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 ← فرض کردیم که flag را بدست آوردیم

این جا دیگر interrupt را غیرفعال نکردیم این روش اتمی بودن ندارد ولی اگر اتمی بودن را بدست می‌آوریم
این جا هم می‌گوییم چون داریم wait یا signal می‌کنیم چون می‌تواند flag را بدست آورد پس
می‌تواند wait و signal کند. حالا که مرزبندی را از ما بگیرد 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 بود و ما
در ناحیه بحرانی نمی‌دانستیم دقیقاً چقدر طول می‌کشد ولی در این جا دقیقاً می‌دانیم چقدر طول می‌کشد

Subject:

Year. Month. Date. ( )

میدانیم test & set را برای ناحیه بحرانی برنامه که نوشته است در هیچ کاربران تمام می‌شود ولی معطل نمی‌مانند چون انجام عمل wait و signal کوتاه است.

این ناحیه بحرانی اصلی را wait و signal کنترل می‌کنند

← بر اساس صف و سمافور کار می‌کنند

test & set (flag)

wait (s) → ترتیبی را حفظ می‌کنند

flag = 0

ناحیه بحرانی

test & set (flag)

signal → از صف سمافور خارج کند

flag = 0

اجرای wait و signal ← پشتیبانی OS را می‌خواهد

← صف و سمافور در هسته قرار دارند

حتماً نباید system call باشد می‌تواند باشد ولی همچنین می‌تواند library باشد که از چند تا system call استفاده کند به طور صریح یعنی از system call استفاده می‌کند

Subject:
Year. Month. Date. ( )

جلسه هفدهم 87/8/21

حل مسائل هماهنگی بوسیله سمافور:

۱- تولیدکننده و مصرف کننده ـ هماهنگی، که توسط بافر بود حالا با سمافورها انجام میدهیم

```
/* Program boundedbuffer */
const int sizeOfbuffer = /* bufferSize */
Semaphor S = 1 ;
Semaphor n = 0 ;
Semaphor e = n ; sizeOfbuffer
```

در تولیدکننده:

```
void producer ( )
{
while (True) {
  → produce ( );
    wait (e);       ← کنترل پر بودن
    wait (S);       ┐ دسترسی به بافر را کنترل می کند
  → append (S);     │ آیتم را به بافر append کن یعنی در بافر بگذار
    Signal (S);     ┘
    Signal (n);
  }
}
```

```
void consumer ( )
{
while (True) {
    wait (n);       ← کنترل خالی بودن    اگر خالی باشد نمی توانیم از آن چیزی برداریم
    wait (S);
  → take ;          مقدار را از بافر بردار
    Signal (S);
```

Subject:
Year. Month. Date. ( )

تعداد مکان های خالی را یکی زیاد می کند

```
signal (e);
→ consume ();
}
}

void main
```

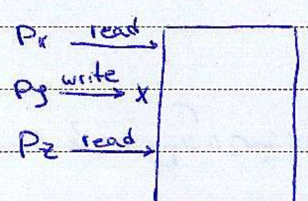
یعنی s برای دسترسی انحصاری به کار رفته در نتیجه مقدار اولیه = 1
اگر می خواستیم در بافر بگذاریم ← جایی که بگذاریم باید پر نباشد
از n بخوانیم ← n ... e خالی را

چون e برای تعداد پر نبودن است باید به اندازه سایز بافر باشد چون اگر اضافه کنیم یکی ازش کم می شود
می توانیم اضافه کنیم تا جایی که صفر بشود بعد منفی می شود و دیگر نمی توانیم از wait عبور کنیم
n برای تعداد خالی نبودن است ← تعداد مکان های پر شده را نشان می دهد
e تعداد مکان های خالی را نشان می دهد.

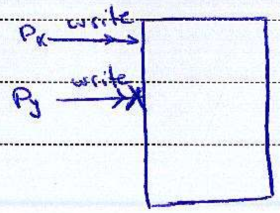
← بین همه این سمافورها فقط s برای ناحیه بحرانی به کار رفته و بقیه برای زمان های دیگر است و این مثال می گوید سمافور برای ناحیه بحرانی فقط نیست

2. خواننده ها و نویسنده ها: Readers & writers prob.

اینجا یک object مشترک مثل file وجود دارد و فرآیندهای مختلف می توانند از این object بخوانند یا بنویسند ولی
بدون هماهنگی نمی توانند این کار را انجام دهند مثلاً در حین خواندن یکی دیگر نمی تواند در آن بنویسد

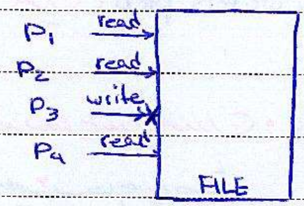


برای read کردن همزمان می توانند انجام دهند چون اطلاعات عوض نمی شود و اگر کسی دارد read می کند فقط نمی توان در آن write کرد



ولی وقتی write می کنیم هیچ کس دیگری نمی توانیم انجام دهیم چون اطلاعات قابل اعتماد نیست و عملیات write انحصاری است

Subject:
Year. Month. Date. ( )



اینکه P4 هم باید منتظر بمانند یا نه بستگی به مسئله دارد:

1. سیستم عامل به صورتی است که تمام کارها پشت هم انجام شوند

2. شاید اینها تا الان بخواهند بخوانند write با پیچی کار دارند

First readers & writes prob ← به P4 اجازه خواندن دهیم

Second " " " ← به P4 اجازه ندهیم قبل از write بخواند

انحصاری بودن write در هر دو مشترک است.

First readers & writers prob: ←

دو فرآیند ← هم read و هم write می‌کنند

فرآیند نویسنده: Pw (نوشتن انحصاری است)

wrt ← برای دسترسی انحصاری به کار بردیم پس مقدار اولیه آن یک است

Pw

```
wait(wrt);
writing();
signal(wrt);
```

فرآیند خواننده: Pr

اگر خواننده اول باشد یا نه رفتارش متفاوت است

اولی باشی ← باید چک کنی کسی می‌نویسد

اولی نباشی ← لازم نیست چک کنی

readcount ← تعداد خواننده‌ها، پس هر کسی می‌خواهد بخواند باید این را یکی زیاد کند

mutex ← دسترسی به readcount را انحصاری می‌کنیم پس مقدار اولیه‌اش یک است

بعد از خواندن باید ببینی اگر آخرین خواننده هستی باید آن کسی را که منتظر write است آزاد کنی

Pr

از این چهار دستوری برای خواندن را انجام دهی

برای متغیر readcount است

```
wait(mutex);
read count ++;
if(read == 1)
    wait(wrt);      ← چک کنی کسی در حال write نباشد
signal(mutex);
reading()
wait(mutex)
readcount--;
if readcount == 0)  ← آخری هستی
    Signal(wrt);
Signal(mutex);
```

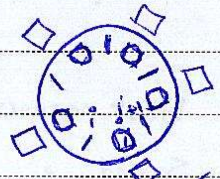
P4PCO

Subject:

Year. Month. Date. ( )

(بعنوان تمرین) Second readers & writers prob را انجام دهید

3. فیلسوف‌ها در حال غذا خوردن: Dining Philosophers



این فیلسوف‌ها چینی هستند

هرکسی می‌خواهد غذا بخورد باید دو تا چوب در اختیار داشته باشد

وگرنه باید منتظر بماند

فرایندها: فیلسوف‌ها (همیشه در حال خوردن نیستند، یه وقت‌هایی فکر می‌کنند)

منابع: چوب‌ها

function: eating (در حال غذا خوردن) و thinking

برای هر chopstick یک semaphore

دسترسی انحصاری است بین دو همسایه

```
Semaphore chopstick [5];
void P(i)
{
 while True {
  wait(chopstick[i]);
  wait (chopstick [(i+1) % 5]);
  eating ( );
  Signal (chopstick [i]);
  Signal ( chopstick [(i+1) % 5]);
  thinking ( );
 }

void main ( )
{
 parbegin (P(0), P(1), ..., P(4));
}
```

} => بالاخره بعد از اینکه این دو را برداشت

می‌خورم

اما این راه حل یک مشکل دارد.

Semaphore ابزاری قوی برای هماهنگی است ولی مشکلات دارد

Subject:
Year. Month. Date. ( )

مشکلات Semaphore

۱. کل Semaphore در اختیار برنامه‌نویس است و این باعث مشکلاتی می‌شود.

۲. بکارگیری صحیح ← به ازای هر wait باید یک signal داشته باشیم و بالعکس چون عمل به درستی برسد

wait(s);
=
signal(s);

= صحیح

signal(s);	wait(s);	wait(s);
=	=	=
wait(s);	wait(s);	wait(s);
از بین رفتن انحصار متقابل	بن بست	

2. بن بست. ممکن است پیش بیاید وقتی از درست استفاده کنیم

هر دو در این حالت قانونی هستند

P1	P2
=	=
① wait(P);	② wait(Q);
④ wait(Q);	③ wait(P);
=	=
signal(P);	signal(Q);
signal(Q);	signal(P);

① ثبات کم زمانی
④ ← مسدود چون در P2 wait شده
③ ← مسدود می‌شود ← نمی‌تواند رد بشود
P را منتظر می‌کند

اینجا هر دو ترتیب درست هیچ جایی برای خطا نمی‌مانند.

این مشکلات در مسئله Dining Philosopher وجود دارند ← هم بی‌درنگ برابر منتظر روی باشند در نتیجه هیچ وقت این مشکل حل نمی‌شود چون نمی‌دانیم کی منتظر این منبع است

← می‌شود کار برهای از این خودشان برد که به درد بخورد

راه حل این مشکل این است که این اختیار را به برنامه‌نویس ندهیم که هر کاری می‌خواهد بکند پس به سراغ ابزار دیگری می‌رویم ← monitore (مانیتور یا ناظر)

اینجا جا نشین هم نیستند بلکه هر کی در جای خودش استفاده می‌شود و جای همدیگر را نمی‌گیرند

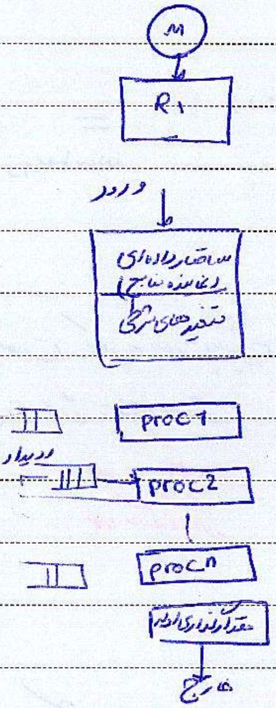
Subject:
Year. Month. Date. ( )

تمرین‌های 9,8,6,7,14,15

جلسه هجدهم 8, 26

یکی از ابزارهای دیگر جهت ایجاد هماهنگی بین فرآیندهای باشد البته می‌تواند جایگزین سمافور شود. اختیاراتش محدودتر است یک سری از اشتباهات سمافور را نمی‌دهد.

سمافور کنترل کامل روی منبع دارد و روی عملیات هم کنترل کامل دارد
پس اختیارات مانیتور محدودتر می‌شود



مانیتور monitor

۱. در هر لحظه فقط باید یک فرآیند داخل monitor فعال باشد
در غیر این صورت انحصار متقابل نقض می‌شود
تعداد این صف‌ها می‌تواند چندین تا باشد
اگر یکی روی شرایطی رفت و منتظر شد می‌توان فرآیند دیگری را وارد کرد و این انحصار متقابل را از بین می‌برد

فرآیندی یا خارج شده است یا منتظر است.
این شرایط متغیرهای شرطی هستند

مثال) تولیدکننده و مصرف‌کننده.

```
/* Program producer Consumer */
monitor boundedbuffer;
chare buffer[N];            → می‌تواند هر چیزی غیر از char باشد
int nextin, nextout;        → Semaphor mutex
int count;
cond notfull, notempty      condition
```

(ساختمان داده‌های مانیتور)

Subject:

Year. Month. Date. ( )

```
void append (char x)
{   if (count==N)          wait(mutex)
        cwait(notfull)
    buffer[nextin]=x;
    nextin=(nextin+1)%N;
    count++;
    csignal(notempty)      signal(mutex)
}

void take(char x)
{   wait(mutex);
    if (count==0)
        cwait(notempty);
    X=buffer[nextout];
    nextout=(nextout+1)%N;
    count--;
    csignal(notfull);      signal(mutex);
}
    nextin=0; nextout=0; count=0;

void Producer()
{   char x;
    while(true){
        produce(x);
        boundedbuffer.append(x);
    }}
```

شرط بررسی

بین monitor و consumer و producer

Subject:
Year. Month. Date. ( )

```
void Consumer
{ char x;
    while (True) {
    boundedbuffer.take(x);
    consumer(x);
  }
}

void main()
{
  parbegin(producer, consumer);
}
```

استفاده از این monitore اضافی هست. این اضافه ی بولد کدی مشخص شده است؟

در keyword ، monitor هست که وقتی این کلمه compiler می‌بیند compiler به دنبال این keyword می‌گردد که یک کد سری باید به آن اضافه شود و این یک ربطی می‌باشد و باید یکسری چیزها به آن اضافه شود تا مناقضت شود در حالت بطور دقیق متدهای * را اضافه می‌کند

← هنگامی که از monitor استفاده می‌شود باید از compiler به معنای خاصی برای monitor داشته باشد استفاده کند. ـ در حالت دقیق‌تر باید در تست. wait هم چیزهایی اضافه می‌شود

مثال) فیلسوف‌ها به غیر از حالت eating و thinking حالت hungry هم دارند

```
monitor Dining Philosophers;

enum itemation {hungry, eating, thinking}
intation state[5];
Condition Self[5];
```

Subject:

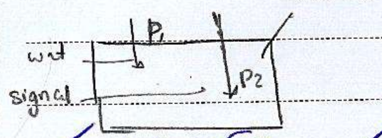
Year. Month. Date. ( )

```
void pickup(i)
{
  state[i] = hungry;
  test(i);
  if(state[i] != eating)
      cwait(self[i]);
}

void putdown(i)
{
  state[i] = thinking;
  test((i+1)%5);
  test((i+4)%5);
}

  test(i)
  {
    if(state[i] == hungry && state[(i+1)%5] != eating &&
                             state[(i+4)%5] != eating) {
      state[i] = eating;
      csignal(self[i]);
    }
  }

For(i=0; i<5; i++)
{ state[i] = thinking;
}

void p(i)
{ while(True) {
  Dining Philosoplers.pickup(i);
  eating();
  Dining philosophers.putdown(i);
  thinking(i);
}}
```

PAPCO

Subject:
Year. Month. Date. ( )

مثلاً) فرض کنید یک monitor داریم که P_1 proc وارد این می‌شود وسط کار wait می‌کند از طرف دیگر یک
P_2 وارد monitor می‌شود پس از مدتی P_2 متغیری که P_1 روی آن wait شده signal می‌شود از این لحظه
به بعد P_1 باید ادامه پیدا کند یا P_2



جواب صحیح بستگی دارد چون ممکن است بگوییم P_2 که متغیر P_1 را سیگنال کرده است باید تمام
شود تا P_1 بعد از آن شروع کند

پس اینکه کدام یک ادامه پیدا کند بستگی به مثال دارد ساده‌ترین راه حل این است که signal کردن P_2 آخرین کاری باشد که
P_2 می‌کند و پس از آن بیرون برود ولی ممکن است نتوان همچین کاری کرد

سؤال: چگونه این راه حل را اجرایی کنیم؟

1. monitor که عملیات را کنترل می‌کند ولی نمی‌تواند این کار را بکند
2. نیاز به پشتیبانی compiler داریم چون اگر اینها را خودمان اجرا کنیم ولی باید دقت زیادی کرد ولی نه فقط به پشتیبانی OS نیاز داریم compiler آن را compile می‌کند سپس برای کنترل پیغام‌هایی که اضافه می‌شود نیاز به OS هم دارد.

3. اینکه P_1 باید اجرا شود یا P_2

چه زبان برنامه‌سازی را می‌شناسید که monitor را پشتیبانی کند؟ Java

کلمه monitor را نداریم ولی اگر یک object را synchronized تعریف کنیم این خاصیت را دارد
مدیریتش هم چون compiler نیز وجود دارد باید این کارها را انجام می‌دهیم

PAPCO

Subject:

Year. Month. Date. ( )

جلسه نوزدهم 28 / 8

OS های موجود بطور مستقیم از مانیتور پشتیبانی نمی‌کنند بنابراین داشتنش مشکل است.

monitor ← نیاز به پشتیبانی compiler

semaphore ← نیاز به پشتیبانی OS دارد

← می‌توانیم برایش یک library routine تهیه کنیم که اجرا شود.

استفاده از فراخوان‌های سیستمی send و recieve ← تبادل پیام یا message passing

این راه هم OS ها دارند و نیاز به پشتیبانی خاصی ندارد خیلی ساده است ولی کاربرانش مشکل است

دو تا فرآیند داریم و می‌توان از هر فرآیندی به فرآیندی دیگر message ارسال کند

این کار چون وسط می‌شود یعنی به OS نیاز دارد و فقط ایجاد کننده این ارتباط است

message هر چیزی است داخلش هر چی می‌تواند باشد مثلاً pointer یک ساختار

Send (m, P2) → send می‌کند

recieve (m) ; → تا وقتی که این اجرا نشده P2 ، message را نمی‌گیرد

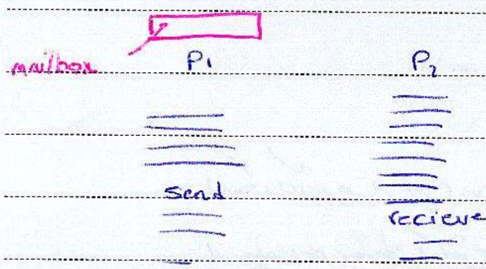
← فقط یک message

از روی header این که مال کیه می‌فهمه که از کی گرفته

هر کسی حق دارد message رابفرستد

هر message به دریافت کننده خود.

مستقل send ، فرآیند بعد حتماً باید recieve را اجرا کند



وقتی send نشده چیزی recieve نمی‌شود

انتظار نیست send باید زودتر از recieve اجرا شود

اما این شرط سنگین است

راه حل: (در حالت منطقی send باید زودتر از recieve باشد)

برای هر فرآیند OS یک بافر به نام mailbox برایش درست می‌کند هر پیامی که به فرآیندی برسد داخل mailbox اش می‌رود

هر وقت دستور recieve را اجرا کرد یک پیام از داخل mailbox بر می‌دارد

(چون message ها در mailbox پشت سر هم لم نمی‌شوند)

← recieve از نوع سیستم call های blocking است یعنی اگر نتوانست کاری انجام دهد (هیچ پیامی نداشتیم) فرآیند را مسدود می‌کند و نگه می‌دارد در حالت مسدود تا زمانی چیزی داخل mailbox بیاید ← کار بر چیزی نمی‌دهد

بهترین نوع recieve همین است چون مطمئن هستی که وقتی از block خارج می‌شوی حتماً اون چیزی را که می‌خواستی داری.

Subject:
Year. Month. Date. ( )

حل مسئله producer_consumer

```
void Producer ( )
{ int item ;
  message m;
  While (True) {
      produce (item);
      recieve (m);
      build_message (m, item) ;   <----  item را کپی message میکند پیش پردازش‌های / آن را رها میکند
      Send (consumer, m) ;  ---->  تا که پیام یکی send شود
  }
}

void Consumer ( )
{ int item, i;
  Message m;
** for (i=0; i<N; i++) Send (producer, m)
  While (True) {
  --> recieve (m) ;            Send که برای گرفتن پیام‌ها recieve می‌شود
      extract_item (item);     از message، item را بخش پیغام دقیق جدا میکند
      Send (producer, m); -->
      consume (item); -->      مصرف کن
  }
}
```

PAPCO

Subject:
Year. Month. Date. ( )

فرض می‌کنیم mailbox نداریم یعنی تا send نکنیم نمی‌توانیم recieve کنیم.

← recieve(m) در producer یک مجوز است که consumer می‌گوید من جا دارم بفرست و وقتی جا نباشد producer در این جا معطل می‌شود

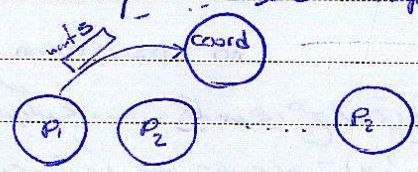
← برای اینکه این دو باهم جلو روند consumer به ازای هر دریافت مجوز صادر می‌کند و این طوری است که تولیدی شود مصرف می‌شود تولید می‌شود تا وقتی مصرف نشده چیزی تولید نمی‌شود

** در این خط قبل از هر کاری N تا مجوز می‌فرستد پس هیچ وقت یعنی recieve در producer بلاک نمی‌کند چون که به mailbox آن پر است و هیچ وقت block نمی‌شود اگر ۱ مجوز بفرستیم بالا معنی است یعنی یکی تو است وی در این جا producer می‌تواند تولید کند و باری نداشته باشد به consumer مصرف می‌کند

← می‌توانیم Semaphore را با استفاده از این بسازیم
اگر بتوانیم سمافور را شبیه‌سازی کنیم می‌توانیم تمام کارهایی را که می‌کردیم الان هم انجام بدهیم

شبیه‌سازی Semaphore

تعدادی فرایند هستیم که می‌خواهیم از یک Semaphore مشترک استفاده کنیم



rec
open message
wait Operation
if not negative send

مثل فرهای کارای
یکی از این‌ها باید Semaphore را پیاده کند به این Semaphore : یک فرایند کلی ایجاد می‌کنیم فرایند کلی در حالت recieve قرار می‌گیرد (block)

P1 یک message (wait) درست می‌کند به coord می‌فرستد بلافاصله به حالت rec می‌رود P1: wait(S)
مشتری ما هم شاید چند تا مشتری دیگر signal کنند

messag | wait | S | .... |

coordinate message را باز می‌کند می‌بیند wait بوده است

پس wait را انجام می‌دهد اگر منفی نشد send می‌کند Send(coord,)
recieve => block شد

در حالی که ارزش منفی شد اگر منفی نشد چیزی برایش نفرستاده است (مثل wait است)

Subject:
Year. Month. Date. ( )

گرسنگی خودرا داشته باشیم. اما monitor را هم می‌توانیم پیاده‌سازی کنیم (اگر OS آن را نداشته باشد)

مثال 5 ← 2، 4، 5، 11

جلسه بیستم

بن‌بست Deadlock

این بحث را زیاد کش نمی‌دهیم چون در سیستم‌های تک پردازنده ما با آن‌ها کار داریم چندان مطرح نیست چون اکثراً هر زمانی که چیزی می‌خواهد از OS درخواست می‌کند البته خودش هم می‌تواند اقدام کند ولی اکثراً این جوری نیست. البته این بحث در جاهای خود بیشتر است.

اگر 4 شرط زیر برقرار باشد بن‌بست پیش می‌آید

1. انحصار متقابل mutual exclusion یعنی اگر منبع انحصاری نباشد بن‌بست پیش نمی‌آید. (مثالی از منبع غیرانحصاری رم، تلفن است)

2. نگهداری و انتظار hold & wait - دو تا منبع داریم برای هر دو تا نخ داریم اگر یکی را رها کنیم (که نمی‌کنیم) ← (وقتی که یکی از نخ‌ها مال یک منبع را از دست می‌دهیم آن را رها نمی‌کنیم تا دیگری را به دست بیاوریم) وقتی کارها پیش می‌رود که همه نخ‌ها را به دست بیاوریم و گرنه گرسنگی پیش می‌آید.

3. قبضه ممکن نباشد no preemption

قبضه پردازنده است که اگر بگیرید بعد دوباره بدهی هیچ اتفاقی نمی‌افتد ولی هر منبع این جوری نیست (اگر یک منبع را که داریم تا کارش تمام شود ممکن است بن‌بست پیش آید ولی اگر بتوان آن را قبضه کرد دیگر بن‌بست پیش نمی‌آید)

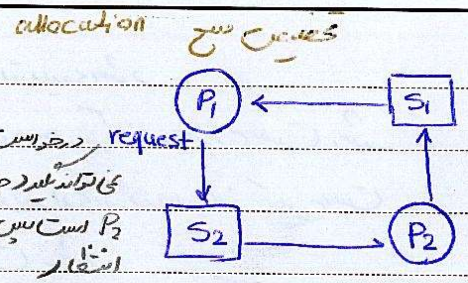
4. انتظار چرخشی circular wait

شرط اختیاری نیست بلکه ممکن است در حین کار پیش بیاید. همه نخ‌ها را برای که قرار است این دیگری را داشت

PAPCO

Subject:

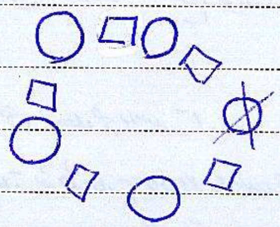
Year. Month. Date. ( )



درخواست می‌خواهد ولی نمی‌تواند بگیرد چون در اختیار P_2 است پس می‌رود در حالت انتظار

حالا P_2 هم درخواست را دارد ولی در اختیار P_1 است پس می‌رود در حالت انتظار

در مثال فیلسوف‌ها انتظار چرخشی بود که در آن انتظار چرخشی بین 5 تا فرایند است و هیچ راهی هم ندارد مگر اینکه یکی را باطل کنیم (ساده) و یا راه‌های دیگر

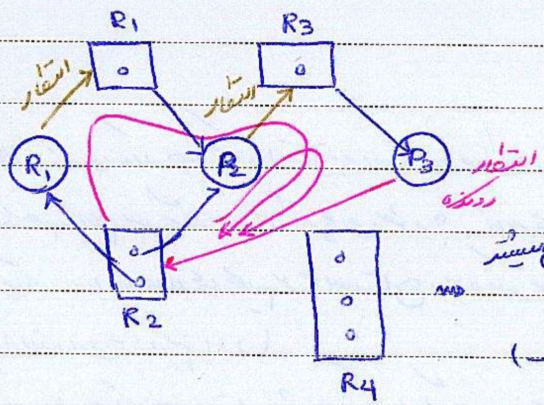


مثال)

R ← منبع resource

* تخصیص منابع
* request

R_1 را می‌خواهد ولی در اختیار P_2 است می‌رود در انتظار



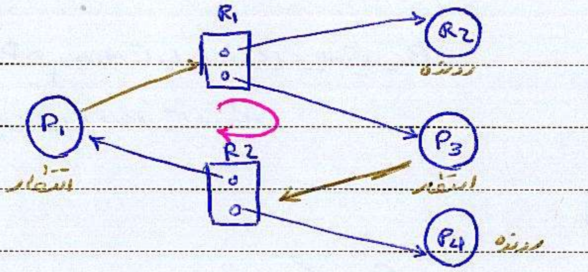
از منبع دیگر نوع بیشتر از یکی داریم (هارد دیسک)

در این وضعیت سیستم در حالت بن بست نیست چون ما امیدواریم (مطمئن نیستیم) که P_3 تا آخر برود و منبع را آزاد کند و بعد P_2 تا آخر و بعد P_1 تا آخر. خلاص می‌شود.

* حالا هر 3 تا در انتظار رفتند منتظر هم اند. بن بست داریم. دو تا حلقه به وجود آمده

Subject:

Year. Month. Date. ( )



مثال) حلقه دیده می‌شود

* لزومی ندارد همه‌ی اینها از بن بست باشند

در حلقه هر تعدادی که در حلقه اند ممکن است در بن بست باشند

با وجود حلقه در بن بست نیستیم می‌توانیم امید داشته باشیم که P_2 تا R_1 را آزاد کند پس P_1 می‌تواند R_1 را بگیرد و تا انجام شود

راه برای خلاصی است.

← وجود حلقه شرط لازم برای بن بست است ولی کافی نیست

(اگر بن بست باشد حتما حلقه هست)

راه حل های بن بست:

1_ در OS های تک پردازنده ای به کاری که بله چون کم اتفاق می‌افتد ← بدون راه حل. در این حالت چون راه حل نداریم یعنی تشخیص بن بست هم نداریم پس اگر پیش آمد چه می‌شود؟ کاربر می‌بیند hang کرده پس یا process را از بین می‌برد یا reset می‌کند. و یا با اصطلاح می‌گوییم بن بست رخ داده فکر می‌کنیم یک مشکل بوده و چون هم نیست راه حل ارائه دادن نداریم!!!)

(در تک پردازنده چون یک پردازش داریم کمتر بن بست پیش می‌آید ولی OS های چند تا پردازش بود ...)

2_ الگوریتم بانکداران _ اجتناب از بن بست Dead lock Avoidance

مسیر ما را طوری تعیین می‌کند که به سمت بن بست نرویم

3_ تشخیص بن بست (مشخص می‌کند ولی راه حل نمی‌دهد. خود همین تشخیص قدم اول است که قطعا ساده هم نیست)

4_ جلوگیری از بن بست Dead lock Prevention ← اصلا در سیستم های تک پردازنده ای کاربرد ندارد

در سیستم های توزیع کاربرد دارد چون به شدت اتلاف منابع دارد

طراحی اولیه سیستم به گونه‌ای که یکی از این 4 شرط حتما وجود نداشته باشد ← نقض حتمی یکی از این 4 شرط

P4PCO

Subject:
Year. Month. Date. ( )

- الگوریتم بانکداران که روی آن مانور زیادی داده شده در هیچ جا کاربرد ندارد ولی برای سوال طرح کردن خوب است! (می شود پیچاند)

منابع را یک جوری تخصیص می دهیم که شما به تحت بن بست نروی. از ترس اینکه بن بست نشود منابع را دارد ترحم داری نکه می ری بی خطا می دهد!!

مثال ربات ها (در نقش زنبورها) همان مسیری را که امید برود دیدا!!

- تشخیص بن بست در سیستم های توزیع خیلی مصرف دارد (با ساختن گراف تخصیص منابع) و بعد از تشخیص با برگشت به عقب از این بن بست خلاص می شود (در حالی که هنوز بن بست رخ نداده ...)

(لازم در روش تخصیص نیست)

جلوگیری از بن بست در سیستم های توزیع شده دیده شده ولی زیاد رایج نیست

مثال برای الگوریتم بانکداران

فرض کنید P_1 و P_2 و P_3 یک منبع R_1 که 12 عدد از آن موجود است.
از قبل این اطلاعات اولیه به ما داده شده که حداکثر تقاضا ها از منبع R_1 به صورت زیر است:
(رحم نیست از کجا می دانستند)

P_1 10
P_2 4
P_3 19

15

حالت فعلی در زمان T

P_1	5
P_2	2
P_3	12

در این لحظه

این حالت بد است یا خوب؟ یعنی آیا احتمال دارد ما را به سمت بن بست ببرد یا نه؟

امن (safe)

95 مصرفی
3 موجودی

PAPCO

Subject:
Year. Month. Date. ( )

باید ببینیم با این 2 تا می‌توانیم هیچ یک از این 3 فرایندها را به پایان برسانیم

با انتخاب P_2 به آخر می‌رسد.

↓

حالا که تمام شد موجودی 5

P_1 را با 5 تا می‌توان به آخر رساند انتخاب P_1 به آخر می‌رسد

موجودی 10

P_3 هم به آخر می‌رسد

این موجودی حالت مسیر پیدا کردیم یعنی اگر به این ترتیب اجرا کنیم به آخر می‌رسند

ترتیب امن <Safe Sequence> $\langle P_2, P_1, P_3 \rangle$

پس بله این حالت امن است.

ادامه مسئله: P_3 تقاضای یک جدید اضافه می‌کند آیا حالت جدید امن است؟

P_1	5
P_2	2
P_3	3

موجودی 2

انتخاب P_2 به آخر می‌رسد

موجودی 4 ← با این موجودی P_1 یا P_3 را نمی‌توان به آخر رساند ترتیب امن وجود ندارد

پس حالت جدید امن نیست

الگوریتم بانکداران می‌گوید اگر P_3 تقاضای یکی اضافه کرد چون حالت جدید امن نیست یعنی احتمال بن بست هست، این یکی اضافه را به او نمی‌دهیم و می‌گذاریم در حالت انتظار بماند

(خلاف ترتیب P_2 و P_1 و P_3 تقاضا کرده)

برای همین منبع به درد نخوری است چون منبع را از دست می‌ازدری ترس بدارد ①

از کجا می‌دانیم فرایند حداکثر چندتا از یک منبع می‌خواهد حتی اگر 100 بار تکرار

شده باشد اگر دوری فرق کند، این هم فرق دارد ② ⇐ مجبور می‌شویم تازه دست بالا هم بگیریم

Subject:
Year. Month. Date. ( )

جلسه بیست و یکم

الگوریتم بانکداران

n فرآیند

n منبع

بردار available[m] منبع j ام k تا وجود دارد — $Available[j] = k$

ماتریس $max_{[n,m]}$ ← هر سطرش max یک فرآیند

ماتریس allocation[n,m] ← سطر i ام مربوط به فرآیند i ام ← چه مقدار از منابع استفاده کرده

← ماتریس تخصیص

ماتریس Claim ← هر فرآیند چه مقدار نیاز دارد تا به max برسد — $claim = max - allocation$

این ساختمان داده هم به اندازه و هم به مقدار در تغییر است. فرآیندها ممکن است ایجاد شوند یا از بین بروند

می‌روند منبعی می‌خواهند چیزهایی می‌خواهیم چه کار می‌کنیم

فرض می‌کنیم $request_i$ رسیده می‌خواهیم چی کار کنیم؟

← یک بردار است

1. این چیزی است که چک می‌کنیم این است که غیرمنطقی نباشد — $Request_i \leq Claim_i$ — از max بیشتر نشه

اگر برقرار نبود ... در غیر این صورت خطاست

2. اگر $Request_i \leq Available$ ادامه وگرنه خطا ← منابع داریم بخشی بدهیم یا نه (منتظر ... ماند)

3. به حالت جدیدی می‌رویم ببینیم امن هست یا نه

حالت جدید ←

$Available = Available - Request_i$

$Allocation_i = Allocation_i + Request_i$

$claim_i = claim_i - Request_i$

اگر حالت جدید امن باشد تخصیص انجام می‌شود وگرنه انتظار

چه جوری بفهمیم امن هست؟

Subject:
Year. Month. Date. ( )

آزمایش حالت امن:

1- work = Available

برای i=1, ..., n : Finish [i] = false

2- اندیس i را طوری پیدا کنیم که (فرآیندی که با منابع فعلی بتوانیم آن را به انتها برسانیم)

Finish [i] = false

$claim_i \le work$

اگر پیدا نشد به مرحله 4 برو

3- work = work + allocation ; (چون منابع فرآیندی که به انتها رسیده آزاد می‌شود) (پس منابع آزاد بیشتر می‌شود)

اگر به این مرحله رسیدیم یعنی فرآیند

بین مراحل 2 و 3 انقدر می‌رویم تا بتوانیم i را پیدا کنیم

4- اگر برای i=1, ..., n ، finish[i] = True باشد سیستم امن است

اگر حالت امن نباشد بمعنای بن‌بست نیست بلکه در حالت انتظار است چون ممکن است به بن‌بست برسیم یعنی در حالت کل منابع را دارد ولی اجازه استفاده نمی‌دهد که ممکن است به بن‌بست برسد

مثال: فرآیندها $P_1 ... P_5$

منابع A B C

کل منابع سیستم 10 5 7

مقدار max هم جزو داده‌های مسئله است

	A	B	C
P_1	7	3	2
P_2	3	2	2
P_3	9	0	2
P_4	2	2	2
P_5	4	3	3

در لحظه t، سیستم را نگه داریم و allocation را این صورت ببینیم

Subject:

Year. Month. Date. ( )

	A	B	C
P_1	0	1	0
P_2	2	0	0
P_3	3	0	2
P_4	2	1	1
P_5	0	0	2
	7	2	5

← T برخط allocation

=> Available = 3 3 2

2 1 0

=> چیزی که مصرف شده

	A	B	C
P_1	7	2	5
P_2	1	2	2
P_3	6	0	0
P_4	0	1	1
P_5	4	3	1

max - allocation ← need (Claim) بترس

available 3 2 1

۱. آیا این حالت امن است؟ احتمال اینکه بن بست برسد هست یا نه

need را نگاه می‌کنیم ← P_2 و P_4 می‌توانند با این مقدار به آخر برسند

یک ترتیب امن پیدا کنیم لزوماً منحصر به فرد (unique) نیست در این جا P_2 را انتخاب می‌کنیم و اجرا می‌شود و در نتیجه P_2 موجودی خود را آزاد خواهد کرد پس داریم

Available = 5 3 2 ✓ 5 2 1

انتخاب P_4	7 4 3	1 4 3
انتخاب P_3	10 4 5	10 4 5
P_1		
P_5		

یک ترتیب امن پیدا شد پس حالت این است $\langle P_2, P_4, P_3, P_1, P_5 \rangle$

PAPCO

۴۹

Subject:
Year. Month. Date. ( )

2- Request(1) = 0 2 0 می‌رسد، آیا حالت جدید امن هست یا نه؟

allocation → need (Claim)

	A	B	C	A	B	C
P_1	0	3	0	7	0	5

available => 3 1 2

باید ببینیم این حالت امن هست یا نه.

در این جا P_2 را نمی‌توانیم انتخاب کنیم پس باید P_4 را انتخاب کنیم

انتخاب	P_4	5 2 3	available
"	P_2	7 2 3	"
"	P_3	10 2 5	"
	⋮		

این حالت هم امن هست ← اگر پیدا می‌کردیم امن نبود.

اگر جایی ترتیبی انتخاب کنیم باید در Safe Sequence باشد در غیر این صورت به حالت بن بست راه‌اندازی(؟) می‌رویم.

تشخیص بن بست Deadlock Detection

الگوریتم قبلی باید به ازای هر request باید اجرا بشود و خیلی وقت‌ها request باید اجرا شود که این کار وقت پردازنده را می‌گیرد
این الگوریتم هم زمان قابل اجراست و هر وقت اجرا کنیم می‌گوید بن بست داریم یا نه. مثلاً سر هر 50s که ببینیم کسی در بن بست هست یا نه. از اطلاعات موجود استفاده می‌کند
← خیلی شبیه الگوریتم قبلی است.

1- work = available

اگر allocation ≠ 0 ⇐ Finish[i] = false

وگرنه Finish[i] = True

از منابعی که شرط انحصار متقابل دارند استفاده می‌کند
از n تا فرایندی که داریم، آن‌هایی که allocation صفر دارند به حساب نمی‌آوریم
← یعنی هیچ منبعی ندارد

P4PCO

Subject:
Year. Month. Date. ( )

منابعی که بلاهای آن ها را انحصار متقابل دارند در بن بست دخالت دارند ← در اصل منابعی را حساب می کنیم که شرط انحصار متقابل نداشته باشند

2. i را که برای آن پیدا کن که Finish [i] = false ← یک چیزی را پیدا کن که منابعی که می خواهد از موجودی کمتر باشد
request i ≤ work

وگرنه برو مرحله 4

3. work = work + allocation i ← فرآیند را انتخاب می کنیم و به آن منابع می دهیم
Finish [i] = True

به مرحله 2 برو

4 - یا همه Finish ها True هست ← که بن بست نیست
اگر " " " نشوند ← بن بست هست
اگر Finish [i] = True برای i = 1, ..., n ← بن بست نداریم
این الگوریتم بی لینکی های در بن بست هست

← ممکن است request ها بن بست ببرد ولی با آن هیچ کاری نمی کنیم

مثال) P1 ... P5 فرآیندها
منابع ← A B C
7 2 6

می توان داشته باشیم که هیچ request نداریم

	allocation A	B	C	request A	B	C
P1	0	1	0	0	0	0
P2	2	0	0	2	0	2
P3	3	0	3	0	0	0
P4	2	1	1	1	0	0
P5	0	0	2	0	0	2
	7	2	6			

P1 ← request ندارد ← کاری نمی کند

available 0 0 0 ← هر چی داشتیم تحویل دادیم

Subject:
Year. Month. Date. ( )

در بن‌بست هستیم یا نیستیم؟

انتخاب P_1 یا P_3 چون هر دو صفر هستند

		A	B	C	
انتخاب	P_1 ←	0	1	0	available
"	P_3 ←	3	1	3	
"	P_2 ←	5	1	5	
"	P_4				
"	P_5				

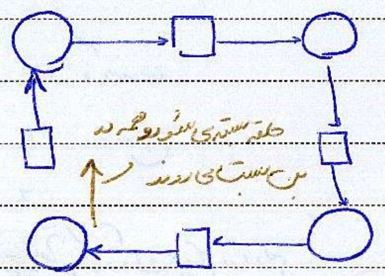
پس system در بن‌بست نیست، همگی True شدند

2_ $Request_3 = 0\ 0\ 1$ می‌رسد ← آیا سیستم تقاضای ! حمد از نیح چه دارند

ماتریس request در P_3 0 0 1

انتخاب P_1 0 1 0 ← بعد از این هیچ کدام را نمی‌توانیم به آخر برسانیم پس بن‌بست می‌رسیم

← P_2 و P_3 و P_4 و P_5 ← در بن‌بست هستند



← این حلقه بسته نشده پس بن‌بست نیست

می‌دانیم بن‌بست وجود دارد که کی در بن‌بست است می‌توانیم از این اطلاعات برای شکستن حلقه استفاده کنیم

← سیستم خلوت ← ready Queue ← خلوت‌تر است
" شلوغ ← " " ← دارای request زیادتر است

الگوریتم بن‌بست را کدام باید بیشتر اجرا کنیم؟
در سیستم خلوت، در جایی که شلوغ است اصلاً وقت ندارد پرسش را بخواند
← شاید این خلوت شدن بن‌بست باشد پس منطقی است که این جا بیشتر اجرا شود

PAPCO