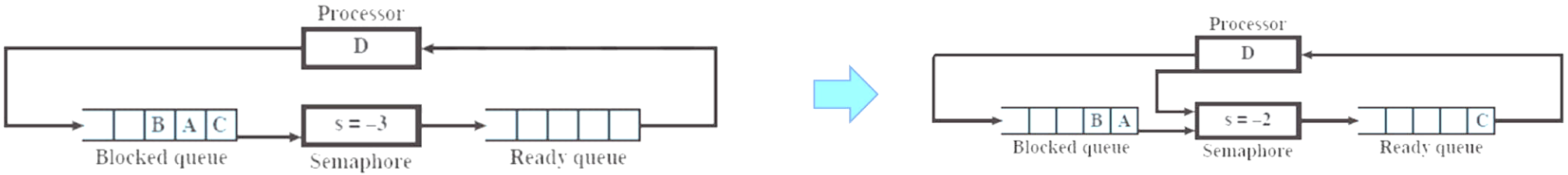
```
void  wait(semaphore s){
    s.count = s.count –1;
    if (s.count < 0){
     place  this process  in  s.queue;
     block  this  process;
    }
}
```



تابع signal

تابع **signal**، یک واحد به شمارنده اضافه می کند، اگر مقدار شمارنده بیشتر از صفر نشود، یک فرایند از قبل مسدود شده در صف، آزاد می شود.

```
void  signal(semaphore s){
   s.count = s.count +1;
   if (s.count <= 0 ){
     remove  a  process  from  s.queue;
     place  this  process  in  ready queue;
   }
}
```



تذکر: در بعضی از متون از **down** (یا **P**) به جای **wait** و از **up** (یا **V**) به جای **signal** استفاده می شود.

انحصار متقابل با استفاده از سمافورها

انحصار متقابل برای دو فرایند P1 و P2:

```
semaphore  mutex =1;
void  p(int  i){
    while(TRUE) {
        wait(mutex);
        critical-section( );
        signal(mutex);
        non-critical-section( );
    }
}
```

پیاده سازی انحصار متقابل به کمک سمافور، شرط های انحصار متقابل، پیشرفت و انتظار محدود را برآورده می کند.

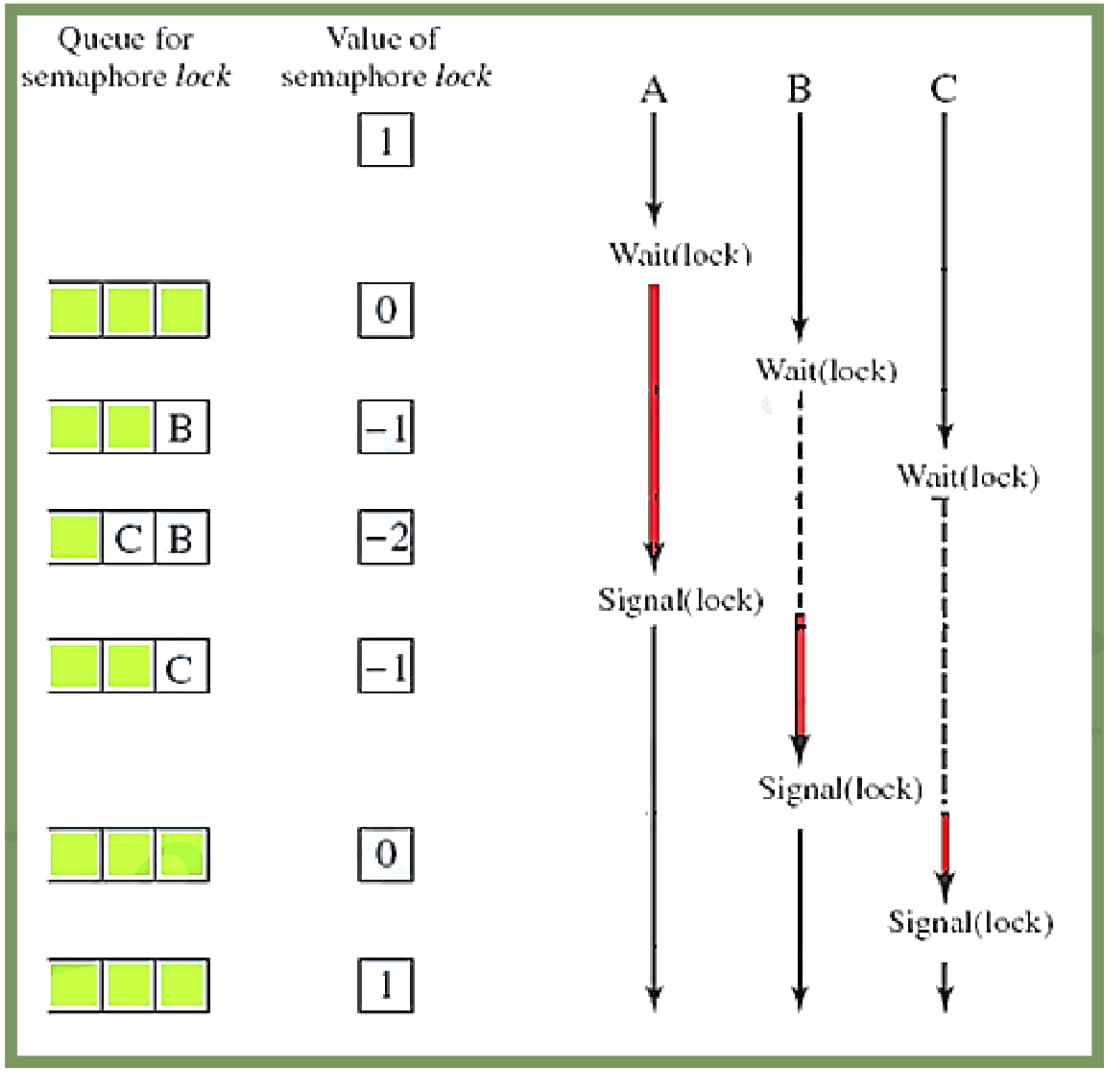
سمافور قوی و ضعیف

در سمافورها از صفی برای نگهداری فرایندهای بلوکه شده استفاده می شود.

اگر خروج از این صف به ترتیب ورود باشد (FIFO)، به آن سمافور قوی می گویند و اگر این چنین نباشد، به آن سمافور_ضعیف می گویند.

در سمافور ضعیف، امکان گرسنگی وجود دارد.

مثال



خط چین : بلوکه شدن روی سمافور **lock**

خط قرمز: ناحیه بحرانی

خطوط قرمز نمی توانند موازی باشند.

انواع سمافور

۱- عمومی

در سمافور عمومی که آن را بررسی کردیم، شمارنده می تواند مثبت، صفر و یا منفی باشد.

۲- دودویی

در سمافور دودویی، فقط مقادیر ۰ و ۱ را دریافت می کند.

قدرت سمافور دودویی با سمافور عمومی معادل است.

توابع wait و signal برای سمافور باینری

```
void wait(semaphore s)
{
  if (s.count =1)
     s.count = 0;
  else{
     place this process in s.queue;
     block this process;
  }
}
```

```
void signal(semaphore s)
{
  if (s.queue is empty )
     s.count =1;
  else{
     remove a process from s.queue;
     place this process in ready queue;
  }
}
```

همگام سازی با استفاده از سمافورها

سمافور دارای توانایی زیادی در حل مسائل همگام سازی است.

مثال

می خواهیم ابتدا دستور S1 و سپس S2 اجرا شود. از سمافوری به نام S با مقدار اولیه صفر استفاده می کنیم.

P1	P2
S1;	S2;

P1	P2
S1; signal(S);	wait(S); S2;

تا زمانی که S1 اجرا نشود، اجرای S2 ممکن نیست.

مثال

کدام یک از ترتیب های اجرا ممکن است؟

(مقدار اولیه سمافورهای S و Q برابر صفر است.)

P1	P2	P3

.
.
signal(S);
	wait(Q);
	.
	.
		wait(S);
		.
		signal(Q);

P1 → P3 → P2

مثال

خروجی را مشخص کنید. (مقدار اولیه دو سمافور S و Q برابر صفر است.)

P0	P1

signal(Q)
wait(S);
cout <<"C";
cout <<"D";
	wait(Q);
	cout << "A";
	signal(S);
	cout << "B";
	cout << "E";

P0	P1
(1) signal(Q); . . . . (6) wait(S); (7) cout << "C"; (8) cout << "D"; .	. (2) wait(Q); (3) cout << "A"; (4) signal(S); (5) cout << "B"; . . . (9) cout << "E";

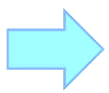
چاپ : **ABCDE**

مثال

خروجی را مشخص کنید؟ (مقدار اولیه دو سمافور S و Q برابر صفر است)

P1	P2

cout<<"1";
signal(S);
wait(Q);
cout<<"2";
signal(S);
	wait(S);
	cout<<"3";
	signal(Q);
	wait(S);
	cout<<"4";



P1	P2
(1) cout<<"1"; (2) signal(S); . . . (6) wait(Q); (7) cout<<"2"; (8) signal(S); . .	. . (3) wait(S); (4) cout<<"3"; (5) signal(Q); . . . (9) wait(S); (10) cout<<"4";

چاپ : **1324**

مثال

مقدار a و b چند خواهد شد؟ (مقدار اولیه سمافور s برابر یک و مقدار اولیه متغیرهای a,b نیز برابر یک می باشند.)

P1	P2
a=a+2;	**a=3;**
b=b+1;	**wait(s);**
signal(s);	**b=a+b;**
b=b+1;	**wait(s);**
	a=a+b;

P1	P2
.	a=3;
.	wait(s);
.	b=a+b;
a=a+2;	.
b=b+1;	.
signal(s);	.
.	wait(s);
.	a=a+b;
b=b+1;	.

a = 3 , b = 4 , s = 0

a = 5 , b = 5 , s = 1

a = 10 , b = 5 , s = 0

a = 10 , b = 6

مسئله تولیدکننده و مصرف کننده

یک تولیدکننده یا بیشتر، نوعی داده را تولید و آنها را در بافری به اندازه n قرار می دهند.

یک مصرف کننده، این اقلام را یکی یکی از بافر برمی دارد.

در هر زمان مصرف کننده یا تولید کننده می تواند به بافر دسترسی داشته باشد.

سمافور های مورد نیاز

۱- سمافور mutex :

سمافوری برای رعایت شرط انحصار متقابل است، تا تولید کننده و مصرف کننده به طور همزمان به بافر دسترسی نداشته باشند. (با مقدار اولیه ۱)

۲- سمافور full :

سمافوری برای شمارش تعداد خانه های پر بافر (با مقدار اولیه ۰)

۳- سمافور empty :

سمافوری برای شمارش تعداد خانه های خالی بافر (با مقدار اولیه n)

```
void  producer( ){
  int   item;
  while(TRUE) {
      item= produce( );
      wait(empty);
      wait(mutex);
      insert(item);
      signal(mutex);
      signal(full);
    }
}
```

```
void  consumer( ){
    int  item;
    while(TRUE) {
      wait(full);
      wait(mutex);
      item=remove( );
      signal(mutex);
      signal(empty);
      consume( );
    }
}
```

مسئله تولید کننده- مصرف کننده ، با بافر نامحدود

نیازی به سمافور **empty** نیست. پس **wait(empty)**و **signal(empty)** حذف می شود.

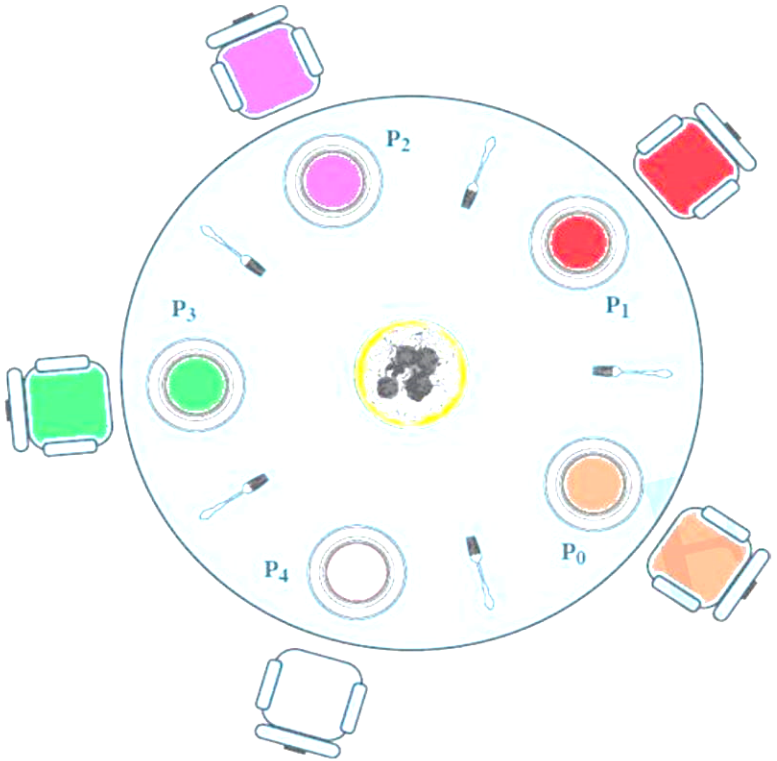
```
void  producer ( ){
  int   item;
  while(TRUE) {
        item= produce( );
        wait(mutex);
        insert(item);
        signal(mutex);
        signal(full);
     }
}
```

```
void  consumer( ) {
     int  item;
     while(TRUE) {
          wait(full);
          wait(mutex);
          item=remove( );
          signal(mutex);
          consume( );
       }
}
```

مسئله غذا خوردن فیلسوف ها

پنج فیلسوف داریم. زندگی هر فیلسوف از دو دوره متناوب خوردن و فکر کردن تشکیل شده است.

هر فیلسوف یک بشقاب ماکارونی دارد. بین هر جفت از بشقاب ها، یک چنگال قرار دارد.



هر فیلسوف برای خوردن از دو چنگال طرفین بشقاب استفاده می‌کند.

زمانی که هر فیلسوف گرسنه می شود، سعی می کند دو چنگال سمت چپ و راست خود را بردارد. اگر موفق شد، برای مدتی غذا می خورد و سپس چنگال ها را زمین می گذارد و به فکر ادامه می دهد.

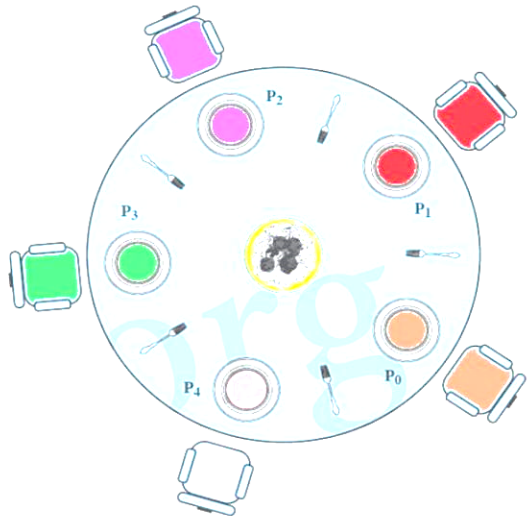
مسأله تغذیه فیلسوفان علاوه بر **انحصار متقابل** (که در یک زمان دو فیلسوف نمی توانند از یک چنگال استفاده کنند)، باید جوابگوی **بن بست و گرسنگی** نیز باشد.

راه حل

```
semaphore  room=4;
semaphore  fork[5]={1};

void   philosopher (int  i){
    while(TRUE){
        think( );
        wait( room );
        wait( fork[i] );
        wait( fork[(i+1) % 5] );
        eat( );
        signal ( fork[(i+1) % 5] );
        signal ( fork[i] );
        signal ( room );
    }
}
```

```
void  main( ){
   parbegin (p(0),p(1),p(2),p(3),p(4)));
}
```



هرفیلسوف ابتدا چنگال چپ و سپس چنگال راست را بر می دارد.

بعد از تغذیه یک فیلسوف، دو چنگالی که استفاده می کرد را روی میز گذاشته و دیگران می توانند استفاده کنند.

سمافور room با مقدار اولیه ۴ ، برای این است که اجازه ورود به بیش از چهار نفر داده نشود.

اگر حداکثر چهار فیلسوف نشسته باشند، حداقل یک نفر به دو چنگال دسترسی خواهد داشت.

اگر از room استفاده نمی شد و اجازه ورود همزمان به هر پنج نفرداده می شد، همه آنها چنگالهای چپ خود را برداشته و دیگر چنگال اضافی نمی ماند که کسی بتواند چنگال راست خود را بردارد.

بنابراین بن بست رخ می داد.

راه حل کتاب تننباوم

```
#define   LEFT     (i-1) % 5
#define   RIGHT    (i+1) % 5
typedef  int  semaphore;
semaphore   mutex=1;
semaphore   s[5];
int             state[5];
```

```
void  philosopher (int i){
  while(TRUE){
     think( );
     take_forks(i);
     eat( );
     put_fork(i);
  }
```

آرایه **state** : وضعیت جاری فیلسوف(فکر کردن(**0**) ،گرسنگی(**1**) و خوردن(**2**))

یک فیلسوف در صورتی که هیچ یک از فیلسوفان چپ و راستش، در حال خوردن نباشند، می تواند غذا بخورد.

این راه حل بن بست ندارد.

```
void take_forks(int i){
  wait(mutex);
  state[i]= 1;
  test(i);
  signal(mutex);
  wait(s[i]);
}
```

```
void put_forks(int i){
  wait(mutex);
  state[i]=0;
  test( LEFT );
  test( RIGHT );
  signal(mutex);
}
```

```
void test(int i){
  if (state[i]==1 && state[LEFT]!=2 && state[RIGHT]!= 2 ) { state[i]=2; }
}
```

فکر کردن(0) ،گرسنگی(1) و خوردن(2))

مسئله خوانندگان و نویسندگان

در این مسئله، ناحیه داده ای مثل فایل وجود دارد که بین تعدادی از فرایندها مشترک است.

فرایند های خواننده می خواهند از این ناحیه بخوانند و فرایندهای نویسنده می خواهند در آن بنویسند.

شرایط این مسئله:

۱- هر تعداد از خوانندگان می توانند به صورت همزمان از فایل بخوانند.

۲- در هر زمان تنها یک فرایند ممکن است در این فایل بنویسد.

۳- هنگامی که نویسنده ای در حال نوشتن است، هیچ خواننده ای نمی تواند فایل را بخواند.

برای مثال یک سیستم رزرواسیون هواپیمایی را در نظر بگیرید که تعداد زیادی فرایند در آن برای نوشتن و خواندن با یکدیگر رقابت می کنند.

چند فرایند می توانند به طور همزمان پایگاه داده را بخوانند ولی اگر یک فرایند در حال به روز رسانی پایگاه داده باشد، فرایندهای دیگر حتی خوانندگان، نباید به پایگاه داده دسترسی داشته باشند.

خوانندگان اولویت دارند

تا زمانی که خواننده ای وجود دارد، به خواننده اجازه ورود می دهیم.

```
typedef int semaphore;
semaphore mutex=1;
semaphore w=1;
int   rc=0;

void  writer( ){
  while(TRUE) {
      wait(w);
      writing( );
      signal(w);
    }
}
```

```
void  reader( ){
   while ( TRUE )  {
          (mutex);
      rc = rc+1;
      if (rc == 1)  wait(w);
           (mutex);
      reading( );
      wait(mutex);
      rc = rc-1;
      if (rc == 0)  signal(w);
      signal(mutex);
    }
}
```

سمافور w : اعمال انحصار متقابل

سمافور mutex : اطمینان از تغییر متناسب rc

متغیر سراسری rc : شمارش تعداد خوانندگان

توسط if اول ، به اولین خواننده اجازه ورود داده می شود. (در صورتی که نویسنده ای فعال نباشد.)

توسط if پایانی، بررسی می کنیم که اگر خواننده فعال دیگری وجود نداشته باشد، در صورت اینکه نویسنده بلوکه شده ای داشته باشیم، به آن اجازه داده شود.

در مسئله خوانندگان – نویسندگان در حالتی که خوانندگان اولویت دارند،

انتظار محدود رعایت نمی شود، چون امکان گرسنگی نویسندگان وجود دارد.