

5- راه حل پیترسون :

```
int turn;
int flag[2]={0};
Enter(int pid)
{
    int other;
    turn=pid;
    flag[pid]=1;
    other=1-pid;
    while((turn==pid)&&(flag[other]== 1));
    .....
Exit(pid)
{
    flag[pid]=0;
}
```

توضیح: این الگوریتم از دو تابع `Enter()` و `Exit()` تشکیل یافته است. قبل از وارد شدن به نامیه بهرانی هر پردازش باید تابع `Enter()` را صدا زده و شماره پردازش خود را به عنوان آرگومان به آن بفرستد. این تابع باعث می شود که پردازش تا زمانی که بتواند بدون فطر وارد نامیه بهرانی شود، منتظر باقی بماند. پس از انجام کارهای نامیه بهرانی (دستکاری متغیرهای مشترک)، پردازش تابع `Exit()` را صدا می زند که نشان دهد کارش در نامیه بهرانی تمام شده و سایر پردازش ها در صورت نیاز می توانند وارد نامیه بهرانی بشوند. پردازش ها از دو متغیر مشترک `turn` و `flag` استفاده می کنند.

دو پردازش همکار P_0 و P_1 توابع فوق را به صورت زیر استفاده می کنند.

پردازش P_0
`Enter(0);`
`Critical_Section();`
`Exit(0);`

پردازش P_1
`Enter(1);`
`Critical_Section();`
`Exit(1);`

الف: این الگوریتم شرط انحصار متقابل را بر آورده می کند، زیرا فرض کنید هر دو پردازش باهم بخواهند وارد نامیه بهرانی بشوند، چون قبل از ورود به نامیه بهرانی حلقه `while()` را داریم، پردازش ای در حلقه می ماند که آخرین بار مقدار `turn` را تغییر داده باشد، و پردازش دیگر وارد حلقه می شود.

ب: شرط پیشرفت نیز برقرار است، زیرا هر پردازش به هنگام خروج از نامیه بهرانی، `flag` متناظر خود را صفر کرده و همین عمل اجازه ورود دیگر پردازش ها را با توجه به شرایط پردازش می دهد. به عبارتی پردازش ای که در قسمت R.C خود باشد در تصمیم گیری ورود دیگر پردازش ها به نامیه بهرانی دخالت ندارد.

از پنج راه حل گفته شده، دو راه حل آخر، که شرایط سه گانه را بر آورده می کنند، مشکل `Busy waiting` (انتظار مشغول) دارند، یعنی اینکه 1- یعنی اگر پروسسی بخواهد به نامیه بهرانی اش وارد شود ولی اجازه نداشته باشد، در یک حلقه بی کار می افتد، تا هنگامی که اجازه او صادر گردد. این روش باعث اتلاف وقت CPU می گردد.

2- فرض کنید دو پردازش یکی با الویت بالا و دیگری با الویت پایین در سیستم داشته باشیم، اگر پردازش الویت پایین در نامیه بهرانی باشد و پردازش الویت بالا از راه برسد و بخواهد وارد نامیه بهرانی بشود، داخل حلقه گیر می افتد. و چون الویت این پردازش بالاتر از الویت پردازش ای می باشد که در نامیه بهرانی است، بنابراین پردازش موجود در نامیه بهرانی هیچ فرصت اجرا (بدست آوردن CPU) را ندارد، پس نمی تواند از نامیه بهرانی خارج شود و پردازش الویت بالا مدام در تست شرایط، مشغول می باشد (تا بی نهایت در حلقه دور می زند). پس هیچ کاری پیش نمی رود.

□ مشکل دیگر روش های گفته شده این است که قابل تعمیم به مسائل پیچیده نیستند.

6- سمافورها (Semaphores-راهنما ها)

سمافور x یک متغیر صحیح می باشد که جدای از مقدار دهی اولیه، فقط از طریق دو عمل اتمیک استاندارد زیر قابل دسترسی است

$$\text{wait}() \rightarrow p() - 1$$

$$\text{signal}() \rightarrow v() - 2$$

تعریف ساده p و v به صورت زیر است.

□ سمافور ها را می توان برای مساله بخش بهرانی n پردازش استفاده کرد.

فرض کنید دو پردازش p_1 و p_2 به شکل زیر اجرا شوند (مقدار اولیه سمافور 1 می باشد)

Semaphore(x)	
P(x)	v(x)
While(x<=0); x--	x++

p_1	p_2
$p(x);$	$p(x);$
$c++;$	$c--;$
$v(x);$	$v(x);$

فرضا اگر ابتدا پردازش p_1 بخواهد اجرا شود با تابع p مقدار سمافور را صفر می کند و بعد وارد ناحیه بهرانی می شود (بجهت دستکاری متغیر مشترک) و در انتها با دستور v مقدار سمافور را یک می کند، که قبل از یک شدن مقدار سمافور، اگر پردازش p_2 می خواست وارد ناحیه بهرانی شود در حلقه $\text{while}(x \leq 0)$ گیر می افتاد. (تا لحظه ای که پردازش p_1 دستور v را اجرا کند و دوباره مقدار سمافور یک شود)

□ اگر تعریف p و v به صورت داده شده باشد و سمافور x فقط در بردارنده یک مقدار باشد مشکل *Busy waiting* این جا نیز برقرار است. برای رفع این مشکل تعریف سمافور p و v را کاملتر می کنیم. در این حالت زمانی که پروسس، اجازه ورود به ناحیه بهرانی اش را ندارد بلوکه یا مسدود می شود (به جای آنکه در یک حلقه *while* پرخ بزند)، بدین ترتیب آن پروسس به حالت تعلیق می رود تا هنگامی که پروسس دیگری آن را بیدار (*wakeup*) کند □ سافتار واقعی سمافور، p و v به صورت زیر است.

Struct semaphore
{
List of process l;
Int value;
}

Semaphore x;	
P(x)	v(x)
x.value - -;	x.value++;
if(x.value<0) begin	if(x.value<=0)begin
add process to x.l;	remove a process p from x.l;
block the process;	wakeup(p);
end;	end;

توضیح: وقتی پردازشی عمل p را اجرا کرده و سمافور را غیر مثبت می یابد، باید صبر کند. اما به جای حلقه $while$ ، پردازش در صف انتظار مربوط به آن سمافور قرار داده می شود و حالت پردازش به $block$ تغییر می یابد. سپس کنترل به زمانبند CPU منتقل می گردد تا پردازش دیگری را برای اجرا، انتخاب کند. فرایندی که $block$ شده بر اثر اجرای عمل p توسط فرایند دیگر از سر گرفته می شود. این فرایند توسط یک عمل $wakeup$ بیدار شده و از حالت انتظار به حالت آماده می رود و در صف $Ready$ قرار داده می شود.

□ در ساختار جدید مقدار سمافور می تواند منفی باشد، که قدر مطلق این مقدار برابر تعداد پردازه هائی است که منتظر این سمافور هستند

□ اگر در هنگام اجرای دستور $wakeup(x)$ پردازش x در حالت بلوکه نباشد، این دستور هیچ عملی خاصی را انجام نمی دهد و اثری ندارد.

مسئله تولید کننده-مصرف کننده: دو پردازه تولید کننده و مصرف کننده وجود دارد که به طور همروند اجرا می شوند، تولید کننده یک سری اقلام اطلاعاتی را تولید می کند و در یک بافر که اندازه آن محدود است قرار می دهد. پردازه مصرف کننده این اقلام اطلاعاتی را از بافر برداشته و اجرا می کند.

نکته: دستورات $count++$ و $count--$ به صورت زیر عمل می کنند.

```
Int Buffer[max];
Int count=0;
Producer(){
While(1){
Produce an item;
Buffer[count]=item;
Count ++;
}
Concumer(){
Count--;
Item=Buffer[count];
Consum an item;
}
```

<u>Count--</u>	<u>Count++</u>
Mov Ax,Count;	Mov Ax,Count;
Sub Ax,1;	Add Ax,1;
Mov Count,Ax;	Mov Count,Ax;

□ محدودیت هائی که در این مسئله وجود دارند عبارتند از:

□ اگر بافر پر باشد و تولید کننده بخواهد یک قلم اطلاعاتی را در بافر قرار دهد، می بایست صبر کند تا بافر خالی شود.

□ اگر بافر خالی باشد و مصرف کننده بخواهد یک قلم اطلاعاتی را از بافر بردارد، می بایست صبر کند تا اطلاعاتی در بافر قرار داده شود.

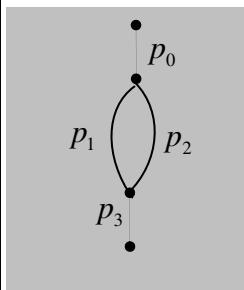
□ مشکل دیگر مربوط به متغییر مشترک $count$ می باشد، به طور مثال فرض کنید $count = 3$ باشد، و پردازه مصرف کننده بخواهد یک

قلم اطلاعاتی را از بافر بردارد، در این صورت اگر این پردازه به دستور $Mov Ax,Count;$ برسد سه را در Ax قرار می دهد، حال قبل از اینکه مقدار تغییر یافته $count$ یعنی دو در Ax نوشته شود، $Context switching$ رخ می دهد و پردازه تولید کننده اجرا می شود. این

پردازه مقدار $count$ را سه می بیند، و بنا براین سه را در Ax قرار می دهد. اگر پردازه تولید کننده ابتدا، مقدار تغییر یافته $count$ را بنویسد و بعد از آن پردازه مصرف کننده مقدار تغییر یافته $count$ را بنویسد، مقدار $count$ ، دو خواهد شد و در حالت بر عکس مقدار نهائی چهار خواهد شد. که هر دو حالت نادرست هستند، زیرا مقدار نهائی باید سه باشد، پس این مشکلات می بایست حل شوند. با استفاده از سمافور ها می توانیم برنامه تولید کننده و مصرف کننده را به شکل زیر بنویسیم.

Semaphor mutex=1, Count=0;	Consumer(){
Semaphor empty=max;	while(1){
Semaphor full=0;	P(full);
Producer(){	P(mutex);
While(1){	Count - -;
Produce an item;	item=Buffer[count];
P(empty);	V(mutex);
P(mutex);	V(empty);
Buffer[count]= item;	consum the item;
Count ++;	}}
V(mutex);	

مثال: فرض کنید پردازش های p_0, p_1, p_2, p_3 می خواهند به صورت هم روند اجرا شوند. با استفاده از سمافور ها دستوراتی بنویسید، که هماهنگی بین پردازش ها را به صورت شکل فوق فراهم کنند



Semaphor $S_{12} = 0$ Semaphore $S_3 = 0$

Cobegin() {

$p_0(), p_1(), p_2(), p_3()$

}

$P_0()$ {

⋮

$V(S_{12})$

$V(S_{12})$

}

$P_1()$ {

$P(S_{12})$

⋮

$V(S_3)$

}

$P_2()$ {

$P(S_{12})$

⋮

$V(S_3)$

}

$P_3()$ {

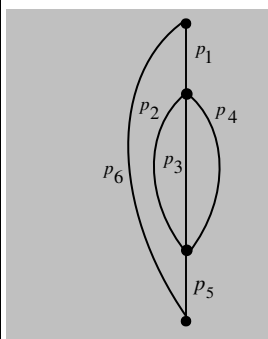
$P(S_3)$

$P(S_3)$

⋮

}

مثال: فرض کنید پردازش های $p_1, p_2, p_3, p_4, p_5, p_6$ می خواهند به صورت هم روند اجرا شوند. با استفاده از سمافور ها دستوراتی بنویسید، که هماهنگی بین پردازش ها را به صورت شکل فوق فراهم کنند



semaphor $s_1 = 0, s_{234} = 0$;

Cobegin() {

$p_1(), p_2(), p_3(), p_4(), p_5(), p_6()$

}

$p_1()$ {

⋮

$V(s_1)$;

$V(s_1)$;

$V(s_1)$;

}

$p_2()$ {

$P(s_1)$

⋮

$V(s_{234})$;

}

$p_3()$ {

$P(s_1)$

⋮

$V(s_{234})$;

}

$p_4()$ {

$P(s_1)$

⋮

$V(s_{234})$;

}

$p_5()$ {

$P(s_{234})$;

$P(s_{234})$;

$P(s_{234})$;

⋮

}

$p_6()$ {

⋮

comands

⋮

}