دانشگاه صنعتی امیرکبیر (پلی تکنیک تهران)
دانشکدهٔ مهندسی کامپیوتر و فنآوری اطلاعات

اصول و مفاهیم سیستم عامل

ترم مهر ۹۸

فصل سوم: مدیریت فرآیند

(همروندی و همگام سازی)

نستوه طاهری جوان

nastooh@aut.ac.ir

فرآیندهای همکار

- ✓ فرآیندها در سیستم های چندبرنامه ای و اشتراک زمانی
  - یک، فرآیندهای مستقل
    - اجرای آنها تاثیری بر روی یکدیگر ندارد.
  - دو، فرآیندهای همکار
    - فرآیندها با هم همکاری دارند و اجرای آنها بر روی یکدیگر تاثیر دارد.

nastooh@aut.ac.ir

شرایط رقابتی

✓ مثال:

○ (فرض: مقدار اولیهٔ متغیرها برابر صفر است)

فرآیند P2

….
….
b=a;
c=a;
….
….

فرآیند P1
…. …. …. a=1; …. ….

○ حالت های مختلف اجرا:

- اگر ابتدا P2 و سپس P1 اجرا شود.
  b=0, c=0, a=1
- اگر ابتدا P2 و سپس P1 اجرا شود.
  b=1, c=1, a=1
- ابتدا P1 اجرا شود و بعد از اجرای دستور شماره (۲)، پردازنده به P1 سوئیچ کرده و در نهایت به P2 بازگردد.
  b=0, a=1, c=1

nastooh@aut.ac.ir

شرایط رقابتی

- **تعریف شرایط رقابتی(Race Condition):** شرایطی که دو یا چند فرآیند قصد دسترسی به یک منبع مشترک (مثلا حافظه) را دارند، به نحوی که حاصل کار آنها بر روی یکدیگر تاثیر گذارد.

- **تعریف ناحیه بحرانی(Critical Section):** در شرایطی که چند فرآیند قصد دسترسی به یک منبع مشترک را دارند، قطعه کدی که این منبع را دستکاری می کند ناحیه بحرانی نام دارد.

nastooh@aut.ac.ir

ناحیه بحرانی

- یک فرآیند سمبلیک در این بخش

```
while(TRUE)
{
دستورات عادی

دستورات ورود به ناحیه بحرانی

دستورات ناحیه بحرانی

دستورات خروج از ناحیه بحرانی

دستورات عادی
}
```

می توان برای حفاظت از ناحیه بحرانی
دستوراتی قبل و بعد از آن اضافه کرد

nastooh@aut.ac.ir

مقابله با شرایط رقابتی

- روش های مقابله با شرایط رقابتی

1. راه کارهای نرم افزاری (مستقل از سیستم عامل و سخت افزار)
2. راه کارهای سخت افزاری (نیاز به حمایت سخت افزار)
3. راه کارهای سیستم عامل (نیاز به حمایت سیستم عامل)
4. راه کارهای زبان های برنامه سازی (نیاز به حمایت زبان های برنامه سازی)
5. راه کارهای مبتنی بر تبادل پیام

nastooh@aut.ac.ir

شرایط راه کارها

✓ راه کارهای پیشنهادی باید دارای سه شرط زیر باشند:

1. انحصار متقابل (Mutual Exclusion)
   - در آن واحد فقط یک فرآیند داخل ناحیهٔ بحرانی خود فعالیت کند.
   - نام های دیگر: دو به دو ناسازگاری، مانعه الجمعی
2. پیشروی (Progress)
   - فرآیندی که **نه** در داخل ناحیه بحرانی اش و **نه** داوطلب ورود به ناحیه بحرانی اش است، نباید مانع ورود فرآیندهای دیگر به ناحیه بحرانیشان شود.
     - یعنی اگر هیچ مزاحمی نیست، یک فرآیند بتواند بارها وارد ناحیه بحرانی بشود.
3. انتظار محدود (Bounded Waiting)
   - یک فرآیند نباید به طور نامحدود منتظر ورود به ناحیه بحرانی اش بماند.
     - به عنوان مثال نباید در شرایطی فرآیندهای دیگر پیاپی بتوانند وارد ناحیه بحرانیشان شوند، اما یک فرآیند منع شود! به عبارت دیگر فرآیندها باید تعداد دفعات محدودی منتظر ورود به ناحیه بحرانیشان بمانند. یعنی اگر فرآیندی منتظر ورود است، یک فرآیند دیگر نباید بتواند بی نهایت وارد شود و فرآیند اول را منع کند.

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ روش اول: استفاده از متغییر قفل (Lock Variable)
  - استفاده از یک متغییر برای قفل کردن دیگر فرآیندها

P0

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

P1

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

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ نکات روش اول:
  - هر فرآیند قبل از ورود به ناحیه بحرانی باید متغییر قفل را چک کند.
    - اگر قفل برابر یک بود، یعنی فرآیند دیگری در ناحیه بحرانی اش است.
  - هر فرآیند قبل از ورود به ناحیه بحرانی باشد متغییر قفل را یک کند.
  - هر فرآیند پس از خروج از ناحیهٔ بحرانی باید قفل را صفر کند.
  - عیب:
    - عدم ارضای شرط انحصار متقابل
      - ➢ فرآیند اول قفل را آزاد می بیند و قصد ورود می کند، قبل از یک کردن قفل نوبت فرآیند دوم می شود، فرآیند دوم نیز قفل را آزاد می بیند. هر دو وارد ناحیه بحرانی می شوند.
    - عدم ارضای شرط انتظار محدود
      - ➢ یک فرآیند می تواند بدون در نظر گرفتن فرآیندهای رقیب، بی نهایت بار وارد ناحیه بحرانی اش شود.

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ روش دوم: تناوب قطعی
  - استفاده از یک متغییر سراسری به نام turn
    - متغییر turn در واقع نوبت ورود به ناحیهٔ بحرانی را نگه میدارد.

P0

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

P1

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

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ نکات روش دوم
  - اگر turn برابر صفر باشد، فرآیند P0 و اگر برابر یک باشد فرآیند P1 اجازه ورود به ناحیه بحرانی را دارد.
  - فرآیند مقابل، تا زمانی که مقدار turn را موافق نبیند، در حلقهٔ بینهایت خود می چرخد.
  - فرآیندها پس از خروج از ناحیهٔ بحرانی، نوبت را به دیگری می سپارند.
  - مزیت ها:
    - ارضای شرط انحصار متقابل
    - ارضای شرط انتظار محدود
      - ➢ چون نوبتی وارد ناحیه بحرانی می شوند، یک فرآیند دچار گرسنگی نمی شود.
  - عیب:
    - عدم ارضای شرط پیشرفت
      - ➢ فرض کنید یک فرآیند پس از خروج از ناحیه بحرانی، نوبت را برای دیگری سِت کرده است، اما فرآیند مقابل تصمیم به ورود به ناحیه بحرانی ندارد. حال فرآیند اول دوباره متقاضی ورود به ناحیه بحرانی است.
    - انتظار مشغول (Busy Waiting)

nastooh@aut.ac.ir

راه کارهای نرم افزاری

✓ نکته مهم پیرامون تفاوت شروط دوم و سوم

- شرط پیشرفت بیان می کند وقتی هیچ فرآیند دیگری در ناحیه بحرانی اش نیست و هیچ فرآیند دیگری نیز متقاضی ورود به ناحیه بحرانی اش نیست، فرآیند A باید بتواند وارد ناحیه بحرانی اش شود.
  - به عنوان مثال فرآیند A **اگر هیچ مزاحمی ندارد** باید بتواند بارها (پشت سر هم) وارد ناحیه بحرانی اش شود.
- شرط انتظار محدود بیان می کند فرآیندها **نباید** با احتمال گرسنگی مواجه شوند. یعنی نباید فرآیند A بخواهد وارد ناحیه بحرانی اش شود، اما فرآیند B در همین حین بارها و بارها وارد ناحیه بحرانی اش شده و خارج شود.
  - در واقع باید به نوعی نوبت رعایت شود.
  - مثلا اگر فرآیند A درخواست ورود به ناحیه بحرانی اش را داد، باید بتواند بعد از تعداد محدود و مشخصی **انتظار**، وارد ناحیه بحرانی اش شود. (این احتمال وجود نداشته باشد که فرآیند دیگری در حین انتظار A بارها و بارها به ناحیه بحرانی وارد شود)
  - به عبارت دیگر: **نباید** فرآیند B پس از خروج از ناحیهٔ بحرانی اش، بدون توجه به درخواست های فرآیندهای رقیب (مثلا فرآیند A) دوباره وارد ناحیه بحرانی شود.

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ روش سوم
  - استفاده از دو متغییر سراسری جداگانه با مقدار اولیه false

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;
……..
}
```


nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ نکات روش سوم:
  - اصلاح روش دوم برای ارضای شرط پیشرفت
  - مزایا:
    - ارضای شرط پیشرفت
    - نیازی به اجرای یک درمیان ناحیه بحرانی نیست.
  - عیب:
    - عدم ارضای شرط انحصار متقابل
      - ➢ هر دو فرایند همزمان شرط ورود (متغییر فرآیند مقابل) را چک کنند.
    - عدم ارضای شرط انتظار محدود
      - ➢ یک فرآیند ممکن است بطور متناوب و هر زمان پردازنده را در اختیار گرفت، وارد ناحیه بحرانی اش شود و در این حین اگر پردازنده به فرآیند مقابل داده شود، آن فرآیند منتظر ورود به ناحیه بحرانی اش می ماند. (شاید تا ابد!)
    - انتظار مشغول

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ روش چهارم
  - جابجایی دو دستور ورود به ناحیه بحرانی

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;
……..
}
```


nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ نکات روش چهارم
  - فرآیند ابتدا متغییر خود را یک می کند، بعد طرف مقابل را چک می کند.
  - مزایا:
    - ارضای شرط انحصار متقابل
    - ارضای شرط پیشرفت
  - عیب
    - عدم ارضای شرط انتظار محدود
      - ➢ هر دو فرآیند ممکن است نتوانند وارد ناحیه بحرانی شوند و تا ابد در حلقه خود می مانند.
    - انتظار مشغول

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- روش پنجم: راه کار Peterson

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;
……..
}
```

- همهٔ حالت های ممکن برای اجرا را تست کنید!

nastooh@aut.ac.ir

راه کارهای نرم افزاری

- ✓ نکات روش پنجم:
  - مزایا:
    - ارضای شرط انحصار متقابل
    - ارضای شرط پیشرفت
    - ارضای شرط انتظار محدود
  - عیب
    - انتظار مشغول

راه کارهای نیازمند به حمایت سخت افزار

- ✓ روش اول: از کار انداختن وقفه ها
  - اعطای قدرت غیر فعال (و فعال) کردن وقفه ها به فرآیندها!
  - فرآیندها قبل از ورود به ناحیه بحرانیشان، وقفه ها را غیرفعال کنند، تا پردازنده از آنها پس گرفته نشود و مطمئن شوند فرآیند دیگری وارد ناحیه بحرانی (و غیر بحرانی!)اش نمی شود و پس از خروج از ناحیه بحرانی دوباره وقفه ها را فعال کند.
  - معایب:
    - اعطای قدرت غیر فعال کردن وقفه ها به فرآیندها عاقلانه نیست. زیرا فرآیندی ممکن است فراموش کند وقفه ها را دوباره فعال کند.
    - در سیستم های چند پردازنده ای کارا نیست. (فقط وقفه های پردازندهٔ جاری غیر فعال می شوند و پردازنده های دیگر کماکان فعال هستند.)

nastooh@aut.ac.ir

راه کارهای نیازمند به حمایت سخت افزار

- ✓ روش دوم: استفاده از دستور TSL (Test & Set Lock)
  - اضافه کردن یک دستور اتمیک (و غیر قابل تجزیه) برای پردازنده
    - این دستور در واقع یک فلگ را ابتدا چک کرده و سپس آن را سِت می کند.
    - می توان از این دستور به همراه متغییر قفل استفاده کرد.
    - در واقع دستور TSL با نمایش شبه کد به صورت زیر پیاده می شود:

```
Boolean TSL (Boolean i)
{
  if (i==false)
     {
        i=true;
        return true
     }
  else
      return false;
}
```


nastooh@aut.ac.ir

راه کارهای نیازمند به حمایت سخت افزار

- ✓ نحوهٔ استفاده از دستور اتمیک TSL
  - مقدار اولیهٔ lock برابر false است

P

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

- معایب:
  - عدم ارضای شرط انتظار محدود
    - یک فرآیند پس از خروج از ناحیه بحرانی و بدون توجه به درخواست فرآیندهای رقیب، می تواند دوباره به ناحیه بحرانی وارد شود.
  - انتظار مشغول

nastooh@aut.ac.ir

راه کارهای نیازمند به حمایت سخت افزار

- ✓ روش سوم: استفاده از دستور اتمیک Swap
  - ○ این دستور به صورت غیر قابل تجزیه مقدار دو مکان از حافظه را تعویض می کند.
    - می توان از این دستور به همراه متغییر قفل استفاده کرد.
    - در واقع دستور Swap با نمایش شبه کد به صورت زیر پیاده می شود:

```
void SWAP(int i, int j)
{
   int temp;
   temp=i;
   i=j;
   j=temp;
}
```


nastooh@aut.ac.ir

راه کارهای نیازمند به حمایت سخت افزار

✓ نحوهٔ استفاده از دستور اتمیک SWAP

- مقدار اولیهٔ lock برابر false است
- به راحتی برای چند فرآیند و حتی چند پردازنده اجرا می شود

Pi

```
while (true)
{
……..
flag[i]=true;
while(flag[i]==true)
   SWAP(flag[i], lock);
Critical Section;
lock=false;
……..
}
```

○ معایب:

- عدم ارضای شرط انتظار محدود
  - ➢ یک فرآیند پس از خروج از ناحیه بحرانی و بدون توجه به درخواست فرآیندهای رقیب، می تواند دوباره به ناحیه بحرانی وارد شود.
- انتظار مشغول

nastooh@aut.ac.ir

راه حل های نیازمند به حمایت سیستم عامل

- ✓ استفاده از سمافورها
  - ○ در واقع سمافور یک متغییر صحیح است.
    - دستیابی به متغییر سمافور توسط دستورات عادی امکان پذیر نیست.
    - بر روی سمافور دو عمل زیر قابل انجام است: **(این اعمال اتمیک هستند)**
      - ➢ Wait: مقدار سمافور را بررسی می کند. اگر برابر صفر بود، فرایند فراخواننده به وضعیت Waitمی رود. اما اگر مقدار آن غیر صفر بود، فقط یک واحد از آن کم می شود.
      - ➢ Signal: اگر فرآیند (یا فرآیندهایی) قبلا به حالت Wait رفته باشند، یکی از آنها آزاد می شود. در غیر اینصورت یک واحد به سمافور افزوده می شود.
    - معمولا سمافورها با مقادیر غیر منفی مقداردهی اولیه می شوند.
    - در کتب مختلف به جای wait ممکن است از down یا P و به جای signal از up یا V استفاده کنند.

nastooh@aut.ac.ir

راه حل های نیازمند به حمایت سیستم عامل

- یکی از روش های پیاده سازی سادهٔ Wait و Signal

wait (s)
while (s<=0); s--;

signal (s)
s++;

- در این حالت، وضعیت wait به صورت انتظار مشغول با حلقهٔ while پیاده شده است.
- همینکه در signal مقدار سمافور s یکی افزایش میابد، فرآیند wait شده از انتظار (اینجا حلقه) خارج می شود.
- منظور از **اتمیک** بودن wait این است که اگر احتمالا یک بار شرط حلقهٔ while برابر false بود، دستور ;s-- بعدی به همراه آن اجرا می شود.

nastooh@aut.ac.ir

راه حل های نیازمند به حمایت سیستم عامل

- ✓ استفاده از سمافورها
  1. برای همگام سازی اجرای فرآیندها
     - تعیین ترتیب خاص برای اجرای دستورات فرآیندهای همروند
  2. برای حل مشکل نواحی بحرانی

nastooh@aut.ac.ir

راه حل های سیستم عامل (سمافور ساده)

- ✓ استفاده از سمافور برای همگام سازی فرآیندها
  - فرض کنید فرآیند P1 و P2 به صورت همروند اجرا می شوند. اما می خواهیم دستور S1 در فرآیند P1 حتماً قبل از دستور S2 در فرآیند P2 اجرا شود. (به هر دلیلی)
    - راه حل: استفاده از یک سمافور با مقدار اولیهٔ صفر به صورت زیر

P1
….. S1 signal(sem) …..

P2
…. wait (sem) S2 ….

nastooh@aut.ac.ir

راه حل های سیستم عامل (سمافور ساده)

- ✓ استفاده از سمافور برای حل مشکل نواحی بحرانی
  - فرض کنیم n فرآیند مشکل ناحیه بحرانی دارند.
  - از یک سمافور با مقدار اولیهٔ 1 استفاده می کنیم.

P

```
while (true)
{
……..
wait (mutex)
Critical Section;
signal (mutex)
……..
}
```


nastooh@aut.ac.ir

راه حل های سیستم عامل (سمافور ساده)

- ✓ مشکل سمافورهای سادهٔ پیشین
  - ○ انتظار مشغول
    - مانند روش های نرم افزاری و سخت افزاری، فرآیندهایی که اجازهٔ ورودی به ناحیهٔ بحرانی ندارند، در یک حلقهٔ بی حاصل، برش زمانی خود (و وقت پردازنده) را تلف می کنند.
  - ○ عدم ارضای شرط انتظار محدود
    - فرآیندی که در ناحیه بحرانی اش فعالیت میکند، بدون توجه به دیگر فرآیندها ممکن است بتواند بارها به ناحیه بحرانی اش وارد شود.

- ✓ راه کار: توسعهٔ تعریف و پیاده سازی سمافورها
  - با استفاده از مفاهیمی مانند صف و بلوکه کردن فرآیندها

nastooh@aut.ac.ir

راه حل های سیستم عامل (تکمیل سمافور)

- ✓ تکمیل تعریف سمافورها
  - فرآیندی که اجازه ورود به ناحیه بحرانی اش را ندارد باید بلوکه شود.
    - به جای آنکه در یک حلقه بی نهایت وقت پردازنده را تلف کند.
  - فرآیند دیگری به موقع فرآیند بلوکه شده را بیدار می کند.
  - هر سمافور یک صف دارد که در آن لیست فرآیندهایی که بلاک شده اند قرار دارد.
  - تعریف کامل تر سمافور به صورت یک رکورد:

```
struct semaphore
{
   int count;
   queue type List;
}
```


nastooh@aut.ac.ir

راه حل های سیستم عامل (تکمیل سمافور)

✓ تکمیل اعمال wait و signal (باز هم اتمیک هستند)

wait (s)

```
if (s.count>0)
   s.count--;
else
  {
     place this process in s.List
     Block this process
  }
```

signal (s)

```
if (s.List is empty)
   s.count++;
else
   {
     Remove a process from s.List
     Place that process in Ready list
   }
```

- فرآیندی که قصد ورود به ناحیه بحرانی را دارد، wait می کند. اگر شمارنده یک بود، یعنی می تواند وارد ناحیه بحرانی شود، پس قبل از ورود شمارنده را صفر می کند. اما اگر شمارنده صفر بود، یعنی نمی تواند وارد شود، پس خود را بلاک می کند.
- فرآیندی که قصد خروج از ناحیه بحرانی را دارد، اول لیست را چک میکند، اگر لیست خالی بود که شمارنده را یک می کند تا فرآیند دیگری بتواند وارد شود. اما اگر لیست پر بود، یک فرآیند از لیست کم کرده و آن فرآیند را بیدار می کند.
- فرآیندی که بیدار شود، از دستور بعد از بلاک شدن اجرا می شود. یعنی وارد ناحیه بحرانی می شود.

nastooh@aut.ac.ir

راه حل های سیستم عامل (تکمیل سمافور)

- ✓ بررسی سمافور ها برای حل مشکل نواحی بحرانی
  - ○ مزیت ها
    - ارضای شرط انحصار متقابل
    - ارضای شرط پیشروی
    - ارضای شرط انتظار محدود

nastooh@aut.ac.ir

راه حل های نیازمند به حمایت سیستم عامل

- ✓ انواع سمافورها
  - ○ سمافورهای باینری
    - سمافور فقط دو مقدار می گیرد.
    - مورد استفاده: همگام سازی فرآیندها و مراقبت از نواحی بحرانی
  - ○ سمافورهای شمارشی (جنرال)
    - سمافور می تواند هر مقداری داشته باشد.
    - مورد استفاده: مدیریت چند نمونه از یک منبع خاص. (مثلا ۷ عدد نوارخوان داریم که تعداد زیادی فرآیند متقاضی استفاده از آنها هستند.)

nastooh@aut.ac.ir

مسائل کلاسیک IPC

- برای بررسی راه حال های مشکلات اجرای فرآیندهای همکار، مسائل کلاسیکی تعریف شده اند و هر راه حل باید برای رفع مشکلات مسائل کلاسیک بررسی شود.

1. مسالهٔ کلاسیک تولید کننده – مصرف کننده
2. مسالهٔ کلاسیک خوانندگان و نویسندگان
3. مسالهٔ کلاسیک فیلسوفان خورنده (غذا خوردن فیلسوف ها)
4. مسالهٔ کلاسیک آرایشگر خواب آلود

nastooh@aut.ac.ir

تعریف مساله کلاسیک تولیدکننده مصرف کننده

- ✓ دو فرآیند جداگانه، هر کدام در یک حلقه اجرا می شوند.
- ✓ یک فرآیند قلم اطلاعاتی تولید می کند و یک فرآیند مصرف می کند.
- ✓ فرآیندها یک بافر اشتراکی با طول n استفاده می کنند.
- ✓ فرآیند تولید کننده، هر بار که یک قلم اطلاعاتی تولید می کند، به شرط آنکه بافر پر نباشد، آن را به بافر اضافه می کند.
  - وقتی بافر پر است، فرآیند تولید کننده دیگر نباید داده ای تولید کند، تا مصرف کننده قدری بافر را خالی کند.
- ✓ فرآیند مصرف کننده به شرط خالی نبودن بافر، یک قلم اطلاعاتی از بافر برمی دارد.
  - وقتی بافر خالی است مصرف کننده نمی تواند کاری کند تا تولید کننده قدری داده تولید کند.

nastooh@aut.ac.ir

تعریف مساله کلاسیک خوانندگان و نویسندگان

- چند فرآیند به عنوان خواننده همزمان می توانند بانک اطلاعاتی را بخوانند.
  - عمل خواندن فقط توسط خوانندگان رخ می دهد.
- اگر یک فرآیند نویسنده در حال تغییر بانک اطلاعاتی باشد، فرآیندهای دیگر (چه نویسنده چه خواننده) نباید به بانک اطلاعاتی دسترسی داشته باشند.
- این مساله می تواند به چهار صورت بررسی شود:
  - خوانندگان اولویت دارند.
  - خوانندگان اولویت **مطلق** دارند.
  - نویسندگان اولویت دارند.
  - بدون اولویت خاص.

nastooh@aut.ac.ir

تعریف مساله کلاسیک فیلسوفان خورنده

- ✓ پنج فیلسوف دور یک میز دایره ای نشسته اند.
- ✓ برای هر فیلسوف یک بشقاب اسپاگتی وجود دارد.
- ✓ بین هر دو بشقاب یک چنگال برای استفادهٔ فیلسوف ها هست.
- ✓ بشقاب ها همیشه اسپاگتی دارند!
- ✓ هر فیلسوف برای خوردن اسپاگتی به دو چنگال نیاز دارد.
- ✓ زندگی فیلسوف ها محدود به خوردن و فکر کردن است.
- ✓ فیلسوفی که گرسنه می شود سعی می کند چنگالها را بر دارد.
- ✓ هر فیلسوف در صورت موفقیت به شروع به خورن و بعد از سیر شدن، چنگالها را روی میز قرار می دهد.
- ✓ بن بست در زندگی فیلسوف ها نباید رخ دهد.

nastooh@aut.ac.ir

تعریف مساله کلاسیک آرایشگر خواب آلود

- یک آرایشگاه مردانه دارای **یک** صندلی آرایشگری و n صندلی انتظار است.
- اگر هیچ مشتری ای در آرایشگاه نباشد، آرایشگر بر روی صندلی اصلی به خواب می رود.
- اگر مشتری وارد شود، آرایشگر بیدار شده و مشتری بر روی صندلی اصلی می نشیند تا اصلاح شود.
- مشتریان بعدی بر روی صندلی های انتظار می نشینند.
- اگر همه صندلی ها پر باشند، مشتری جدید از آرایشگاه خارج می شود.

nastooh@aut.ac.ir

حل مساله کلاسیک تولید کننده و مصرف کننده به کمک سمافورها

- ✓ مساله تولید کننده و مصرف کننده
  - استفاده از ۳ سمافور.
    - full از نوع شمارشی و با مقدار اولیهٔ صفر
    - empty از نوع شمارشی و با مقدار اولیهٔ n
    - mutex از نوع باینری و با مقدار اولیهٔ یک
  - دو سمافور شمارشی برای همگام سازی تولیدکننده و مصرف کننده برای استفاده از بافر است و سمافور باینری برای انحصار متقابل.

nastooh@aut.ac.ir

حل مساله کلاسیک تولید کننده و مصرف کننده به کمک سمافورها

✓ مساله تولید کننده و مصرف کننده

producer()

```
while (true)
{
    produce_item (item);
    wait(empty);
    wait(mutex);
    enter_item (item);//C.S.
    signal(mutex);
    signal(full);
}
```

consumer()

```
while (true)
{
    wait(full);
    wait(mutex);
    remove_item (item);//C.S.
    signal(mutex);
    signal(empty);
    consume_item(item);
}
```

- empty خانه های خالی بافر را شمارش می کند و ممکن است تولید کننده را به خواب بفرستد.
- full خانه های پر بافر را شمارش می کند و ممکن است مصرف کننده را به خواب بفرستد.

nastooh@aut.ac.ir

حل مساله کلاسیک خوانندگان و نویسندگان به کمک سمافورها

- ✓ مسالهٔ خوانندگان و نویسندگان (با اولویت خوانندگان)
  - منظور از اولویت خوانندگان این است که مادامی که یک خواننده مشغول فعالیت است، خوانندگان جدید می توانند وارد شوند. (حتی اگر نویسنده ای منتظر است.)
  - به دلیل اولویت خوانندگان، فقط خوانندهٔ اول نویسندگان را به خواب می فرستند و خوانندگان بعدی نیازی به این کار ندارند.
  - یک متغیر با مقدار اولیهٔ صفر با عنوان ReaderCount تعداد خوانندگان را شمارش می کند.
  - یک سمافور باینری با مقدار اولیهٔ یک باید انحصار متقابل را تضمین کند.
  - یک سمافور باینری با مقدار اولیهٔ یک و نام write ورود نویسندگان را حفاظت می کند. (حداکثر یک نویسنده)

nastooh@aut.ac.ir

حل مساله کلاسیک خوانندگان و نویسندگان به کمک سمافورها

✓ مسالهٔ خوانندگان و نویسندگان (با اولویت خوانندگان)

readers()

```
while (true)
{
    wait(mutex);
    ReaderCount++;
    if(ReaderCount==1)
        wait(write);
    signal(mutex);
    Read from Source;
    wait(mutex);
    ReaderCount--;
    if(ReaderCount==0)
        signal(write);
    signal(mutex);
}
```

writers()

```
while (true)
{
wait(write);
Writing to Source;
signal(write);
}
```

- در این حالت منظور از اولویت خوانندگان این است که مادامی که یک خواننده وجود داشته باشد، خوانندگان جدید می توانند وارد شوند.

nastooh@aut.ac.ir

حل مساله کلاسیک خوانندگان و نویسندگان به کمک سمافورها

✓ مسالهٔ خوانندگان و نویسندگان (با اولویت **مطلق** خوانندگان)

- اگر در حین نوشتن یک نویسنده، صفی از نویسندگان در سیستم موجود باشد و خواننده (یا خواننده هایی) از راه برسد، اولویت این خواننده از تمام نویسندگان منتظر بیشتر است. حتی اگر نویسنده های منتظر، زودتر وارد شده باشند.
  - دقت کنید در هر حالت، هنگامی که یک نویسنده در حال نوشتن است، خوانندگان باید منتظر اتمام کار وی باشند.

writers()

```
while (true)
{
wait(Q);
wait(write);
Writing to Source;
signal(write);
signal(Q);
}
```

  - از وقوع صف نویسندگان بر روی سمافور write جلوگیری میکنیم. در واقع اگر نویسنده ای در سیستم موجود است، پشت سمافور Q می ماند و اجازهٔ ورود به صف write را ندارد.

nastooh@aut.ac.ir

یک مثال از استفاده از سمافور در حل مساله

- مثال: فرض کنید دو نوع فرآیند در سیستم موجود هستند. نوع A و نوع B. قوانین زیر موجودند.
  - فرآیندهای نوع A می توانند پشت سر هم وارد ناحیه بحرانی شان شوند.
  - به محض اینکه یک فرآیند از نوع B وارد ناحیه بحرانی اش شد، دیگر هیچ فرآیند جدیدی **نتواند** وارد ناحیه بحرانی شود.

A()

while (true)
{
wait(S);
Signal(S);
C.S.
}

B()

while (true)
{
wait(S);
C.S.
Signal(S);
}

« مقدار اولیه سمافور S برابر یک است.

nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

- ✓ مسالهٔ فیلسوف های خورنده
  - ○ حالت اول:
    - هر فیلسوف ابتدا چنگال سمت راست خود را بر میدارد.
    - اگر چنگال سمت راست را برداشت، سراغ چنگال سمت چپ می رود.
    - اگر هر یک از چنگال ها مشغول بودند، منتظر می ماند.
    - برای هر یک از چنگالها نیاز به یک سمافور باینری با مقدار یک داریم.
    - سمافور fork(i) مربوط به چنگال شماره i است.

nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

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

- عیب بزرگ: همهٔ فیلسوف ها ممکن است چنگال سمت چپ خود را برداشته و بر روی چنگال سمت راست خود wait شوند و بن بست رخ دهد.

nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

- حالت دوم
  - تقسیم فیلسوف ها به دودستهٔ راست دست و چپ دست.
  - گروه راست دست، اول چنگال سمت راست خود را برمی دارند و گروه چپ دست اول سعی میکنند چنگال سمت چپ خود را بردارند.
  - **فرض مهم:** دور هر میز حداقل یک فیلسوف چپ دست و یک فیلسوف راست دست وجود دارد!

چپ دستان

```
while (true)
{
    think();
    wait(fork[(i+1)%5]);
    wait(fork[i]);
    eat();
    signal(fork[(i+1)%5]);
    signal(fork[i]);
}
```

راست دستان

```
while (true)
{
    think();
    wait(fork[i]);
    wait(fork[(i+1)%5]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1)%5]);
}
```


nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

- حالت سوم
  - فرض مهم: فقط به چهار فیلسوف اجازهٔ رقابت می دهیم!
  - یک سمافور شمارشی با مقدار اولیهٔ ۴ نیاز داریم. (مثلا با عنوان table)
  - عیب: ایجاد محدودیت در مساله.

void philosephor(int i)

```
while (true)
{
    think();
    wait(table);
    wait(fork[i]);
    wait(fork[(i+1)%5]);
    eat();
    signal(fork(i));
    signal(fork((i+1)%5));
    signal(table);
}
```


nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

حالت چهارم:

- هرفیلسوف سعی می کند شروع به خوردن کند.
  - ابتدا باید وضعیت همسایه های کناری را چک کند. اگر همسایه ها در حال خوردن باشند به این معنی است که چنگال ها آزاد نیستند.
  - اگر مزاحمی نداشت، وضعیت خود را به eating تغییر می دهد و شروع می کند.
- اگر یک فیلسوف گرسنه شد، اما کنار یک فیلسوف در حال خوردن بود، نمی تواند شروع به خوردن کند.
  - پس خود را بلوک می کند و بر روی سمافور خودش wait می شود.
- فیلسوفی که خوردنش به پایان رسید، با خبر دادن به هر دو همسایه ی خود، آنها را چک می کند.
  - در این حالت، اگر فیلسوف های کناری در وضعیت گرسنه باشند، چک می کنند که آیا می توانند غذا بخورند یا خیر.
  - اگر فیلسوف های کناری در وضعیت فکر کردن باشند، که اتفاقی نمی افتد.
  - در این حالت فیلسوف های کناری نمی توانند در وضعیت خورن باشند! **چرا؟**

nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

- نکات حالت چهارم:

- در این راهکار مفهوم سمافور چنگال که در راهکارهای قبلی استفاده شده بود، استفاده نشده است و به جای آن از سمافور فیلسوف استفاده شده است. (منطق روش ها کاملا متفاوت است!)
- به هر فیلسوف یک سمافور باینری با مقدار اولیهٔ صفر نسبت می دهیم(ph[i])
- برای هر فیلسوف یک متغییر برای نشان دادن وضعیتش نسبت می دهیم (state[i])
  - این متغییر فقط سه مقدار برای وضعیت هر فیلسوف می گیرد: مقادیر thinking و hungry و eating که مقدار اولیه آنها «فکر کردن» است.
- از یک سمافور باینری برای انحصار متقابل استفاده می کنیم.
- فیلسوفی که امکان دریافت دو چنگال را ندارد، متوقف می شود.
  - بر روی سمافور خودش wait می شود.

nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

void philosephor(int i)

```
while (true)
{
    think();
    take_forks(i);
    eat();
    put_down_forks(i);
}
```

void take_forks(int i)

```
wait(mutext);
state[i]=hungry;
test(i);
signal(mutext);
wait(ph[i]);
```

void put_down_forks(int i)

```
wait(mutext);
state[i]=thinking;
test(LEFT);
test(Right);
signal(mutext);
```

void test(int i)

```
if (state[i]==hungry       &&
    state[LEFT]!=eating  &&
    state[RIGHT]!=eating)
       {
           state[i]=eating;
           signal(ph[i]);
        }
```


nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

- ✓ توضیحات قطعه کدهای صفحه قبل:
  - آرایه ای از سمافورها با مقدار اولیه صفر و با نام ph[i] برای هر فیلسوف داریم.
  - آرایه ای از وضعیت ها با مقدار اولیه فکر کردن و با نام state[i] برای هر فیلسوف داریم.
  - مقدار LEFT در واقع برابر (i-1)%5 است. (همسایه چپ)
  - مقدار RIGHT در واقع برابر (i+1)%5 است. (همسایه راست)

nastooh@aut.ac.ir

حل مساله کلاسیک فیلسوف های خورنده به کمک سمافورها

- ✓ نکات حالت چهارم:
  - وقتی وضعیت فیلسوفی به eating تغییر می کند، دیگر می تواند با هر ترتیبی چنگالها را در روال ()eat برداشته و شروع به خوردن کند.
    - در هر مرحله ای از برداشتن چنگالها اگر تعویض متن رخ دهد، هیچ مشکلی رخ نمی دهد. چک کنید!
  - روالهای چک کردن وضعیت ها و سِت کردن وضعیت ها به عنوان نواحی بحرانی باید با انحصار متقابل اجرا شوند. (در واقع هنگام کار کردن با متغیرهای State) به همین دلیل از یک سمافور باینری برای محافظت آن ها استفاده می کنیم.

nastooh@aut.ac.ir

راه حل های نیازمند به حمایت زبان های برنامه سازی

✓ مانیتور

- یک زیرساخت نرم افزاری سطح بالا، شامل مجموعه ای از رویه ها، متغییرها و ساختمان داده ها (در یک قالب نرم افزاری در سطح کامپایلر)
  - داده های داخل مانیتور فقط توسط رویه های داخل مانیتور قابل دسترسی هستند.
  - فرآیندها می توانند رویه های داخل مانیتور را فراخوانی کنند.
  - در یک لحظه فقط و فقط یک فرآیند می تواند داخل مانیتور **فعال** باشد.
    - مابقی فرآیندها می توانند در صف ورود به مانیتور باشند یا در داخل مانیتور خواب باشند.
    - در واقع با فراخوانی یک رویه از مانیتور توسط یک فرآیند، مانیتور بررسی می کند که آیا فرآیند دیگری در داخل مانیتور فعال هست یا خیر؟ اگر فرآیند دیگری فعال باشد، این فرآیند جدید در صف ورود به مانیتور قرار می گیرد.
    - در واقع دیگر برنامه نویس مسئول شرط انحصار متقابل **نیست**.
  - برنامه نویس باید منابع بحرانی خود را در مانیتور قرار دهد.

nastooh@aut.ac.ir

راه حل های نیازمند به حمایت زبان های برنامه سازی

- ✓ نکات مهم در مورد مانیتورها
  - ○ استفاده از مانیتورها در یک زبان برنامه سازی باید پشتیبانی شود.
    - عملا زبان های برنامه سازی معروف و متداول، مانیتور را پشتیبانی **نمی کنند**. (یا لااقل تا سالها پشتیبانی نمی کردند.)
    - برخی از زبانهایی که از مانیتور پشتیبانی می کنند:
      - ➢ ADA (در واقع از ADA95 به بعد)
      - ➢ دلفی (در واقع از سال ۲۰۱۰ به بعد)
      - ➢ نسخه های اخیر زبان های سکوی دات نت.
      - ➢ C++ (در واقع از نسخهٔ C++11 به بعد)
  - ○ کامپایلرها برای پیاده سازی مانیتور در عمل از سمافورها استفادهٔ موثری می کنند.
    - بنابراین سیستم عامل باید سمافور را پشتیبانی کند. (رجوع شود به تمرین مربوط به **متغیرهای شرطی**)

nastooh@aut.ac.ir

راه حل های نیازمند به حمایت زبان های برنامه سازی

✓ شبه کد تعریف مانیتور

Monitor Sample

```
monitor  monitor_name
    int i,j;
    condvar x,y;
    تعریف سایر داده های مشترک....

  void proc1()
  {
  ......
  }

  void proc2()
  {
  .......
  }

End of Monitor
```

- فرآیندها می توانند رویه های داخل مانیتور را فراخوانی کنند.
- داده های تعریف شده در داخل مانیتور، فقط توسط رویه های مانیتور قابل دسترسی هستند.
- برنامه نویس فقط باید نواحی بحرانی خود را در داخل مانیتور قرار دهد

nastooh@aut.ac.ir

استفاده از تبادل پیام

- مناسب هم برای سیستم های متمرکز و هم سیستم های توزیع شده
- استفاده از دو فراخوان سیستمی در سطح سیستم عامل
  - Send
  - Receive
- استفاده از تبادل پیام در سیستم های توزیع شده مشکلات متعددی در پی دارد. مانند گم شدن پیام ها یا نیاز به احراز هویت و ...

nastooh@aut.ac.ir

استفاده از تبادل پیام

✓ ارتباط بین فرآیندها به دو صورت وجود دارد

- مستقیم
  - ارسال مستقیم از فرستنده به گیرنده. در این حالت فرستنده و گیرنده دقیقا طرف مقابل را مشخص می کنند.
  - توابع ارسال و دریافت به فرمت زیر استفاده می شوند
    - Send(destination, msg)
    - Receive(source, msg)
- غیرمستقیم
  - از یک جعبه پیام استفاده می شود. پیام ها در یک صف و در این جعبه پیام نگهداری می شوند. فرستنده با دستور ارسال و مشخص کردن جعبه پیام، یک پیام برای صندوق ارسال می کند و گیرنده با دستور دریافت و مشخص کردن جعبه پیام، یک پیام از صندوق برمی دارد.
  - توابع ارسال و دریافت به فرمت زیر استفاده می شوند
    - Send(MailBox, msg)
    - Receive(MailBox, msg)

nastooh@aut.ac.ir

استفاده از تبادل پیام

- ✓ استفاده از دستور Send می تواند به دو صورت پیاده سازی شود:
  - ○ مسدود شونده
    - • فرستنده با اجرای دستور send تا زمان دریافت پیام (توسط مقصد!) مسدود می شود.
  - ○ مسدود نشونده
    - • فرستنده پس از اجرای دستور send به کار خود ادامه می دهد.
- ✓ استفاده از دستور Receive می تواند به دو صورت پیاده سازی شود:
  - ○ مسدود شونده
    - • گیرنده بعد از اجرای دستور receive تا زمان دریافت پیام مسدود می شود.
  - ○ مسدود نشونده
    - • گیرنده پس از اجرای دستور receive به کار خود ادامه می دهد.

nastooh@aut.ac.ir

استفاده از تبادل پیام

- ✓ معمولا عمل ارسال را از نوع مسدود نشونده و عمل دریافت را از نوع مسدود شونده پیاده سازی می کنند.
- ✓ مثال: ایجاد انحصار متقابل
  - ○ فرض ها
    - عملیات ارسال مسدود نشونده است
    - عملیات دریافت مسدود شونده است
    - در برنامهٔ مادر و قبل از شروع به کار فرآیندهای همکار، یک پیغام تهی به میل باکس ارسال شده است

مثلا با دستوری شبیه به زیر:

send(MB, null);

```
void p(i)
while (true)
{
….
receive(MB, msg);
C.S.
send(MB, msg);
…..
}
```


nastooh@aut.ac.ir

تمرین ها

- تمرین ۱: راه کار Dekker برای مقابله با شرایط رقابتی را **به طور دقیق** بررسی کنید. (هم بررسی منطق آن و هم ارضای شروط)
  - راهنمایی: مرجع [3]، بخش 1-5.

P0

```
while (true)
{
flag[0]=true;
While(flag[1]==true)
  if(turn==1)
    {
       Flag[0]=false;
       While(turn==1);
       flag[0]=true;
    }
Critical Section;
turn=1;
flag[0]=false;
}
```

P1

```
while (true)
{
flag[1]=true;
While(flag[0]==true)
  if(turn==0)
    {
       Flag[1]=false;
       While(turn==0);
       flag[1]=true;
    }
Critical Section;
turn=0;
flag[1]=false;
}
```


nastooh@aut.ac.ir

تمرین ها

- ✓ تمرین ۲: مزایای برنامه سازی چندسرنخی را بیان کنید.
  - مرجع [1]، بخش 2-1-4
- ✓ تمرین ۳: تفاوت بین سمافور قوی و سمافور ضعیف را بیان کنید. هر کدام چه مزایا و معایبی دارد؟
  - مرجع [3] بخش 4-5
- ✓ تمرین ۴: دربارهٔ انواع موازی سازی تحقیق کنید.
  - مرجع [1]، بخش 2-2-4.

nastooh@aut.ac.ir

تمرین ها

- ✓ تمرین ۵: هدف این تمرین آشنایی با مفهوم PCB (Process Control Block) است.
  - سورس کد سیستم عامل آموزشی XV6 را از آدرس زیر دریافت کنید:

https://github.com/mit-pdos/xv6-public

PCB این سیستم عامل را یافته و به سوالات زیر پاسخ دهید:

1. Struct مربوط به PCB از چه فیلدهایی تشکیل شده است؟
2. هر یک از متغیرهای زیر در PCB به چه منظوری ذخیره می شوند؟
   i. Sz
   ii. State
   iii. Context
   iv. Ofile
   v. Killed

nastooh@aut.ac.ir

تمرین ها

- ✓ تمرین ۶: دربارهٔ تفاوت های Fork و Thread تحقیق کنید.

- ✓ تمرین ۷: هنگام ایجاد سرنخ ها، تفاوت بین سرنخ های سنکرون و آسنکرون چیست؟
  - مرجع [1]، بخش 4-4.

- ✓ تمرین ۸: دربارهٔ حافظهٔ Transactional در بانکهای اطلاعاتی و ارتباط آن با همگام سازی پروسه ها تحقیق کنید.
  - تفاوت های STM و HTM را بیان کنید.
  - مرجع [1]، بخش 7-5-1

nastooh@aut.ac.ir

تمرین ها

- ✓ تمرین ۹: نخ ها در ویندوز چه شباهت ها و تفاوت های رفتاری با Pthread دارند؟
  - مرجع [1]، بخش های 1-4-4 و 2-4-4.
- ✓ تمرین ۱۰: استخر نخ ها چیست و برای حل چه مشکلاتی از آن استفاده می شود؟
  - مرجع [1]، بخش 1-5-4.
- ✓ تمرین ۱۱: استاندارد IEEE 1003.1c در مورد نخ ها را به اختصار بررسی کنید.
  - مرجع [2]، بخش 3-2-2.

nastooh@aut.ac.ir

تمرین ها

- تمرین ۱۲: متغییرهای شرطی در پیاده سازی مانیتورها چه نقشی دارند؟
  - مرجع [2]، بخش صفحه ۲۵۸.
- تمرین ۱۳: استفاده از barrier در کد به چه منظور است؟ با مثال نشان دهید.
  - مرجع [2]، بخش 9-3-2.

nastooh@aut.ac.ir

تمرین ها

- ✓ تمرین ۱۴: راه کارهای الف تا ح را به صورت جداگانه برای محافظت از نواحی بحرانی در نظر گرفته و ارضای شروط را در مورد هر یک بررسی کنید.

  - بخش الف:
    - مقدار اولیهٔ هر دو فلگ صفر است.

A
1: Flag[A]=0; If(flag[B]!=0) then goto 1; Flag[A]=1; If(flag[B]!=0) then goto 1; **C.S.** Flag[A]=0

B
1: Flag[B]=0; If(flag[A]!=0) then goto 1; Flag[B]=1; If(flag[A]!=0) then goto 1; **C.S.** Flag[B]=0

nastooh@aut.ac.ir

تمرین ها

- بخش ب: مقدار اولیهٔ فلگ ها برابر صفر است. دستور pause سبب ایجاد تاخیر با اندازهٔ تصادفی در اجرای برنامه می شود.

P1

```
while (true)
{
flag[1]=1;
While(flag[2])
    {
       Flag[1]=0;
       pause();
       flag[1]=1;
    }
Critical Section;
flag[1]=0;
}
```

P2

```
while (true)
{
flag[2]=1;
While(flag[1])
    {
       Flag[2]=0;
       pause();
       flag[2]=1;
    }
Critical Section;
flag[2]=0;
}
```


nastooh@aut.ac.ir

تمرین ها

- بخش ج:
  - مقدار اولیهٔ N1 و N2 صفر است.

P1

```
Loop
{
   …..
   N1=1;
   N1=N2+1;
   Loop exit when (N2==0 OR N1<=N2);
   C.S.
   …..
}end of loop;
```

P2

```
Loop
{
   …..
   N2=1;
   N2=N1+1;
   Loop exit when (N1==0 OR N2<=N1);
   C.S.
   …..
}end of loop;
```


nastooh@aut.ac.ir

تمرین ها

- بخش د:
  - در واقع فقط دو فرآیند P1 و P0 کد زیر را اجرا می کنند.
  - مقدار اولیهٔ فلگ ها false است.

P(int i)

```
While (true)
{
    ….
    While((flag[i+1] mode 2)!=false);
    Flag[i]=true;
    C.S.
    Flag[i]=false;
    ….
}
```


nastooh@aut.ac.ir

تمرین ها

- بخش ه:
  - مقدار اولیهٔ فلگ ها false است.

P(int i)

```
While (true)
{
    ….
   Flag[i]=true;
   While (turn != i)
    {
      while (flag[1-i]);
      turn=i
    }
C.S.
    Flag[i]=false;
    ….
}
```

- یادآوری: در زبان های خانوادهٔ سی، می توان شرط if(x!=0) را به صورت If(x) نوشت. هر عددی به جز صفر برابر True است و صفر برابر false.

nastooh@aut.ac.ir

تمرین ها

- بخش و:
  - مقدار اولیهٔ فلگ ها false است.

P0

```
…..
   Flag[0]=true;
   While (flag[1]==true)
    {
      if (turn==1)
         {
             Flag[0]=false;
             While(turn==1);
             Flag[0]=true;
          }
       }
    C.S;
    turn=1;
    Flag[0]=false;
…..
```

P1

```
…..
   Flag[1]=true;
   While (flag[0]==true)
    {
      if (turn==0)
         {
             Flag[1]=false;
             While(turn==0);
             Flag[1]=true;
          }
       }
    C.S;
    turn=0;
    Flag[1]=false;
…..
```


nastooh@aut.ac.ir

تمرین ها

- بخش ز:
  - مقدار اولیهٔ سمافورها برابر یک است.

P0

```
While(true)
{
   wait(A);
   wait(B);
   C.S.
   signal(B);
   signal(A);
}
```

P1

```
While(true)
{
   wait(B);
   wait(A);
   C.S.
   signal(A);
   signal(B);
}
```


nastooh@aut.ac.ir

تمرین ها

- ✓ تمرین ۱۵: مسائل a تا e را به کمک سمافور مدیریت کنید.

  - بخش a: مسالهٔ تولید کننده و مصرف کننده را در حالتی که بافر **نامحدود** است با سمافور مدیریت کنید.

  - بخش b: مسالهٔ تولید کننده و مصرف کننده را در حالتی که بافر فقط یک واحد است با سمافور مدیریت کنید.

  - بخش c: مسالهٔ نویسندگان و خوانندگان را با اولویت نویسندگان و به کمک سمافور مدیریت کنید.
    - اگر صفی از خوانندگان منتظر اتمام کار یک نویسنده هستند و یک نویسنده از راه برسد، اولویت این نویسنده بیشتر از تمام خوانندگان منتظر است. (شبیه به مساله اولویت **مطلق** خوانندگان)

nastooh@aut.ac.ir

تمرین ها

- بخش d: مسالهٔ نویسندگان و خوانندگان را بدون اولویت و به کمک سمافور مدیریت کنید.
  - خوانندگان و نویسندگان بدون اولویت هستند و به ترتیب ورودشان به سیستم سرویس می گیرند.
  - یعنی اگر در ابتدای امر چند خواننده پشت سر هم وارد شوند، باید بتوانند منبع را در اختیار بگیرند، اما اگر در این حین یک (یا چند) نویسنده وارد شد، نویسندگان باید به ترتیب منتظر اتمام خواندن خوانندگان اولیه بمانند. و اگر بعد از آنها یک خواننده وارد شد، (در حالی که هنوز خوانندگان اولیه در حال خواندن هستند) این خواننده جدید نمی تواند شروع به خواندن کند و باید بعد از نوشتن نویسندگانِ منتظر، شروع به خواندن کند.

- بخش e: سعی کنید مسالهٔ کلاسیک آرایشگر خواب آلود را با سمافورها پیاده سازی کنید. (تعریف مساله در متن اسلایدها)
  - بن بست نباید رخ دهد.
  - دونمونه از حالت های پیچیده تر این مساله در مرجع [3] ضمیمهٔ A.2 وجود دارند. دقت کنید این تمرین بسیار ساده تر از آن دو حالت هستند.

nastooh@aut.ac.ir

پروژه فصل سه

- برای این فصل دو پروژه برنامه نویسی در نظر گرفته شده است که جزئیات، نحوهٔ پیاده سازی و ارائه آنها در کلاس های تدریسیار ارائه می گردد.

nastooh@aut.ac.ir

نمونه تست های کنکور ارشد

✓ ارشد، دولتی، ۷۶

- فرض کنید دو پردازهٔ P1 و P2 به صورت همروند وجود دارند. در این صورت کدام یک از موارد زیر نمی تواند خروجی اجرای دو پردازه باشد؟

P1
Repeat print “C” print “D” forever

P2
Repeat print “A” print “B” Forever

الف) C*A*B*D* ب) (AB)*(CD)* ج) A(CD)*B د) BCAD

nastooh@aut.ac.ir

نمونه تست های کنکور ارشد

✓ ارشد، آزاد، ۸۷

- اگر مقدار اولیهٔ متغییر x برابر صفر باشد، در صورتی که i متغییر مشترک بین دو فرآیند p0 و p1 باشد با اجرای همروند در یک سیستم تک پردازنده ای حداکثر مقدار متغییر x پس از اجرای دو فرآیند چه مقداری می تواند باشد؟

P0
For(i=0; i<3; i++) x++;

P0
For(i=0; i<3; i++) x--;

الف) 6 ب)3 ج) 0 د) 4

- راهنمایی: هنگام اجرای حلقهٔ for، وقفهٔ تعویض متن می تواند پس از اجرای هر کدام از مراحل سه گانهٔ حلقهٔ For رخ دهد. (مراحل انتصاب اولیه، چک کردن شرط، اجرای گام حلقه)

nastooh@aut.ac.ir

نمونه تست های کنکور ارشد

✓ ارشد، آزاد، ۸۴

- اگر مقدار اولیهٔ در سمافورهای S1 و S2 برابر صفر باشد، با اجرای همروند سه فرآیند P0 و P1 و P2 کدام رشته خروجی از چپ به راست چاپ نمی شود؟

P0	P1	P2

While(true)
{
Wait(S1);
Printf (C);
Wait(S1);
Printf(C);
}
	While(true)
	{
	signal(S2);
	Printf (A);
	Wait(S1);
	Printf(A);
	}
		While(true)
		{
		wait(S2);
		Printf (B);
		signal(S1);
		Printf(B);
		Signal(S1);
		}

الف) ABBC　　ب) ABCB　　ج) BBCC　　د) BCAA

nastooh@aut.ac.ir

منابع

[1]. A. Silberschatz, P. B. Galvin and G. Gagne, **“Operating System Concepts,”** 10th ed., John Wiley Inc., 2018.

[2] A. S. Tanenbaum and H. Bos, “**Modern Operating Systems,**” 4rd ed., Pearson, 2015.

[3] W. Stallings, **“Operating Systems,”** 9th ed., Pearson, 2018.

[4] A. S. Tanenbaum, A. S. Woodhull**, “Operating Systems Design and Implementation,”** 3rd ed., Pearson, 2006.

[5] نستوه طاهری جوان و محسن طورانی، **“اصول و مفاهیم سیستم عامل،”** انتشارات موسسه آموزش عالی پارسه، ۱۳۸۶.

nastooh@aut.ac.ir

پایان

nastooh@aut.ac.ir