Subject:

Year. Month. Date. ( )

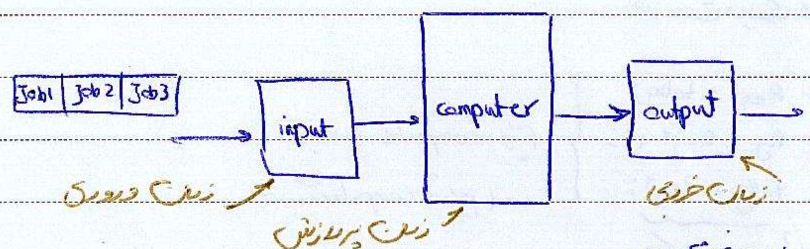
کارهای که یک دسته ای به آن می دهند

تمرین) راجع به یک مسئله خیلی تیپیکی (در درس گفته نشده)

جدول زیر زمان های لازم برای ورود، پردازش و خروج سه کار را در یک سیستم batch نشان می دهد. حداقل کل زمان مصرفی برای اجرای هر سه کار چقدر است؟ (ترتیب ورود تعیین کننده ترتیب پردازش و خروج است)

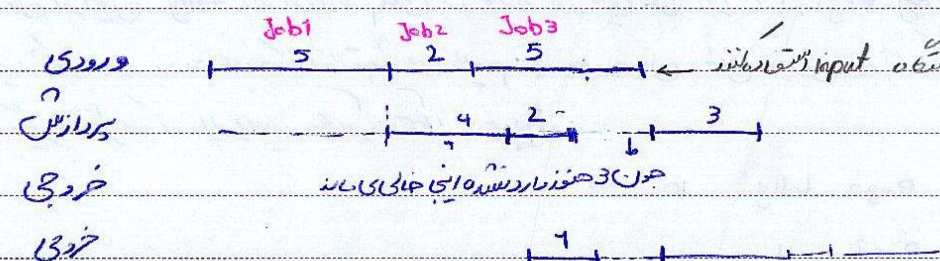
	زمان ورودی (Input time)	زمان پردازش (processing time)	زمان خروجی (output time)
Job1	5	4	1
Job2	2	2	3
Job3	5	3	2

سیستم batch



این سیستم ها تعاملی نیستند چون ورودی را می گیرد و از آن در خروجی را تحویل می دهند

می توانیم با این سیستم یک سری Job برای انجام بدهیم هر کدام از این ها زمان های لازم دارند که خوانده می شوند ← زمان ورودی



پردازش باید وقتی شروع شود که ورودی به طور کامل خوانده شود یعنی پردازش Job1 بعد از 5 واحد زمانی شروع می شود ولی بعد از آن چون Job2 به طور کامل وارد شده آن را پردازش می کنیم ولی چون بعد از آن Job3 هنوز کامل وارد نشده نمی توانیم آن را شروع کنیم پس سیستم تا مدتی بیکار می ماند و وقتی Job3 کاملا آمد شروع به پردازش آن می کند.

Subject:
Year. Month. Date. ( )

تمرین) دو تا فرایند P1 و P2 به طور همروند اجرا می‌شوند

```
P1   (loop بی‌پایان)            P2

while (TRUE){                   while (TRUE);
    print "A";                  print "C";
    print "B";                  print "D";
}                               }
```

بدون همزمانی ← A CDCD

اگر خروجی را درست ثبت بود چون هر ترتیبی امکان داشت

(AB)* (CD)* — می‌تواند جواب باشد چون همزمانی در پایان هر loop تمام می‌شود

A(CD)*B — یک A را print می‌کنیم همزمانی ما تمام می‌شود روی P2 چندین بار تکرار می‌شود

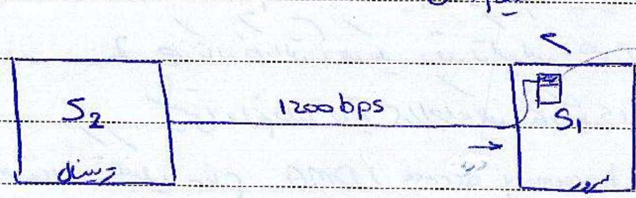
BCAD ← حتمی از خروجی — از یک جایی شروع می‌کنیم B را print می‌کند همزمانی تمام می‌شود بعد print C می‌کند و را از دست می‌دهد

C*A*B*C* — این امکان ندارد باشد

مثال دیگر) در مورد کنترل ورودی‌ها و خروجی‌ها با وقفه

دو تا سیستم به هم وصل هستند — سیستم اصلی

به محض وقفه در مکان خاص ذخیره — کانتیم



S2 برای یک سری ترمینال عمل می‌کند و پردازش‌ها را S1 انجام می‌دهد

سرعت ارتباط بین اینها خیلی کم نوشته ← 1200 bitps ، واحد ارتباطی این دو سیستم کاراکتر است،
هر کاراکتر 10bit است

دریافت هر کاراکتر با یک وقفه مشخص می‌شود و با این سرعت 120 کاراکتر در ثانیه می‌توانیم ارسال کنیم

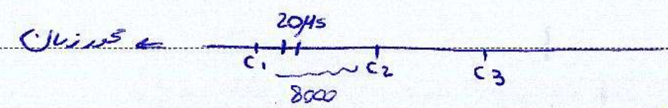
PAPCO

وقفه حاصل می‌شود ← باید داخل روتین وقفه برویم و آن را اجرا کنیم

روال وقفه خواندن پورت و ذخیره در حافظه ← 20 µs

سوال اول: آیا این سیستم مشکلی پیدا می‌کند؟

فاصله ورود دو کاراکتر $8000\,\mu s = 8ms = \frac{1}{120} s$

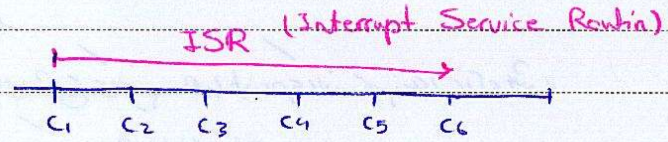


در 8000 می‌خواهیم، 20 تا برای این کار مصرف کنیم ← هیچ مشکلی پیش نمی‌آید.

حالت دوم:

در حالت دوم می‌آید شبکه را سریع‌تر می‌کند به طوری که به ازای هر 4µs یک کاراکتر وارد می‌شود

1 char per 4µs



کاراکتر اول یک وقفه تولید می‌کند، پردازش هیچ فرقی نمی‌کند، c1 بردن وقفه را راه می‌اندازد

برای این رو به روتین وقفه برای c1 رسیدگی کند 4 تا char را از دست داده و هم چنین تا ISR تمام می‌شود دوباره یک char جدید وارد می‌شود و همیشه ISR در حال اجراست.

راه حل‌ها: 1- کندش کنیم و اجازه ندهیم هر 4µs بیاید (در صورتی که امکان پذیر باشد)

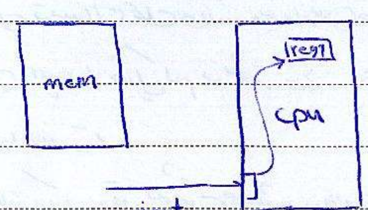
2. چند تا کاراکتر را هر یک وقفه تولید کنیم، همه را با هم به خودشون بسپریم ولی این هم چون سیستم دائم دارد روتین را اجرا می‌کند کارگیری نمی‌تواند بکند

راه حل اصلی: استفاده از تکنیکی به نام (Direct memory access) DMA ← دسترسی مستقیم به حافظه

← دسترسی غیر مستقیم به حافظه داریم

دسترسی port به حافظه غیر مستقیم است

و توسط CPU است.



input 01, reg1

ld reg1, M1

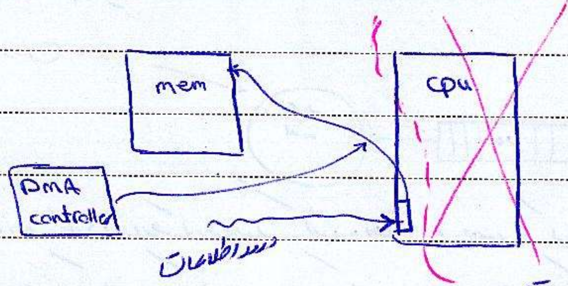
inc address ← چون می‌خواهیم روی همان خانه ننویسیم که

به ازای هر چیزی که وارد می‌شود باید چندین Instruction اجرا کنیم

حال DMA را می‌گوییم که واسطه را بردارد و ارتباط را مستقیم برقرار کند

چون لازم نیست Instruction اجرا بشود چون هر چیز خودش می‌رود سر جایش پس خیلی سریع اجرا می‌شود ولی

کمک ابزار خاصی برای اجرای این کار می‌خواهد



کنترل DMA controller در دست OS است

هر وقت 1000 تا char آمد یک Interrupt می‌دهد به این هم نمی‌شوند بلکه در حافظه ذخیره می‌شوند بعد در این جا

حافظه نقش buffer عمل می‌کند

مثال: یک دفعه برای 1000 ورودی

جلسه سیزدهم 7/8

همگامی فرایندها: process synchronizing

چند تا فرایند هم روند دارای منابع مشترک پس باید برای همگام بشوند که منجر به کار یکپارچه بشوند یعنی استفاده

مشترک از فرایندها به صورت هماهنگ بشد باشد.

این همگامی ← توسط خود فرایندها

↙ توسط OS

monitor

مثالی از همگامی فرایندها

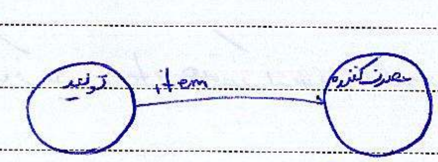
مثال تولیدکننده (producer) مصرف‌کننده (consumer)

در فرایندی یک چیزی تولید می‌کند که فرایند

دیگر آن را مصرف می‌کند برای اینکه با هم

کار کنند باید با هم هماهنگ باشند

مصرف‌کننده اگر بخواهد چیزی را مصرف کند باید قبلش تولید شده باشد

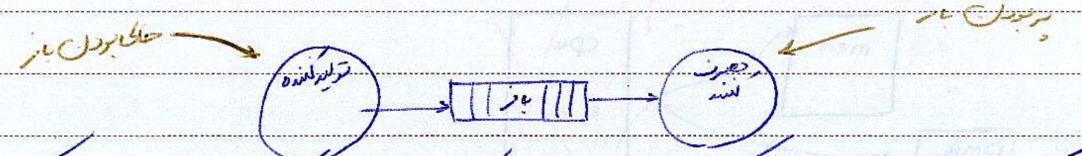


پس تولید باید قبل از مصرف باشد. اگر چیزی تولید نشده باشد نمی‌تواند مصرف کننده، مزاحم مصرف کننده

اگر تولید کننده به طور مرتب تولید کند و مصرف کننده به طور مرتب مصرف کند

این مثال محدود کننده است. مصرف ⟵ تولید
تولید ⟵

و اشکال این درهم خیلی زیاد است



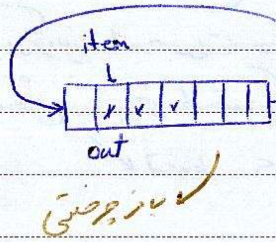
در این حالت تولید کننده کاری ندارد که مصرف کننده item را مصرف کرده یا نه، کار خود را انجام می‌دهد و تولید می‌کند و هم مصرف کننده. مصرف کننده هم بلند و مصرف کننده هم هر موقع که بخواهد می‌تواند item ها را مصرف کند اگر خالی نباشد

همچنین: برای تولید کننده: بافر پر نباشد

" مصرف کننده: آزمون خالی بودن بافر

← در این حالت خیلی زمین رفته بلکه مستقل تر شده و نسبت به حالت قبل که پله پله به هم وابسته بودند کمتر شده.

پیاده سازی بافر



یک آرایه است و pointer داریم به نشان می‌دهد که بافر را می‌توانیم input کنیم و از طرف دیگر یک pointer برای مصرف کننده است که نشان می‌دهد تا کجا خوانده شده و عقب تر از pointer اول است. این دو پشت سر هم هستند و می‌چرخند و خانه‌هایی از بافر که خوانده شده آزاد می‌شود. این بافر به صورت بافر حلقوی می‌تواند ادامه دهد

هر کدام از خانه‌های بافر یک item هستند که ساختاری دارند و هر چی می‌توانند باشند

```
typedef item ____ ;
item buffer[n];      → آرایه‌ای از item تعریف شد
int in, out;         → pointer هستند
```

وقتی بافر خالی است in=out است و شرط خالی بودن: in=out

به نظر پر بودن هم همین است، اگر همین باشد تست پر یا خالی بودن چه فرقی دارند؟

قرارداد می‌کنیم ← پر می‌کنیم تا جایی که به قبل out برسیم و یک خانه مانده به out می‌گوییم پر است و برای این کار یک خانه را قربانی می‌کنیم پس داریم: شرط پر بودن: in+1=out

چون حالت چرخشی دارد باید محاسبه بشود ← (in+1) mod n = out

اگر تعداد بافرها زیاد باشد از دست دادن یک مکان (خانه) خیلی مهم نیست

PRODUCER:

```
While (True) {                            حلقه‌ای است که دائماً تولید می‌کند
  ---
  produce an item in nextp                item را تولید و در متغیر nextp قرار بده
  --
  while ((in+1) mod n = out));  →         تا زمانی که پر هست در این حلقه بمان تا یک مکان خالی شود
  buffer[in] = nextp;
  in = (in+1) % n;              →         یک واحد pointer را اضافه می‌کند
}
```

CONSUMER

```
while (True) {
    while (in == Out);          →         تا زمانی که خالی هست در همین حلقه بچرخ
    nextc = buffer[out];
    out = (out+1) % n;
    --
    consume the item in nextc;            item موجود در nextc را مصرف کن
}
```

Subject:
Year. Month. Date. ( )

update کردن این ها باهم صورت نمی‌گیرد

متغیرهایی را می‌توان توسط OS تعریف کرد که بین فرایندها مشترک باشد در این حالت مشکل پیش

می‌آید زمانی مشکل پیش می‌آید که دو فرایند باهم بخواهند متغیر را تغییر دهند

← می‌خواهیم راه حلی بدهیم که مکان از دست رفته نداشته باشیم؟

یک متغیر در این جا تعریف می‌کنیم که تعداد خانه‌های پر بافر را به ما می‌گوید

اگر counter=0 ← خالی

اگر counter=n ← پر

وقتی های کد بین به صورت زیر تغییر می‌کند.

```
producer:  while (counter == n);
           buffer [in] = nextp;
   I       counter = counter + 1

consumer:  while (counter == 0)
           nextc = buffer [out];
   II      counter = counter - 1;
```

این راه حل کار نمی‌کند: چون counter را هر دو مشترکاً می‌کنند و کافی است در ازایش یا کاهش counter اشتباه بشود ← پر و خالی بودن را عوض می‌کنیم

ممکن است چیزی را بنویسیم که یک خانه خوانده شده را دوباره می‌خوانیم یا یک چیزی را که خواندیم

هنوز نشده پس یک item از دست می‌دهیم

← دستورات I و II باید به زبان سطح پایین ای باشند

$$\text{counter} = \text{counter} + 1 \begin{cases} \text{reg1} = \text{counter} \\ \text{reg1} = \text{reg1} + 1 \\ \text{counter} = \text{reg1} \end{cases}$$

$$\text{counter} = \text{counter} - 1 \begin{cases} \text{reg2} = \text{counter} \\ \text{reg2} = \text{reg2} - 1 \\ \text{counter} = \text{reg2} \end{cases}$$

PAPCO

Subject:
Year. Month. Date. ( )

در producer هستیم و مقدار counter=6 است

counter = 6

reg1 = 6

reg = 7 ⟹ قبل از نوشتن کم زمانی تمام شده

⟹ الان باید counter=7 باشد

consumer counter = 6

reg2 = 6

reg2 = 5

counter = 5 ⟹ کاملا واضح است این عدد غلط است

در حالت اصلی باید به 6 می‌رسیدیم

← در روش اول هر دو فرایند فقط می‌خوانند و چون رستاره‌ای می‌کنند به مشکل برمی‌خوریم. ممکن است پر یا خالی بودن را نادرست یا دیرتر بفهمیم

← در این جا از یک متغیر مشترک integer استفاده کردیم در صورتی که هر چیزی می‌تواند باشد پس استفاده از یک منبع مشترک بدون حفاظت (در part، printer)

همه مشکل اصلی در این جا بود

فرایندها باید در استفاده مشترک از منابع با هم هماهنگ کنند در صورتی که هماهنگ باشند ولی تنها منبع مشترک اطلاعات باشد

← حفاظت شده با فرکانس کار کردن و قبل از دسترسی حتما تست می‌کنند

— یک مشکل کلی است برای همه منابع پس باید یک راه حل کلی باشد و فقط در مورد این متغیر نباشد

هر منبع مشترکی که استفاده می‌شود باید حتما یک controller باشد و مدیریت بر منابع مشترک نداریم و درخواست‌های خود را به OS می‌دهیم و OS این را handle می‌کند چون همه چیز را می‌بیند این توانایی را دارد و همه چیز را به دست OS می‌سپاریم برایش ساده است چون همه جا را می‌تواند ببیند.

اینکه OS این کار را انجام دهد و یا خودمان مزایا و معایبی دارد.

که اگر زیاد به سراغ os برویم کارها کند ولی سیستم را کند می‌کند و کارهایش را چون کلی انجام دهد برای یک کار کوچک کد زیادی را اجرا می‌کند چون باید همه حالت‌ها را در نظر بگیرد.

مسئله را به صورت کلی بیان می‌کنیم:

مسئله ناحیه بحرانی (Critical Section problem)

ناحیه بحرانی: جایی که می‌خواهیم حفاظت شده باشد در مثال قبل ناحیه بحرانی موارد زیر است

counter=counter-1 و counter = counter+1

برای حل این یک مدل برای ناحیه بحرانی تعریف می‌کنیم

مدلی برای ناحیه بحرانی:

n تا فرایند را در نظر می‌گیریم که می‌خواهیم با هم هماهنگ کار کنند

که می‌توانند مختلف باشند ولی به منبع مشترکی نیاز دارند

```
void P(int i)
{
while (True) {
    enter_critical (i);
    /* critical_section */ ;
    exit critical (i);
    /* noncritical_section */ ;
}
}
```

- enter_critical: این جا چک می‌کند آیا می‌تواند وارد ناحیه بحرانی شود یا نه
- critical_section: کدهایی که در این جا هستند که در این بخش به منبع مشترک دسترسی پیدا می‌کنند
- noncritical_section: در این جا به منبع مشترک کاری ندارد.

حفاظت منبع و داده مشترک

به بقیه اطلاع می‌دهد
جای محدودی در ناحیه بحرانی سیستم

بدترین حالت است که n تا فرایند از یک منبع مشترک دائما در حال استفاده هستند

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

معمولا که برای همه موارد کار کند چیزی در راه حل‌ها است و همه چیز را در نظر می‌گیرد در صورتی که خیلی وقت‌ها لازم نیست و انجام این راه حل هزینه اش بالا می‌رود.

Subject:
Year. Month. Date. ( )

چون این مشکل راه حل ساده‌ای ندارد برای دو فرایند آن را حل می‌کنیم سپس آن را به n تعمیم می‌دهیم

جلسه چهاردهم

http://groups.google.com/group/OS_aut_fall_87

برای این هفته (1-2-3-12-13)

← راه حل های ناحیه بحرانی را برای دو فرایند بررسی می‌کنیم

راه حل های دو فرایندی برای مسئله ناحیه بحرانی

← برای درست بودن راه حل باید سه شرط برقرار باشد

شرط های راه حل صحیح:

1. اگر فرایندی داخل ناحیه بحرانی شود دیگر فرایندی نمی‌تواند وارد این ناحیه شود ← انحصار متقابل mutual exclusion

تقاضای فرایند در ناحیه بحرانی باشد

2. انتظار محدود bounded wait ← برای وارد شدن به ناحیه بحرانی باید تضمین شود که میزان انتظار فرایند محدود است

← این دو تا شرط برای صحیح بودن راه حل کافی است اما یک شرط دیگر هم بعداً می‌گوییم

تلاش اول (first attemp 1)

```
P0 → while (True) {
        while (turn != 0);
        critical_section0();
        turn = 1;
        noncritical_section0();
     }

P1 → while (True) {
        while (turn != 1);
        critical_section1();
        turn = 0;
        noncritical_section1();
     }
```

boolean متغیر (turn)

فرایندها متغیرهایی که با منبع مشترک کار می‌کنند یکی باشد.

اگر Turn=0 باشد نوبت برای P0 است و می‌تواند وارد ناحیه بحرانی است اگر 0 نبود در این حلقه گیر می‌ماند

ولی وقتی شرط برقرار بود وارد ناحیه بحرانی خودش می‌شود وقتی کارش در ناحیه بحرانی تمام شد نوبت را به P1 می‌دهد

PAPCO

Subject:
Year. Month. Date. ( )

در این روش انحصار متقابل وجود دارد چون turn یا صفر است یا ۱ در یک لحظه فقط یک مقدار را دارد. نوبت بین P_0 و P_1 در رفت و آمد است و تعداد نوبت ها مساوی است چون هر کی نوبت را بعدش به اون یکی می دهد ← پس هم ترتیب هیچ تاثیری بر درست و غلط بودن این روش ندارد.

← شرط دوم برقرار است چون هرکسی هر قدر که بخواد نوبت منتظر می شود چون نوبت بعد از این که کار فرایندای ۲ شد به دیگری می دهد این که این یک نوبت چقدر طول می کشد ربطی به موضوع ندارد.

← پس از این راه حل خوشمان می آید چون شرط سوم که نگفتیم ندارد یعنی این راه حل خوبی نیست حاله چرا؟

فرض کنیم ناحیه بحرانی P_0 کم و برای P_1 زیاد باشد. (و غیر بحرانی)

نوبت با P_0 و P_0 وارد ناحیه بحرانی می شود و کارش را انجام داد و انجام می کند و نوبت را به P_1 می دهد و کار ناحیه غیربحرانی خود را انجام می کند

نوبت با P_0
ناحیه بحرانی P_0
نوبت با P_1
ناحیه غیربحرانی P_0
انتظار

P_1 در ناحیه غیر بحرانی

P_0 باید انتظار بکشد که کار ناحیه غیر بحرانی P_1 تمام شود و خیلی طول می کشد ولی اگر P_1 کاری با ناحیه بحرانی ندارد پس باید P_0 بتواند کار خود را با ناحیه بحرانی اش انجام دهد و اینکه نوبت ها را مساوی می دهد و اصلاً خوب نیست (چون P_1 اصلاً ناحیه بحرانی اش را نمی خواهد ولی ما صبر می کنیم تا بخواهد) اگر یکی تند تند نوبت بخواهد و دیگری یواش یواش ← سریع تر را به سرعت فرایند با سرعت کندتر می رساند ← به عبارتی آنها را به سرعت پایین می رساند

شرط سوم) پیشرفت Progress ← کسی که داوطلب ورود به ناحیه بحرانی نیست در رقابت شرکت نکند

این راه حل خوبی نیست چون شرط سوم را ندارد می خواهیم راه حلی بدهیم ← متغیری تعریف می کنیم که نشان دهیم کی داوطلب ورود به ناحیه بحرانی است.

PAPCO

Subject:
Year. Month. Date. ( )

(second attemp)

```
P0 → while (True) {
    ① flag[0] = True;
    ------------------ پایان تکه زمانی
    ④ while (flag[1]);        ← wait
    critical_section0();
    flag[0] = false;
    noncritical_section0();
}

P1 → while (True)
    ② flag[1] = True;
    ③ while (flag[0]);        ← wait
    critical_section1();
    flag[1] = false;
    noncritical_section1();
}
```

flag ← نشان می‌دهد کی داوطلب است. اگر کسی بخواهد وارد ناحیه بحرانی شود، flag خود را true می‌کند و برای وارد شدن به ناحیه بحرانی flag دیگری را چک می‌کند.

این راه‌حل از نظر انحصار متقابل صحیح است و هر دو نمی‌توانند وارد ناحیه بحرانی شوند.
شرط سوم را هم دارد ← کسی که داوطلب نیست در رقابت نخواهد بود چون flag آن true نیست.
آیا شرط دوم هم برقرار است؟

اصلاً به شرط دوم نمی‌رسد و یک مشکل دیگری دارد که کار نمی‌کند ← فرض کنیم تکه زمانی برای P_0 است و flag خود را True می‌کند ① قبل از وارد شدن به ناحیه بحرانی تکه زمانی تمام می‌شود و در بعدی وارد می‌شویم. نوبت به P_1 می‌رسد و ② flag خود را True می‌کند ولی در حلقه چون flag[0] True است ③ گیر می‌کند نمی‌تواند رد شود و صبر می‌کند تا تکه زمانی تمام شود. نوبت به P_0 می‌رسد چون flag[1] هم true است ④ P_0 در حلقه گیر می‌کند و هر دو منتظر هستند flag دیگری false شود ← شبیه حلقه بن‌بست است و راهی برای نجات ندارند و احتمال غیرصفر برای اتفاق افتادن این وجود دارد.

راه‌حل درست: این دو راه‌حل را با هم مخلوط می‌کنیم ← هم turn داریم و هم flag

تلاش سوم (third attemp)

P_0 →

```
while (True) {
    flag[0] = True;        ← می‌خواهد وارد شود   (قسمت ورود)
    turn = 1;              ← برای نوبت دهی
    while (flag[1] && turn == 1)
    critical_section0();
    flag[0] = false;       (ناحیه خروج)
    noncritical_section0();
}
```

P_1 →

```
while (True)
    flag[1] = true;        (قسمت ورود)
    turn = 0;
    while (flag[0] && turn == 0);
    critical_section1();
    flag[1] = false;       (ناحیه خروج)
    noncritical_section1();
}
```

flag خودرا True کنم ← می‌خواهم وارد ناحیه بحرانی شوم

اگر کسی Turn را دارد اما flag ندارد ← نمی‌تواند جلوی کسی را برای ورود به ناحیه بحرانی بگیرد.

احتمال دارد طرف مقابل را دچار گرسنگی نکند.

← اگر هرکس turn را قبل از ورود به ناحیه بحرانی به خودش بدهد (الگوریتم حریصانه) ممکن است طرف مقابل را به حالتی ببیند که دیگر نوبت به دیگری نرسد. مثلاً P_0 چندین بار turn را بگیرد و به P_1 ندهد این چندین بار معلوم نیست چقدر است ← امکان اتفاق افتادن این غیرصفر است.

با اینکه نوبت را به دیگری بدهیم اگر درخواست نباشد هیچ اتفاقی نخواهد افتاد و یک نوبت بیشتر معطل نمی‌شود و نه بیشتر چون کسی که الان Turn را دارد اول Turn را به من می‌دهد.

محدودیت [illegible] تایید است و تنها تاثیری که دارد این است که حداکثر یک نوبت می‌تواند من را عقب بیندازد

تعمیم به n فرایند:

همین راه‌حل را می‌توان به n فرایند تعمیم داد البته کار خیلی سختی است. اینکه این را بدانیم خوب است ولی اگر خودمان بخواهیم به دست بیاوریم سخت است

در عمل دو حالت داشتیم ← می‌خواهند وارد ناحیه بحرانی شوند

← نمی‌خواهند " " "

Subject:

Year. Month. Date. ( )

ابتدا این سه حالت را تعریف می‌کنیم ← idle نمی‌خواهد وارد شود

want_in ← تمایل دارد وارد ناحیه بحرانی شود

in_cs ← وارد ناحیه بحرانی شده

```
enum state {idle, want_in, in_cs};
state flag[n];
int turn;

    enter_critical_section:
    do { flag[i] = want_in;
         j = turn;
         while (j != i) {
              if (flag[j] != idle) j = turn;
              else j = (j+1) % n;
         }

         flag[i] = in_cs;
         j = 0;
         while (j<n && (j == i || flag[j] != in_cs)) j++;
    } while (j>=n && (turn==i || flag[turn]==idle)); break
         [while(True)]

       turn = i;
```

ناحیه مهم

می‌بیند تا چه حالی باشد

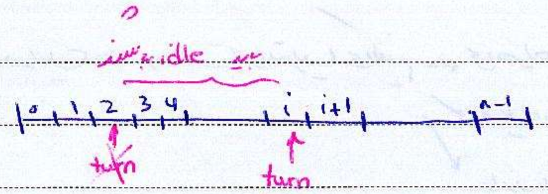
شرط انحصار متقابل

اگر فرایندی را پیدا می‌کند که in_cs نبیند اگر دید دوباره از اول، اگر مشغول نبود وارد ناحیه بحرانی می‌شود.

وقتی می‌خواهد وارد ناحیه بحرانی شود ← turn را به خودش می‌دهد بعد از اینکه همه چیز را چک کرده باشد که آخرین کاری است که انجام می‌شود یعنی بلافاصله بعدش وارد ناحیه بحرانی می‌شود

PAPCO

ناحیه خروج چی کار می‌کنند؟



چک می‌کند دونه دونه را از turn تا خودش که همه باید idle باشند اگر اینطور باشد یعنی که حق با من است اگر idle نباشند در این ناحیه گرفتار می‌شود و انقدر می‌ماند تا همه را idle ببیند می‌گه حق با خودش است اگر این حالت اتفاق افتاد flag خودش را به in_cs تغییر می‌دهد

اما هنوز این راه حل، راه حل کاملی نیست و این مشکلات را در ناحیه خروج حل می‌کنیم

```
exit_critical_section :
    j = (turn + 1) % n ;
    while (flag [j] == idle)  j = (j + 1) % n ;
    turn = j ;
    flag [i] = idle
```

همه را چک می‌کنیم ببینیم کی هست که idle نیست و اولین کسی که ~~نیستش هست~~ idle نیست می‌است
turn را به اولی می‌دهیم اگر در این n تا فرایند کسی وجود نداشت که idle نیست turn به خودم برمی‌گردد
چون به نوبت کار می‌کنند حداکثر n-1 نوبت طول می‌کشد تا نوبت به من برسد

تمام چک‌هایی که آخر می‌کند برای ایجاد انحصار متقابل
شرط انتظار محدود وجود دارد که حداکثر n-1 نوبت
واضح است شرط سوم هم هست چون رفتار افراد idle را دخالت نمی‌دهد

جلسه پانزدهم 8,19

ارسال تمرین ها به Pedram@aut.ac.ir

راه حل جلسه قبل ربطی به OS ندارست

الگوریتم نانوایی baker's algorithm ← که در حل استفاده می شود دقت گفته می شود

هرکس که می خواهد وارد ناحیه بحرانی شود شماره می دهیم (نوبت دهی) و براساس آن شماره می توانند ترتیب بگیرند یعنی وارد ناحیه بحرانی شوند

```
boolean choosing[n];
int number[n];
```

int number[n]; ← وقتی صفر شد یعنی داوطلب نیست و صف محسوب نمی شود

```
enter_critical_Section :
    choosing[i] = True;
    number[i] = max(number[0], ..., number[n-1]) + 1;
    Choosing = False;
    for (j=0 ; i<n ; j++)
        while (choosing[j]) ;
        while (number[j] != 0 && (number[j], j) < (number[i], i) ;
    }
```

choosing[i] = True; ← نشان می دهد در حال گرفتن number هستیم

number[i] = ...; ← یک شماره بگیری

Choosing = False; ← max را می گیرد یکی به آن اضافه می کند

for (...) ← نگاه می کند هرکس جلوی من است ببرد یعنی اگر من پنجم بشم تا 5 باید بروند بعد نوبت من شود

while (choosing[j]); ← تا وقتی بعدی در حال شماره گیری است صبر می کند تا آن شماره اش را بگیرد

(number[j], j) < (number[i], i) ← برای شکستن تساوی در شماره

اگر من باید number از همه کمتر باشد

تساوی number ها اگر کوچکتر است پس می توانیم وارد ناحیه بحرانی بشوم

```
exit_critical_section :
    number[i] = 0;
```

Subject:
Year. Month. Date. ( )

مرحله شماره‌گیری، مرحله طولانی است ← بیشترین تعداد max بگیرد، به خاطر زمان ماشین آن طولانی‌تر شد پس در وسط کار می‌توان قطع شود ← به همین خاطر همه را باید به حساب آورد حتی آنهایی که در حال شماره‌گیری است چون ممکن است قبل از من باشد و وسط کار قطع شده

همچنین امکان شماره‌دهی مساوی وجود دارد ← این کار را می‌کند اگر دو تا شماره مساوی بود به شماره فرایند برای شکستن حالت تساوی نگاه کن ← اول شماره‌ها چک می‌شود اگر مساوی بود شماره فرایند چک می‌شود

مشکل الگوریتم رفع نشده: number ها تا زمانی بالا می‌روند که overflow بدهد وقتی overflow داد چه کار کنیم؟ حل آن به سادگی نیست ← دچار اشتباه می‌شویم

این راه‌حل تحت شرایطی کار می‌کند ← اگر زمان‌هایی وجود داشته باشد که هیچ کس طالب ورود به ناحیه بحرانی نیست در این صورت number ها صفر می‌شود یعنی زمانی کار می‌کند که این‌ها دائم در رقابت نباشند و از این زمان‌ها این وسط باشد در عمل هم این طور یعنی دائم در حال رقابت نیستند

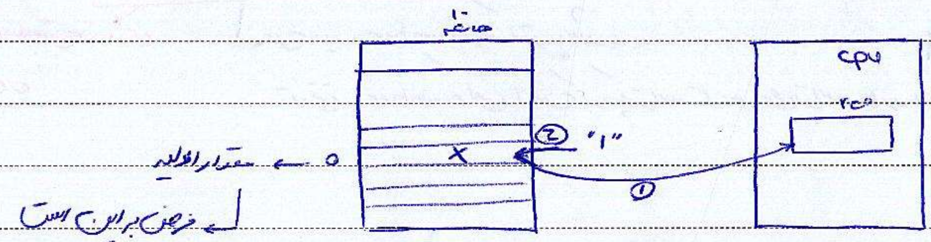
راه‌حل بسیار ساده

تا به حال در مورد دستورالعمل‌های ماشین فرضی نکردیم (بسیار ساده‌اند) اما الان فرض می‌کنیم که مقداری دستورالعمل پیچیده دارد (دستورالعمل‌های غیرساده)

دستورالعمل زبان ماشین

tsl → Test_and_Set ← بیش از یک عمل را انجام می‌دهد

test-and-set-logical



① محتویات حافظه به reg

② به عدد غیر صفر تبدیل یا set می‌کند

اگر درستش بگیریم cpu این دستورالعمل را دارد، راه‌حل بسیار ساده می‌شود.

Subject:
Year. Month. Date. ( )

زبان assembly:

```
enter_critical_section
    tsl  reg2, flag
    cmp  reg2, #0
    jnz  enter_critical_section
    ret
exit_critical_section
    move flag, #0
    ret
```

(tsl reg2, flag ← یکسره است با تست — با یک کلاک منتقل)

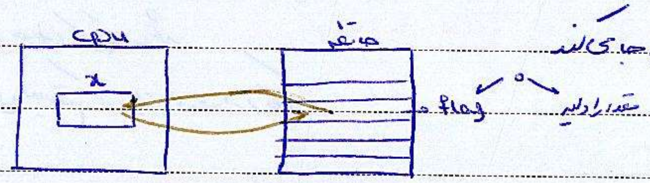
(jnz ← یعنی جستجو این باشد که اگر جستجو باشد zero = ∅ می‌شود)

هر فرایندی که وارد ناحیه بحرانی می‌شود flag را یک می‌کند که اگر flag=1 باشد یعنی کسی دارد از ناحیه بحرانی استفاده می‌کند پس اگر فرایند جدیدی می‌خواهد وارد ناحیه بحرانی شود تا زمانی که flag صفر نشده نمی‌تواند وارد شود. در ضمن هم‌زمان دوتا فرایند نمی‌توانند tsl انجام دهند

در tsl ← هم‌زمانی در وسط کار قطع نمی‌شود چون یک دستورالعمل واحد است.

← فرض در مورد CPU درست بعد ــ Intel این را ندارد چیزی شبیه این دارد

swap →



محتویات این‌ها را جابه‌جا می‌کند

```
enter:
    mov  regx, 1
    swap regx, flag
    cmp  regx, #0
    jnz  enter
    ret

exit:
    mov flag, #0
    ret
```

برای هر منبع می‌توانیم ناحیه بحرانی زیادی تعریف کنیم پس می‌توان به تعداد flag زیادی داشته باشیم

Subject:
Year. Month. Date. ( )

این راه حل انحصار متقابل را کاملا حفظ می کند. (تضمین شده و نمی تواند خراب کند)

مشکل این راه حل کجاست؟

شرط محدودیت زمانی را تضمین نمی کند اما شرط سوم را دارد (کسی که داوطلب نیست، دخالت داده نمی شود)

محدودیت زمانی تضمین نمی شود.

← نوبت به طور کاملا تصادفی گرفته می شود یعنی اگر 5 تا فرآیند داریم یکی گرفت، 9 تای خواهان بین این 4 تا تصادفی است. احتمال غیر صفر دارد که یکی اصلا نوبت بهش نرسد

← این راه حل صد سالی، صحیح نیست

← دنبال راه حل صحیح نبودیم، دنبال چیزی بودیم که تحت شرایطی کار کند

این راه حل تحت چه شرایطی کار می کند؟ اگر به طور متوالی داشته باشیم که هیچ کس داوطلب نباشد کار می کند ولی اگر همیشه فرآیندها در رقابت باشند این راه حل کار نمی کند و در واقعیت این است که زمان هایی وجود دارد که هیچ کس داوطلب نباشد

← اگر در ناحیه بحرانی لیر کند شانس بن بست است و برای تمام این راه حل ها ممکن است اتفاق بیفتد

← قسمت ناحیه بحرانی باید خیلی کوچک باشد

این راه حل ها ست و ما به دنبال راه حلی نیستیم که برای همه موارد کار کند

این راه حل زمانی درست کار می کند که سه دائم در حال رقابت نباشند

← ناحیه بحرانی کوتاه باشد

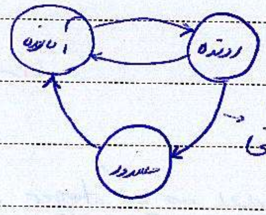
← نکته مشترک تمام راه حل های ارائه شده:

از انتظار مشغولی (busy waiting) استفاده می کند

← برای انتظار از حلقه استفاده می کنند که زمان پردازنده ی جهت لقی ای اشغال می شود، هیچ کاری مفیدی انجام نمی شود ← راه حل های ارائه شده که فرآیندها را مسدود می کند

می خواهیم راه حلی ارائه کنیم که انتظار مشغول نداشته باشد و از حالت مسدود استفاده می کنیم.

آنهایی که می خواهند منتظر ناحیه بحرانی باشند به حالت مسدود می برند



انتظار برای ناحیه بحرانی را مثل درخواست برای سرویس در نظر بگیریم

عمومی بیان می شود که تحت عنوان یک ابزار پیاده سازی شده

می شود (Semaphor پرچم) ← یک متغیر که به همراه صف که استفاده می کنیم چه موقع باید وارد ناحیه بحرانی شده و چه موقع باید استراحت کرد

ابزاری برای هماهنگی فرآیندها (ورود به ناحیه بحرانی)

یک ساختار داده ای است ← یک متغیر int
↙ یک صف

در حالت مسدود فرآیندها بنابر سرویسی که می خواستند در یک صف منتظر می شوند این صف Semaphor را هم در این جا می گذاریم ← متغیری که به بقیه سرویس ها می شود

Semaphor ← یک مجوز برای وارد شدن به ناحیه بحرانی که برای هماهنگی به کار می بریم

11365
83201969
۵۵۶ - ۸۸۷۷۲۴۰-۴
۵۵۸ - 139625 - 1

جلسه شانزدهم ۱۹/۸/

تمام راه حل های ارائه شده از busy waiting استفاده می کنند

Semaphore (سمافور، راهنما)

ابزاری برای همگامی

ساختاری شامل یک integer و یک صف دارد. (از فرایندها)

عملیاتی که روی این ساختار داده هست را هم باید بگوییم

عملیات روی Semaphore

مقداردهی اولیه ← مقدار integer را باید set کنیم یعنی برای شروع به آن بدهیم

عملیات مهم روی Semaphore ← wait

↙ signal

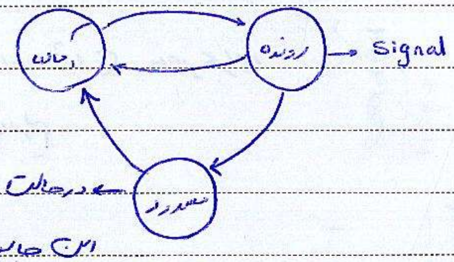
wait ← برو استفاده کن

wait ← یک واحد از integer کم می کند نگاه می کند که آیا مقدار منفی شده یا نه. اگر منفی شد فرایند مسدود می شود (در صف)

و در صف قرار می گیرد در غیر اینصورت فقط یک واحد کم می کند و عبور می کند

signal ← یک واحد اضافه می کند و چک می کند که مثبت نباشد ← اگر مثبت نباشد یکی از فرایندهای مسدود شده

آزاد می شود (آن که سر صف است آزاد می کند. آن که نوبتش است)



← در حالت signal یکی از فرایندهایی که

این جا بوده به حالت آماده می رود برای

خود کسی که signal را اجرا کرده اتفاقی نمی افتد

Subject:
Year. Month. Date. ( )

```
struct Semaphore {
        int count;
        queueType queue;       یک صفی از فرایندها
    }

void wait (Semaphore s)
{
    s.count--;
    if (s.count < 0) {          => اگر منفی شد blocked می‌کند
        place this process in s.queue;
        block this process;
    }
}

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

Binary Semaphore

نوع دیگری از semaphore ← binary sema... ← به جای integer یک bool استفاده می‌کنند ولی منطق روش کار همین است

اصطلاحاً می‌گویند عملیات wait و signal اتمی هستند یعنی نمی‌شود تقسیمش کرد (مثل Test & Set)
باید یا همه انجام شود، نمی‌شود تجزیه کرد. اگر wait را می‌گیریم باید کل wait انجام شود یا هیچ انجام نشود.
پیاده‌سازی همچین چیزی راحت نیست مثل اینکه بگم وقفه‌ای نیاید

Subject:
Year. Month. Date. ( )

این استفاده از semaphore در ناحیه بحرانی است.

حل مسئله ناحیه بحرانی (برای n فرآیند) ← busy waiting نداریم

```
/* program mutualexclusion */        ← انحصار متقابل
const int n = /* number of processes */
semaphore s = 1;
void P(int i)
{
  while (True) {
      wait (s);
      critical_section(i);
      signal (s);
      non_critical_section (i);
  }
}

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

مقدار اولیه 1 داریم چون این کاربرد 1 بودن دیگه خودش

1 ← 0 اول هر کس زودتر بیاد وارد ناحیه بحرانی می‌شود

کم کند منفی می‌شود دیگر نمی‌تواند وارد ناحیه بحرانی شود

کار ما تمام شد یکی از فرآیندهای منتظر را آزاد می‌کند اگر کسی نبود که

آزاد کند 1 count می‌کند تا کسی دیگر بیاید از

ناحیه بحرانی استفاده کند

این راه حل ساده هم کار می‌کند هم busy waiting نیست

← قدر مطلق عدد منفی (s.count) نشان می‌دهد چند فرآیند در صف داریم.

1. انتظار محدود داریم چون در صف می‌گذاریم و نوبت دهی می‌کنیم

2. چون مقدار اولیه 1 بود فقط یک فرآیند می‌تواند وارد شود اگر 2 می‌دادیم 2 تا فرآیند وارد می‌شدند

3. کسانی که داوطلب نیستند که در صف قرار نمی‌گیرند پس اصل پیشرفت داده می‌شود

PAPCO

Subject:
Year. Month. Date. ( )

پیاده‌سازی این با شرط اتمی بودن wait و signal خیلی سخت است ولی بدون این شرط‌ها این راه حل کار نمی‌کند مثلاً فرض کنید ۲ پردازه اتمیک wait را انجام داد و کانتکست سوئیچ شد و پردازه بعدی هم ۱ می‌بیند و آن هم وارد ناحیه بحرانی می‌شود.

با داشتن این ابزار می‌توانیم کاربردهای دیگری نیز داشته باشیم

← دو فرآیند P_1 و P_2 داریم که دستورالعمل‌ها را به صورت همروند اجرا می‌کنند
سیستم طوری است که S_1 باید زودتر از S_2 انجام شود (وقتی که S_1 چیزی را تولید می‌کند که S_2 باید از آن استفاده کند)
S_1 زودتر از S_2 اجرا شود؟ حل
معمولاً ترتیب تضمین نیست که هم‌زمانی ← با ترتیب یک semaphore این کار را انجام می‌دهیم

```
semaphore sync = 0   =>
        P1                      P2
        ....                    ....
        S1                      wait(sync);
        signal(sync);           S2
        ....                    ....
```

تبیین S_1 و S_2: فرض می‌کنیم S_1 را اجرا می‌دهیم و قبل S_2، block می‌شود وقتی S_1 انجام شد S_2 را آزاد می‌کند اگر هم S_1 زودتر بخواهد اجرا شود و sync=1 می‌شود قبل wait می‌کنیم ۰ می‌شود و منفی نیست S_2 که هم انجام می‌شود.

پیاده‌سازی:
دو راه حل وجود دارد:
راه حل اول: چیزی که اتمیک را نقض می‌کند وقفه هم‌زمانی است پس اگر همه وقفه‌ها را از کار بیندازیم دیگر قطع نمی‌شود
البته باید برای از کار انداختن وقفه‌ها در System mode باشیم

```
void wait (semaphore s)
{
  inhibit interrupt;        -> interrupt از کار انداختن
  s.count--;
  if (s.count < 0) {
        place this ...
        block this ..., allow interrupts   -> interrupt به کار انداختن
  }
  else  allow interrupts
}
```

Subject:
Year. Month. Date. ( )

← کی داره این کارها را انجام میده؟ یکی فکر میکنه چون در user mode است، عملاً این کارها را
OS انجام میده و وقتی دستور میکنه خودش را دستور میکنه

```
void Signal (semaphore s)
{
    inhibit interrupts;
    s.count ++;
    if (s.count <= 0) {
        remove a ... ;
        place ... ;
    }
    allow interrupts;
}
```

← راه حل خوبی نیست چون از کار انداختن interrupt خطرناک است چون ممکن است interrupt های
مهمی بیاید که همان موقع باید جوابش را بدهیم (مثلاً اینکه یک متغیری از max خودش گذشت مثل درجه حرارت)
ممکن است استفاده داشته باشد و بستگی به کاربرد داره
ما کاری می‌کنیم که این را قفل نکنیم ← زمان ما در این فاصله از دست میره

آیا راه حل مناسب‌تری داریم؟ همان جعبه جادویی

```
void wait (semaphore s)
{
    while (!testset (s.flag)) ;
    s.count --;
    if (s.count < 0) {
        place ... ;
        block ... ;
        set s.flag to zero ;
    }
    else set s.flag to zero;
}
```

ممکنه نوشته یک flag هم داره ← DS آن عوض شده

PAPCO

Subject:
Year. Month. Date. ( )

```
void signal                    s.flag
{
  while (!testset (s.flag));
  s.count ++ ;
  if (s.count <= 0) {
    remove ...
    place ...
  }
  s.flag = 0;
}
```

← wait(s) و signal(s) را ناحیه بحرانی تعریف کردیم یعنی گفتیم اول test & set کند که ببیند می‌تواند flag را بدست بیاورد یا نه

```
{ test & set (flag)          { test & set (flag)
{ wait (s)                   { signal (s)
{ set flag to 0              { set flag to 0
```

test&set ← فقط شدیم که flag را بدست آوردیم

این جا دیگر interrupt نداریم با این روش اتمیک بودن نداریم ولی اثر اتمیک بودن را بدست می‌آوریم
این جا هم busy wait داریم چون wait یا signal می‌کنیم چون نمی‌توانی flag را بدست آوری پس نمی‌توانی wait و signal کنی. مکانیزمی را از جا بپرید مهم نیست چون نمی‌تواند flag را بگیرد پس نمی‌تواند wait یا signal کند

ظاهر قضیه این است که داریم نقض عرض می‌کنیم روش‌های قبلی برای ناحیه بحرانی کوچک مناسب بودند ما در ناحیه بحرانی نمی‌دانستیم دقیقاً چقدر طول می‌کشد ولی در این جا دقیقاً می‌دانیم چقدر طول می‌کشد

PAPCO