فصل چهارم

فرآیندهای همروند و مسایل هم‌زمانی

۴-۱- هم‌زمانی پردازش‌ها و اصول هم‌زمانی

یک پردازش همکار، پردازشی است که می‌تواند به سایر پردازش‌های در حال اجرا، اثر گذارد و یا از آن‌ها متاثر شود. پردازش‌های همکار، ممکن است یا مستقیماً یک فضای آدرس منطقی (کد یا داده) را به اشتراک گذارند و یا اجازه داشته باشند که داده را تنها از طریق فایل‌ها تقسیم نمایند. حالت اشتراک حافظه از طریق استفاده از فرآیندهای سبک وزن یا رشته‌ها، حاصل می‌گردد و دسترسی همروند به داده مشترک ممکن است منجر به ناسازگاری داده گردد. چنین وضعیتی را که دو یا چند فرآیند به متغیر یا متغیرهای اشتراکی دسترسی پیدا می کنند و آن‌ها را تغییر می دهند و نتیجه آن‌ها بستگی به زمان دسترسی فرآیندها دارد، شرط رقابتی یا (Race Condition) گویند. در این فصل مکانیزم‌های گوناگونی مورد بررسی قرار می گیرد تا اجرای مرتب فرآیندهای همکار با فضای آدرس منطقی مشترک تضمین شود و در نتیجه سازگاری داده محفوظ بماند.

در یک سیستم تک پردازنده‌ای و چند برنامگی، فرآیندها در طول زمان در بین یکدیگر اجرا می‌شوند، تا اجرای هم‌زمان را نشان دهند. گرچه پردازش موازی واقعی حاصل نمی‌شود اما منافع عمده ای را در پردازش موثر و سازماندهی برنامه موجب می‌شود. در سیستم‌های چند پردازنده‌ای، نه تنها ممکن است فرآیندها در بین یکدیگر اجرا شوند، بلکه می‌توانند واقعاً به موازات هم و با هم پوشانی اجرا گردند. در سیستم تک پردازنده‌ای، سرعت اجرای فرآیندها به یکدیگر وابسته هستند و چگونگی رفتار سیستم عامل با وقفه‌ها و سیاست‌های زمان‌بندی سیستم عامل تاثیر قابل ملاحظه ای بر روی سرعت اجرای فرآیندها دارد. در سیستم‌های چند پردازنده‌ای هم، سرعت اجرای فرآیندها قابل پیش بینی نیست و همانند سیستم تک پردازنده‌ای به مسایل هم‌زمانی اجرای فرآیندها نیز باید پرداخت. در سیستم تک پردازنده‌ای مشکل هم‌زمانی ناشی از وقفه ای است که می‌تواند اجرای دستورالعمل را در هر کجای فرآیند متوقف نماید و همین وضعیت در سیستم چندپردازنده‌ای می‌تواند رخ دهد. به اضافه این که ممکن است دو فرآیند هم‌زمان اجرا شوند و هر دو سعی در دسترسی به یک متغیر سراسری داشته باشند. در هر حال، راه حل هر دو نوع مساله، یکسان و آن کنترل دست‌یابی به منابع مشترک است.

۴-۱-۱- نواحی بحرانی (Critical Section)

سیستمی متشکل از n فرآیند را در نظر بگیریدکه هر فرآیند دارای قطعه کدی است به نام ناحیه بحرانی که در آن ناحیه ممکن است متغیرهای مشترک تغییر نمایند، جدولی به هنگام گردد و یا فایلی به اشتراک گذاشته شود.

در این سیستم هنگامی که یک فرآیند در ناحیه بحرانی خود به سر می برد، هیچ فرآیند دیگری نباید اجازه اجرا در ناحیه بحرانی خود را داشته باشد. هر فرآیند باید برای وارد شدن به ناحیه بحرانی اش اجازه بگیرد و سپس در صورتی‌که مجوز صادر شد، آن‌گاه وارد ناحیه بحرانی‌اش گردد. بخشی از کد که این درخواست را پیاده سازی می‌کند. بخش" **ورود به ناحیه بحرانی"(Entry Section)** نامیده می‌شود. ناحیه بحرانی با یک بخش " **خروج از ناحیه بحرانی "(Exit Section)** می‌تواند تمام گردد و کد باقیمانده را **بخش "(remainder Section) باقیمانده کد"** می‌نامند.

در ذیل نمونه‌ای از کد نمایش داده می‌شود.

Repeat

Entry Section ← ورود به ناحیه بحرانی

Critical Section ←

Exit Section ←

Remainder Section ← باقیمانده کد

Forever

یک راه حل برای مشکل ناحیه بحرانی باید سه نیاز زیر را ارضا نماید:

۱ـ دوبه‌دو ناسازگار یا انحصار متقابل (Mutual Exclusion)

اگر فرآیندی در حال اجرا در داخل ناحیه بحرانی باشد، در آن‌صورت هیچ فرآیندی در ناحیه بحرانی خود نمی‌تواند اجرا گردد.

۲- پیشرفت (Progress)

اگر هیچ فرآیندی در ناحیه بحرانی خود نباشد و فرآیندهای دیگر تمایل به ورود به نواحی بحرانی خود داشته باشند، فقط آن فرآیندهایی که هنوز به بخش ناحیه بحرانی نرسیده‌اند، می‌توانند در تصمیم‌گیری درباره این‌که کدام فرآیند به ناحیه بحرانی خود وارد گردد، شرکت کنند و این انتخاب نمی‌تواند به طور نامحدود به تعویق انداخته شود.

۳- انتظار محدود (Bounded Waiting)

برای تعداد دفعاتی که به فرآیندهای دیگر اجازه داده می‌شود به ناحیه بحرانی خود وارد شوند، باید حدی وجود داشته باشد، این محدودیت می‌باید بعد از این‌که یک فرآیند تقاضایی برای ورود به ناحیه بحرانی اش درخواست کرد و قبل از این‌که آن تقاضا مورد پذیرش قرار گیرد، اعمال شود.

در ذیل پنج راه حل برای نواحی بحرانی توضیح داده می‌شود:

۱- راه حل نرم افزاری

فرآیندها بدون هیچ حمایتی از زبان برنامه‌سازی یا سیستم عامل خود با یکدیگر همکاری می کنند.

۲- راه حل سخت افزاری

فرآیندها به کمک دستورالعمل سخت افزاری ماشین، همگام می‌شوند.

۳ـ سیستم عامل و ابزارهای هم‌زمانی

فرآیندها به کمک سمافورها همگام می‌شوند.

۴- حمایت ساختاری زبان برنامه نویسی

فرآیندها به کمک مانیتور همگام می‌شوند.

۵ـ تبادل پیام

فرآیندها در یک محیط غیرمتمرکز همگام می‌شوند.

۴-۱-۱-۱- راه حل نرم افزاری

در این بخش توجه مان را در مورد الگوریتم‌های قابل اعمال به تنها دو فرآیند همروند معطوف می‌نماییم. فرآیندها به P0 و P1 شماره‌گذاری می‌شوند. وقتی Pi را نشان می‌دهیم، منظورمان از Pj فرآیند دیگر است (j=i-1).

ضمناً این نکته حایز اهمیت است که برای اثبات صحت و سقم الگوریتم‌های مربوط به نواحی بحرانی درذیل، همیشه می‌بایست فرض بر این باشد که زمان پردازنده برای سرویس‌دهی فرآیندهای همروند در فاصله زمانی محدود و مشخص بنا به زمان‌بندی پردازنده تسهیم می‌شود و این احتمال وجود دارد که در انتهای هر دستورالعمل ماشین بر اثر وقوع وقفه، کنترل پردازنده از یک فرآیند به فرآیند دیگر منتقل گردد.

الگوریتم اول:

کد فرآیند Pi را در نظر بگیرید.

```
Repeat
While (Turn<> i);
   Critical Section
Turn=j;
   Remainder Section
Forever
```

متغیر Turn به صورت اشتراکی بین فرآیندها به مقدار اولیه i یا j مقدار دهی می‌شوند.

اگر Turn= i باشد فرآیند Pi وارد ناحیه بحرانی اش می‌شود و سپس Turn=j می‌شود تا اجازه دهد فرآیند بعدی وارد ناحیه بحرانی گردد. این راه حل اطمینان می‌دهد که تنها یک فرآیند در یک زمان، می‌تواند وارد بخش بحرانی‌اش گردد، اما نیاز پیشرفت را ارضا نمی‌نماید. زیرا چنان‌چه یکی از فرآیندها در خارج ناحیه بحرانی کارش به اتمام برسد، پردازش بعدی، پس از اجرای ناحیه بحرانی‌اش نوبت را به فرآیندی که هم اکنون به پایان رسیده است می‌دهد و دیگر خودش نمی‌تواند وارد ناحیه بحرانی گردد، بدین معنی که اجرای تناوبی فرآیندها در بخش بحرانی شان ضروری است.

الگوریتم دوم:

کد زیر را در نظر بگیرید:

```
Repeat
(1) -------- Flag[i] = True;
(2) -------- While (Flag[j]) ;
   Critical Section
   Flag[i] = False;
   Remainder Section
    Forever
```

Flag[i] و Flag[j] آرایه‌ای از نوع منطقی (Boolean) می‌باشند. مقدار اولیه هر دو متغیر False,Flag می‌باشد.

چنان‌چه هر دو فرآیند Pi , Pj، Flag های خود را True نمایند آن‌گاه هر دو فرآیند در حلقه While، باقی می‌مانند و **بن‌بست** رخ می‌دهد و نیاز پیشرفت مرتفع نمی‌شود. توجه داشته باشید که نیاز انحصار متقابل ارضا می‌گردد.

ضمناً تغییردادن ترتیب دستورالعمل‌های ۱ و ۲ در کد فوق الذکر مشکل را حل نمی‌نماید، بلکه وضعیتی خواهیم داشت که هر دو پردازش ممکن است به طور هم‌زمان در بخش بحرانی اشان وارد شوندکه **نقض نیاز انحصار متقابل** رخ خواهد داد.

الگوریتم Dekker

اولین راه حل نرم افزاری است که توسط ریاضی دان آلمانی به نام Dekker ارایه شد.

```
Repeat
  .........
  .........

Flag[i] =True;
While  (Flag[j])
     If (Turn==j)
        {Flag[i]=False ;
         While (Turn==j);
         Flag[i]=True; }

  Critical  Section

Turn=j;
Flag[i] =False ;
  Remaineder  Section
  ..........
  ..........
  Forever
```

در این الگوریتم دو متغیر سراسری Flag[i],Flag[j] به‌عنوان آرایه‌ای از متغیرهای منطقی و Turn به‌عنوان یک متغیر Integer (که نوبت ورود فرآیند مطلوب را به داخل CS نشان می‌دهد) به مقدار i , j تغییر می‌نماید، مورد استفاده قرار می‌گیرند.

فرآیند Pi,Pj تمایل خود را برای ورود به ناحیه بحرانی با True کردن متغیر منطقی Flag مربوط به خود اعلام می دارند. سپس این فرآیندها در شرط دستور While بررسی می کنند که آیا فرآیند طرف مقابل مایل به ورود به ناحیه بحرانی خود می‌باشد یا خیر. در صورتی که متغیر منطقی فرآیند طرف مقابل " False" باشد، فرآیند مربوطه، بدنه حلقه While را پشت سر می‌گذارد و به ناحیه بحرانی خود وارد خواهد شد.

فرض کنید هنگام بررسی، فرآیندPi متوجه گردد که متغیر منطقی متناظر با فرآیند Pj، True است.

این امر سبب خواهد شد تا فرآیند Pi وارد بدنه حلقه While خود گردد که در این‌جا متغیر Turn را مدنظر قرار می‌دهد. زمانی که هر دو فرآیند در آن واحد بخواهند وارد ناحیه بحرانی خود گردند، برای جلوگیری از تداخل از متغیر Turn استفاده می‌شود.

اگر Pi فرآیند مطلوب باشد (بدین معنی که Turn=i) بدنه "If" را پشت سر می‌گذارد و با این انتظار که فرآیند Pj متغیر منطقی متناظرش را "False" نماید، به طور مکرر شرط While را بررسی می‌نماید. فرآیند Pj نهایتاً مجبور به انجام این کار است. در صورتی که Pj فرآیند مطلوب باشد (بدین معنی که Turn=j). فرآیند Pi وارد بدنه "If" می‌گردد. و متغیر منطقی خود را "False" می‌نماید، سپس در حلقه "While" بعدی، تا زمانی که فرآیند Pj مطلوب باشد، در انتظار باقی می‌ماند. به این ترتیب Pi با "False" کردن متغیر منطقی متناظرش اجازه می‌دهد تا Pj وارد ناحیه بحرانی خود گردد. نهایتاً Pj از ناحیه بحرانی خود خارج می‌شود که خروج از این ناحیه را اجرا می‌کند و در طی آن Pi را فرآیند مطلوب تعیین کرده و ضمناً متغیر منطقی خاص خود را " False" می‌نماید. Pi اکنون می‌تواند حلقه داخلی خود را پشت سر بگذارد و متغیر منطقی خود را "True" نماید. سپس متعاقباً حلقه " While" خارجی خود را به اجرا می‌گذارد. اگر متعیر منطقی متناظر با Pj که اخیراً "False" شده بود هنوز در همان حالت باقی مانده باشد، Pi وارد ناحیه بحرانی خواهد شد. در صورتی که Pj به سرعت سعی نماید وارد ناحیه بحرانی خود گردد، متغیر منطقی آن در این لحظه True بوده و لذا Pi بار دیگر ناچاراً به "While" خارجی خود باز می‌گردد. در این لحظه Pi فرآیند مطلوب است، بنابراین از بدنه "If" می‌گذرد و مکرراً "While" خارجی را اجرا می‌کند تا این‌که Pj ناچاراً متغیر منطقی خود را "False" کرده و به Pi امکان ورود به ناحیه بحرانی را بدهد.

احتمال جالب توجهی که در این الگوریتم وجود دارد این است که Pj متغیر منطقی متناظر خود را سریعاً (قبل از این‌که نوبت به Pi برسد تا حلقه خارجی While را تست نماید) True نماید و مجدداً وارد ناحیه بحرانی شود و Pi با تست حلقه خارجی While در انتظار ورود به ناحیه بحرانی باقی بماند. توجه داشته باشید که این بار زمانی که Pj از ناحیه بحرانی خود خارج می‌شود، دیگر احتمال ورود آن به ناحیه بحرانی ممکن نخواهد بود و این بار نوبت Pi است که وارد ناحیه بحرانی خود گردد، زیرا با "False" کردن متغیر منطقی خود اجازه ورود به ناحیه بحرانی را به Pi می‌دهد و خود وارد بدنه "While" خارجی و سپس وارد بدنه "If" و نهایتاً در بدنه "While" داخلی در انتظار باقی می‌ماند.

الگوریتم پیترسون

به عنوان دومین راه حل نرم افزاری که هر ۳ شرط ناحیه بحرانی را ارضا می‌نماید ارایه شد.

کد زیر را در نظر بگیرید:

```
Repeat
Flag[i]=True;
Turn=j;
While (Flag[j] and Turn==j);
Critical Section
Flag[i]=False;
Remainder Section
Forever
```

فرآیند i و j از دو متغیر مشترک Flag[i], Flag[j] به‌عنوان آرایه‌ای از متغیرهای منطقی و Turn که یک متغیر Integer می‌باشد و به مقدار i یا j مقدار دهی می‌شود، استفاده می‌نماید.

در آغاز Flag ها هردو False و مقدار Turn به دلخواه صفر یا یک خواهد بود.

ابتدا ثابت می‌کنیم که هیچ گاه دو فرآیند با هم نمی‌توانند وارد ناحیه بحرانی خود گردند. برای ورود به ناحیه بحرانی ابتدا Pi مقدار Flag[i] را True می‌نماید و سپس اعلام می‌کند که نوبت فرآیند دیگر است تا وارد ناحیه بحرانی گردد. (Turn=j).

در صورتی که هر دو فرآیند سعی کنند که در آن واحد وارد ناحیه بحرانی گردند، تقریباً متغیر Turnدر یک لحظه به هردو مقدار i و j مقدار می‌گیرد که تنها یکی از این دو عمل انتساب در آخر حفظ می‌شود. مقدار نهایی Turn بیان می‌کند که کدام‌یک از دو فرآیند ابتدا اجازه ورود به ناحیه بحرانی را خواهد یافت. هر فرآیندی که مطلوب محسوب شود (مقدار متغیر Turn به آن فرآیند نسبت داده شود) ویا مقدار Flag فرآیند طرف مقابل False باشد به ناحیه بحرانی وارد می‌گردد. از طرفی اگر هر دو فرآیند بتوانند در آن واحد در نواحی بحرانی شان اجرا شوند باید مقدار Flagهایشان True باشند و چون مقدار Turn یا i و یا j است و نه هر دوی آن بنابراین Pi و Pj نمی‌توانند دستور While خود را تقریباً در یک لحظه با موفقیت اجرا نمایند و در نتیجه یکی از آن‌ها در حلقه While باقی می‌ماند و هر دو هم‌زمان وارد ناحیه بحرانی نخواهد شد.

حال نشان می‌دهیم که شرط پیشرفت و انتظار محدود نیز برقرار است. فرآیند Pi در صورتی از ورود به ناحیه بحرانی اش منع می‌گردد که در حلقه While به شرط (Turn=j ، Flag[j]= True) باقی بماند.

اگر Pj آماده ورود به ناحیه بحرانی نباشد Flag[j]=False خواهد بود وPi قادر به ورود به ناحیه بحرانی اش می‌باشد.

اگر فرآیند Pj، مایل به ورود به ناحیه بحرانی خود باشد (Flag[j]=True) آن‌گاه با اجرای دستور while و مقدار منطقی متغیر Turn=j وارد ناحیه بحرانی می‌شود و در غیر این‌صورت فرآیند Pi به ناحیه بحرانی خود وارد می‌شود.

هر زمان Pj از ناحیه بحرانی خارج شود Flag[j] را False می‌کند و اجازه می‌دهد که فرآیند Pi به ناحیه بحرانی وارد گردد. فرآیند Pj بعد از True کردن Flag خود مقدار متغیر Turn را نیز به مقدار i تغیر می‌دهد، بدین معنی که فرآیند Pi با موفقیت وارد بخش بحرانی خواهد شد (بعد از حداکثر یک ورود توسط Pj).

۴-۱-۱-۲- همگام سازی سخت افزاری

در این بخش دستورات سخت افزاری ساده‌ای معرفی می‌گردند که در اکثر سیستم ها موجود می‌باشند، از این دستورات سخت افزاری برای رفع اشکال ناحیه بحرانی استفاده خواهد شد.

ناتوان سازی وقفه‌ها

در یک محیط تک پردازنده‌ای می‌توان وقفه را در زمان تغییر یافتن یک متغیر مشترک، ناتوان ساخت (DISABLE). در این حالت، می‌توان مطمئن شویم که توالی کنونی دستورالعمل‌ها بدون پس گرفتن پردازنده اجرا می‌شود و هیچ تغییر غیر قابل انتظار در مورد متغیرهای مشترک نمی‌تواند صورت پذیرد. متاسفانه در سیستم‌های چند پردازنده‌ایی ممنوع کردن وقفه برای تمام پردازنده‌ها کاری **وقت گیر و باعث کاهش کارایی سیستم** خواهد شد. ضمناً این نکته قابل ذکر است که این روش فقط در هسته سیستم عامل امکان‌پذیر است. زیرا در لایه کاربر استفاده از این دستور باعث Trap می‌شود.

به کارگیری دستورالعمل‌های سخت افزاری

بسیاری از ماشین ها، دستورالعمل‌های سخت افزاری مناسبی ارایه می دهند که محتوای یک کلمه را می‌توان تست و تغییر داد و یا محتویات دو کلمه را با یکدیگر مبادله نمود، **بدون این‌که وقفه ای بین آن‌ها اتفاق افتد** و یا به عبارت دیگر این دستورالعمل‌ها به طور "**اتمی**" **یا لایتجزا** اجرا می‌شوند.

دستورالعمل TEST-AND-SET

دستور TEST-AND-SET را به شکل زیر تعریف می نماییم.

```
Function TEST-AND-SET (var Target:Boolean): Boolean;
  Begin
    TEST-AND-SET = Target;
    Target = True;
    Return (Test – AND-Set);
  End
```

چنان‌چه سیستمی دستورالعمل TEST-AND-SET را حمایت کند، می‌توان مشکل ناحیه بحرانی را با تعریف کردن یک متغیر منطقی lock با مقدار اولیه "False" حل کرد.

ساختار فرآیند Pi به صورت کلی زیر است:

```
Repeat
While (TEST-AND-SET (lock));
  Critical Section
lock = False;
Forever
```

دستورالعمل Swap

دستورالعمل Swap را به شکل زیر تعریف می نماییم.

```
Procedure Swap ( Var a,b : Boolean );
  Var  Temp: Boolean;
  Begin
      Temp=a;
      a=b;
      b= Temp;
  End
```

اگر سیستم دستورالعمل Swap را حمایت نماید آن‌گاه انحصار متقابل به شرح زیر فراهم می‌شود. ساختارِ فرآیند Pi را به شکل ذیل در نظر بگیرید.

```
Repeat
  Key = True ;
  Repeat
      Swap (lock , key);
  Until  key = False ;
  Critical Section
  Lock=  False ;
  Remainder Section
Forever
```

یک متغیر سراسری lock (به مقدار اولیه"False") و یک متغیر منطقی محلی key را برای هر فرآیند در نظر بگیرید.

هر فرآیندی که زودتر دستور "Swap" را اجرا نماید وارد ناحیه بحرانی می‌شود و متغیر lock را "True" و key خود را "False" می‌نماید. بنابراین فرآیند بعدی در حلقه "repeat" داخلی باقی می‌ماند. به محض خارج شدن یک فرآیند از ناحیه بحرانی، متغیر

سراسری False ،Lock می‌شود و اجازه می‌دهد که فرآیند بعدی وارد ناحیه بحرانی شود. البته توجه کنید که در این الگوریتم و الگوریتم مربوط به TEST-AND-SET احتمال این وجود دارد که همان فرآیندی که اخیراً از ناحیه بحرانی خود خارج شده مجدداً موفق به ورود به ناحیه بحرانی خود شود (بدون توجه به این‌که فرآیندهای دیگر هم اکنون در انتظار ورود به ناحیه بحرانی خودشان می‌باشند)، بدین معنی که **نیاز انتظار مقید را برآورده نمی سازند.**

۴-۱-۱-۳- همگام سازی با استفاده از ابزارهای هم‌زمانی

***فراخوانی سیستم* Sleep *و* wakeup**

هردو روش نرم افزاری و سخت افزاری برای حل مشکل ناحیه بحرانی صحیح می‌باشد، اما نقصی که در این دو روش مشاهده می‌شود نیاز آن‌ها به وضعیتی است به نام " **انتظار مشغول**" (Busy Waiting).

هنگامی که فرآیندی خواستار ورود به ناحیه بحرانی اش باشد، راه حل های مذکور بررسی می کنند که آیا این ورود امکان پذیر است یا خیر. اگر امکان ورود وجود نداشته باشد تا زمانی‌که شرایط ورود مهیا گردد، فرآیند در یک حلقه انتظار می نشیند. این وضعیت را **انتظار مشغول** گویند.

این رویکرد نه تنها زمان پردازنده را به هدر می‌دهد بلکه اثرات غیر قابل انتظاری را نیز خواهد داشت. کامپیوتری با دو فرآیند H با اولویت بالا و L با اولویت پایین را در نظر بگیرید. فرض کنید الگوریتم زمان‌بندی اولویتی باشد. در یک لحظه خاص که فرآیند L در ناحیه بحرانی اش قرار دارد، فرآیند H برای اجرا آماده می‌شود. فرآیند H، پردازنده را در اختیار می گیرد و در انتظار مشغول باقی می‌ماند، اما از آن‌جایی که فرآیندL تا زمانی‌که فرآیند H در حال اجرا باشد زمان‌بندی نمی‌شود، بنابراین فرآیند L هرگز شانس ترک ناحیه بحرانی اش را نخواهد داشت، به طوری که فرآیند H برای همیشه در حلقه باقی می‌ماند. این مساله را " **مساله وارونه سازی اولویت**" (Priority Inversion) گویند.

حال نگاهی به چند ارتباط بین فرآیندی خواهیم داشت که به جای هدر دادن زمان پردازنده در هنگامی که امکان ورود به ناحیه بحرانی خود را نداشته باشد. مسدود (Wait) خواهند شد.

ساده‌ترین ابزار استفاده از سرویس‌های سیستمی Wakeup، Sleep می‌باشد که در آن فراخوانی Sleep باعث مسدود شدن فرآیند فراخواننده و فراخوانی Wakeup (همراه با پارامتری که نام فرآیند را مشخص می‌نماید) باعث بیدار کردن فرآیند مربوطه می‌شود.

نحوه عملکرد این سرویس‌های سیستمی را در **مساله معروف تولید کننده و مصرف کننده** بررسی می نماییم.

فرآیند تولید کننده اطلاعاتی را تولید می‌کند که مورد مصرف فرآیند مصرف کننده قرار می‌گیرید. برای این‌که به فرآیندهای تولید کننده و مصرف کننده امکان داده شود تا هم‌زمان اجرا شوند باید از یک حافظه میانگیر برای داده‌های تولید شده به وسیله تولید کننده استفاده کرد. این حافظه میانگیر به وسیله فرآیند مصرف کننده از سوی دیگر خالی می‌شود. تولید کننده و مصرف کننده باید همگام باشند. به طوری که مصرف کننده سعی در مصرف کردن فقره اطلاعی که هنوز تولید نشده است، ننماید. در چنین وضعیتی فرآیند مصرف کننده باید در انتظار بماند تا یک فقره اطلاع ایجاد گردد. از طرفی دیگر ممکن است تولید کننده مجبور شود که برای فضای پر بافر در انتظار مصرف کننده باقی بماند. به منظور همگام سازی این دو فرآیند در جایی که قرار است مصرف کننده یا تولید کننده منتظر همدیگر بمانند می‌توان از دستور Sleep استفاده کرد. زمانی‌که تولید کننده به خواب رفت چنان‌چه مصرف کننده یک یا چند فقره از داده‌ها را به مصرف رساند، تولید کننده را با دستور سیستمی "Wakeup" بیدار می‌نماید. زمانی که مصرف کننده به خواب می‌رود چنان‌چه تولید کننده یک یا چند فقره را تولید نمود مصرف کننده را با دستور سیستمی "Wakeup" بیدار می‌کند. برای ثبت و ردیابی تعداد اقلام داده‌ای در بافر نیاز به متغیری به نام Counter داریم.

فرض کنیم حداکثر تعداد داده‌های بافر N باشد. کد مصرف کننده و تولید کننده را به شکل ذیل در نظر بگیرید.

```
Procedure Producer() /* تولید کننده */
  { While (True)
      { Produce Item;                          /× یک فقره اطلاعاتی تولید کن ×/
    If (Counter==N) Sleep;                     /× چنان‌چه بافر پر بود به خواب برو ×/
        InSert Item to Buffer                  /× فقره اطلاعاتی تولید شده را در بافر قرار بده ×/
      Counter ++;                              /× تعداد اقلام در بافر را یک واحد افزایش بده ×/
      If (Counter ==1) Wakeup(Consumer);       /× چنان‌چه بعد از افزایش تعداد اقلام
      }                                        /× در بافر، فقط ۱ فقره اطلاع موجود باشد ×/
  }                                            /× آن‌گاه فرآیند مصرف کننده را که احتمالا ممکن است در انتظار
                                               /× باشد با دستور Wakeup بیدار کن ×/
```

توجه داشته باشید که چنان‌چه فرآیند مصرف کننده در خواب نباشد، دستور Wakeup هیچ عملی انجام نمی‌دهد. (سیگنال به هدر می‌رود.)

```
Procedure Consumer( )
  { While (True)
      { If (Counter ==0) Sleep;   /× چنان‌چه بافر خالی است به خواب برو ×/
        Remove Item from Buffer
        Counter --;
        If (Counter == N-1) Wakeup (Producer);
        Consume Item ;
      }
  }
```

چنان‌چه قبل از برداشت یک فقره اطلاعات از بافر، بافر کاملاً پر باشد احتمال این وجود دارد که تولید کننده به خواب رفته باشد، پس هم اکنون که یک فضای خالی در بافر ایجاد شده است تولید کننده را بیدار کن. اگر تولید کننده در خواب نباشد، دستور Wakeup هیچ عملی انجام نمی‌دهد. (سیگنال به هدر می‌رود)

کدهای مذکور به نظر ساده می رسند، اما تحت شرایط خاصی **به شرط رقابتی** ختم خواهند شد.

سناریوی ذیل را در نظر بگیرید.

فرض کنید بافر خالی است و مصرف کننده، Counter را می‌خواند برای آن‌که ببیندمقدار آن صفر است یا خیر، در آن لحظه زمان‌بند پردازنده موقتاً اجرای مصرف کننده را متوقف می‌نماید و اجرای تولید کننده آغاز می‌شود. تولید کننده یک فقره داده را در بافر وارد می‌کند و مقدار Counter را یک واحد افزایش می‌دهد. چون مقدار آن صفر بود هم اکنون ۱ می‌شود، بنابراین تولید کننده Wakeup را صادر می‌کند تا مصرف کننده را بیدار نماید. متاسفانه مصرف کننده هنوز به خواب نرفته است، بنابراین سیگنال Wakeup مفقود می‌شود. چنان‌چه مصرف کننده برای بار بعدی اجرا شود چون مقدار Counter قبلی را خوانده بود، در این‌جا به خواب (Sleep) می‌رود.

فرآیند تولید کننده مرتباً داده‌هایی را تولید می‌کند و در بافر قرار می‌دهد تا این‌که بافر پر شود. به محض پر شدن بافر این فرآیند هم به خواب می‌رود. در نتیجه هر دو فرآیند **برای همیشه به خواب خواهند رفت.** مشکل اصلی این‌جاست که سیگنالWakeup ارسال شده برای یک فرآیندی که هنوز به خواب نرفته است گم می‌شود. در صورتی که این سیگنال مفقود نشود تمام مسایل حل می‌شود.

برای تصحیح این مشکل یک راه حل افزودن یک بیت انتظار بیدار شدن (Wakeup Waiting Bit) به فرآیند است.

۴-۱-۱-۳-۱- سمافورها (Semaphores)

به‌عنوان یک ابزار هم‌زمانی برای مشکل ناحیه بحرانی به کار گرفته می‌شود. یک سمافور S، متغیر صحیحی است که مقدار آن غیر از آغاز دهی (initialize) می‌تواند فقط با دو عمل اتمی استاندارد Wait یا P و Signal یا V قابل دست‌یابی و تغییر باشد. سمافورهای باینری می‌توانند فقط مقدار صفر و ۱ را بپذیرند. و سمافورهای شمارشی تمام اعداد صحیح و مثبت را می پذیرند.

تعاریف کلاسیک Wait و Signal به شرح زیر آست:

```
Wait (S)      ------>  While  (S <= 0);
                       S=S-۱

Signal (S)    ------>  S=S+۱
```

دو دستور فوق (signal و Wait) باید بدون وقفه اجرا شوند. وقتی فرآیندی مقدار سمافور را تغییر می‌دهد، هیچ فرآیند دیگری نمی‌تواند همان سمافور را عوض نماید.

کاربرد سمافور باینری

می‌توانیم سمافورهای باینری را در کار با **مساله بخش بحرانی** برای n فرآیند مورد استفاده قرار دهیم. n فرآیند یک سمافور ME (Mutual Exclusion) را با مقدار اولیه 1 به اشتراک می گذارند. هر فرآیند Pi به شکل زیر سازماندهی می‌شود.

```
Repeat
Wait (ME) ;
Critical  Section
Signal (ME);
Remainder Section
Forever;
```

به این ترتیب برای n فرآیند شرط انحصار متقابل برآورده می‌شود.

سمافور باینری نیز می‌تواند برای **مسایل هم‌زمانی** به کار گرفته شود. به عنوان مثال، فرآیند P1 را با مجموعه دستورالعمل‌های S1 و فرآیند P2 را با مجموعه دستورالعمل‌های S2 در حال اجرای همروند در نظر بگیرید.

در نظر داریم که S2 تنها بعد از اجرای S1 اجرا شود. می‌توانیم این نیاز را با سمافور باینری Sync با مقدار اولیه صفر به شکل ذیل پیاده سازی نماییم.

```
P1                    P2
-------------         -------------
  S 1;                Wait(Sync);
  Signal (Sync);      S2;
```

چون مقدار Sync صفر است چنان‌چه فرآیند P2 اول اجرا شود، با دستور Wait در انتظار خواهد ماند تا این‌که Signal مربوطه از طرف فرآیند P1 صادر شود و زمانی این سیگنال ارسال می‌شود که S1 اجرا شده به اتمام برسد. بعد از ارسال Signal، S2 اجرا می‌شود و به این ترتیب نیازمان برآورده می‌شود.

کاربرد سمافور های شمارشی

سمافورهای شمارشی هنگامی‌که منبعی از بین مجموعه‌ای از منابع یکسان به فرآیندی اختصاص پیدا می‌کند مفید خواهند بود. این نوع سمافور با تعداد منابع مجموعه، مقدار اولیه می گیرد. هر عمل Wait سمافور را یک واحد کاهش می‌دهد که به مفهوم این است که منبعی از مجموعه حذف و توسط فرآیندی به کار گرفته شده است. هر عمل Signal یک واحد به سمافور می افزاید که به مفهوم بازگشت یک منبع به مجموعه می‌باشد. اگر هنگامی که سمافور صفر است عمل Wait اجرا شود، فرآیند باید تا زمانی که منبعی توسط عمل Signal به مجموعه بازگردد، در انتظار بماند.

پیاده سازی سمافورها

مشکل اصلی راه حل سمافور تعریف شده فوق **انتظار مشغول** بودن آن است. این نوع سمافور به نام **قفل چرخشی** (زیرا فرآیند در حال انتظار برای قفل، می چرخد) می‌باشد. این سمافورها در سیستم‌های چند پردازنده مفید است. مزیت این نوع سمافور، آن است که نیازی به "Context Switch" در موقع انتظار فرآیندی برای قفل ندارد. بنابراین اگر انتظار رود که قفل ها زمان کوتاهی طول می‌کشند، قفل های چرخشی، مفیدند.

به منظور رفع مشکل انتظار مشغول، تعریف اعمال Wait و Signal را برای سمافورها می‌توانیم تغییر دهیم. سمافور را به مانند یک رکورد تعریف می‌کنیم.

```
Type Semaphore = Record
        value : Integer;
        l: List of Processes;
    End
```

هر سمافوری دارای یک مقدار صحیح و یک لیست از فرآیندها می‌باشد. وقتی فرآیندی باید به انتظار سمافوری بماند، به لیست فرآیندهای منتظر اضافه می‌شود. عمل Signal یک فرآیند را از صف انتظار فرآیندها خارج می‌کند و آن‌را بیدار می‌نماید.

اعمال سمافور به شکل زیر تعریف می‌شوند.

```
Wait (S) :  S. value=S. value-1;
              If (S. value <0)
            { Add this process to S. l ;
              Sleep;
            }
Signal(s) :  S. value=S. value + 1;
            If (S. value  >=0)
            { Remove  process  P from S. l
              Wakeup(P)
            }
```

دستورات Wakeup، Sleep همان سرویس‌های سیستمی یا فراخوانی های سیستمی می‌باشند.

دقت کنید که اگر چه بر اساس تعریف کلاسیک سمافورها با خاصیت انتظار مشغول، مقدار سمافور هرگز منفی نمی‌شود، این تعریف فوق می‌تواند مقادیر منفی سمافور را در برداشته باشد. قدر مطلق مقدار سمافور منفی تعداد فرآیندهای منتظر آن سمافور را نشان می‌دهد. این حقیقت نتیجه تعویض ترتیب عمل تست و کاهش یک واحدی، در تعریف Wait می‌باشد.

برای پیاده سازی تعریف فوق بایستی تضمین کنیم که هیچ دو فرآیندی نمی‌توانند اعمال Wait و Signal را بر روی یک سمافور در یک آن اجرا نمایند. این وضعیت مساله بخش بحرانی است و در یک محیط تک پردازنده‌ای، به راحتی می‌توانیم درحین اجرای اعمال Waitو Signal وقفه‌ها را موقوف نماییم. بدین ترتیب تنها یک فرآیند در حال اجرا، دستورات را اجرا کرده تا وقفه‌ها مجدداً مجاز گردند و زمان‌بند بتواند کنترل را به دست آورد. در یک محیط چند پردازنده‌ای چنان‌چه سخت افزار دستورالعمل ویژه‌ای را ارایه ندهد می‌توانیم از راه‌های نرم افزاری برای مسایل بخش بحرانی کمک گرفته، مساله بخش بحرانی شامل روتین های Waitو Signal فوق الذکر را حل نماییم.

توجه داشته باشید که استفاده از روش‌های نرم افزاری وسخت افزاری برای پیاده سازی سمافورها، مشکل انتظار مشغول را به همراه دارد. برای جلوگیری و اجتناب از این مشکل، پیاده سازی عملیات سمافورها معمولاً در هسته سیستم عامل صورت می‌پذیرد.

۴-۱-۱-۳-۲- مانیتورها

یک مانیتور یک ساخت هم‌زمانی سطح بالا است که شامل داده‌ها و رویه‌های مورد نیاز جهت انجام عمل تخصیص منابع یا گروهی از منابع مشترک می‌باشد. فرآیندهای بسیاری ممکن است بخواهند در لحظات مختلف وارد مانیتور گردند، اما در مانیتور نیز **فقط یک فرآیند در هر لحظه اجازه ورود خواهد داشت.** فرآیندهای خواستار ورود به مانیتور هنگامی که مانیتور مشغول باشد، باید در انتظار بمانند. این انتظار **به طور خودکار به وسیله مانیتور مدیریت** می‌گردد.

داده‌های داخل مانیتور می‌توانند یا برای تمام رویه‌های مانیتور سراسری باشند و یا تنها رویه خاصی از آن‌ها به صورت محلی استفاده کند. تمام این داده‌ها فقط در مانیتور قابل دسترسی هستند. این ویژگی را **پنهان سازی اطلاعات** می‌نامند.

در صورتی‌که فرآیند فراخواننده مانیتور دریابد که منبع مورد نظرش پیشتر اختصاص یافته است، دستور Wait بر روی یک **متغیر شرطی** فراخوانی می‌شود تا باعث شود که فرآیند مورد نظر تا زمان آزاد شدن منبع مورد نظر خود **خارج از مانیتور در انتظار** باقی بماند. زمانی که یک متغیر شرطی داخل مانیتور تعریف می‌شود، صفی ایجاد می‌گردد که فرآیند فراخواننده Wait به این صف ملحق خواهد شد. دستور دیگری بنام Signal بر روی یک متغیر شرطی باعث می‌شود که فرآیند فراخواننده Signal، یک فرآیند منتظر از صف مربوط به متغیر شرطی مربوطه را خارج نموده و آن‌را وارد مانیتور نماید. فرآیندی که منبع را در اختیار دارد، با فراخوانی یکی از رویه‌های مانیتور منبع را به سیستم باز می گرداند. این رویه فراخوان، با دستور Signal اجازه می‌دهد یکی از فرآیندهای منتظر، منبع را در اختیار بگیرد فرآیندی که باعث فراخوانی دستور Signal می‌شود از مانیتور خارج می‌گردد.

چنان‌چه فرآیندی در انتظار منبع نباشد، این سیگنال فقط سبب خواهد شد که سیستم منبع را در اختیار بگیرد و اثر دیگری نخواهد داشت.

توجه کنید که برای فرآیندی که باعث فراخوانی دستور Signal می‌شود سه حالت تعریف و پیاده‌سازی شده است:

۱ـ ساختار مانیتور پاسکال در زبان برنامه‌نویسی هم‌زمان پاسکال:

در این ساختار بلافاصله بعد از دستور Signal دستور بازگشت از روتین مربوطه در مانیتور صادر می‌گردد، بدین معنی که روتین به پایان می‌رسد و هیچ دستور دیگری نباید در روتین نوشته شود.

۲ـ ساختار مانیتور Hoar در سیستم برنامه‌سازی هم‌زمان BASI (معنی هم‌زمانی Ben – ARI):

فرآیندی که دستور Signal را صادر کرده چنان‌چه داخل مانیتور کارش تمام نشده باشد باید بر روی صف جدیدی با اولویت بالا منتظر بماند (تا زمانی که فرآیند داخل مانیتور کارش تمام شود) بنابراین فرآیندی که داخل مانیتور است به محض تمام شدن کارش باید فرآیندی از صف جدید را در داخل مانیتور فعال نماید.

۳ـ ساختار مانیتور Mesa در زبان برنامه‌سازی سیستم Modula -3:

در این ساختار دستور اولیه notify (x) جایگزین Signal (x) شده است. وقتی فرآیندی در یک مانیتور notify (x) را اجرا می‌کند باعث می‌شود تا صف شرط x آگاه گردد، اما فرآیند علامت دهنده به‌کار خود ادامه می‌دهد.

فرآیند منتظر روی شرط x در زمان مناسبی در آینده، زمانی که مانیتور آزاد می‌شود وارد مانیتور می‌گردد و دستور بعد از wait (x) اجرا می‌گردد.

توجه داشته باشید که هیچ تضمینی نیست که قبل از این فرآیند، فرآیند دیگری زودتر وارد مانیتور نشود.

حالت سوم، اشکال حالت دوم را که باعث ایجاد دو تعویض متن برای فرآیند (در انتظار رفتن و از سر گرفتن فرآیند به هنگام آزاد شدن مانیتور) می‌شود را برطرف نموده است.

• تخصیص ساده منبع با استفاده از مانیتورها :

فرض کنید فرآیندهای چندی نیاز به دست‌یابی به یک منبع خاص داشته باشند. این منبع می‌تواند در هر لحظه فقط به وسیله یک فرآیند استفاده گردد. یک مانیتور ساده برای کنترل تخصیص و بازپس گیری چنین منبعی به صورت زیر است:

```
Monitor  ResourceAllocator;
  Var
      ResourceInUse : Boolean
      ResourceIsFree : Condition
Procedure GetResource
      { if (ResourceInUse) Wait(ResourceIsFree);
        ResourseInUse = True;}
Procedure ReturnResource
      {  ResourceInUse =False;
         Signal (ResourceIsFree);}
ResourceInUse =False;
End Monitor
```

این تخصیص دهنده منبع، دقیقاً شبیه یک سمافور باینری عمل می‌کند.

تابع GetResource همانند P یا Wait

تابع ReturnRosource همانندV یا Signal می‌باشند.

۴-۱-۱-۳-۳ پیام‌ها:

پیام‌ها یک مکانیزم ساده و مناسب جهت همگام سازی و ارتباط بین فرآیندها در محیط غیر متمرکز و توزیع شده می‌باشند. بسیاری از سیستم‌های عامل چند برنامگی از نوعی پیام‌های بین فرآیندها پشتیبانی می نمایند. ارسال و دریافت پیام‌ها یک شکل استاندارد ارتباط بین گره ها (Internode) در شبکه‌های کامپیوتری می‌باشند و افزودن این تسهیلات وظایف همگام سازی در ارتباط بین فرآیندها را میسر می‌سازد. به این دلیل پیام‌ها در سیستم‌های عامل توزیع شده خیلی متداول هستند.

در حقیقیت یک پیام مجموعه‌ای از اطلاعات است که بین فرآیندهای ارسال کننده و دریافت کننده مبادله می‌گردد.

به طور کلی قالب پیام قابل انعطاف و قابل تبادل توسط هر زوج خاصی از فرستنده، دریافت کننده می‌باشد. یک پیام توسط نوعش، طوش، فرمت‌های فرستنده و دریافت کننده اش و یک فیلد داده‌ها متمایز می‌گردد.

فرآیندهایی که بخواهند ارتباط برقرار نمایند، باید راهی برای اشاره کردن و نامیدن یکدیگر داشته باشند. فرآیندها می‌تواند از ارتباط مستقیم یا غیر مستقیم استفاده کنند.

• ارتباط مستقیم:

در شیوه ارتباط مستقیم هر فرآیند که بخواهد پیامی را ارسال کند و یا دریافت نماید، باید به وضوح و صریحاً ارسال کننده یا دریافت کننده، پیام را نام‌گذاری کند. اعمال Receive, Send به صورت زیر تعریف می گردند:

پیامی را به فرآیند A بفرست Send (A, massage)

پیامی را از فرآیند B دریافت کن Receive(B, massage)

• ارتباط غیر مستقیم:

در این مورد، پیام‌ها مستقیماً از فرآیند به گیرنده فرستاده نمی‌شود، بلکه به یک ساختمان داده مشترک که شامل بر صف‌هایی است که می‌توانند پیام‌ها را به طور دایمی نگهدارند، ارسال می‌گردد. به این صف‌ها معمولاً **صندوق پستی** می‌گویند، بنابراین برای ارتباط دو فرآیند، یکی پیامی را به صندوق پستی مناسب می فرستد و فرآیند دیگر آن پیام را از صندوق پستی بر می‌دارد.

Receive, Send در این شیوه به صورت زیر تعریف می گردند:

Send (MailBox, massage) پیام را به صندوق پستی بفرست

Receive (MailBox, massage) پیام را از صندوق پستی دریافت کن

رابطه بین فرستنده ها و گیرنده ها می‌تواند یک به یک، چند به یک، یک به چند و یا چند به چند باشد.

رابطه یک به یک برقراری یک پیوند ارتباطی بین دو فرآیند را میسر می‌کند.

رابطه چند به یک برای مدل Client – Server مفید است که یک فرآیند به تعدادی از فرآیندها خدمت می‌کند.

رابطه یک به چند رابطه یک فرستنده و گیرنده های متعدد را میسر می‌نماید و برای کاربردهایی مفید است که پیام باید برای مجموعه‌ای از فرآیندها پخش شود.

نکته‌ای که باید مورد توجه قرار گیرد، مالکیت صندوق پستی است. در مورد رابطهClient–Server، معمولاً صندوق پستی به وسیله یک فرآیند گیرنده ایجاد شده و در تملک آن است. بنابراین وقتی آن فرآیند از بین می‌رود، صندوق پستی نیز از بین می‌رود. در مورد صندوق پستی عمومی، ممکن است سیستم عامل، صندوق را ایجاد نماید.

تبادل پیام بین دو فرآیند متضمن سطحی از همگام سازی بین آنهاست، تا زمانی که پیامی توسط فرآیندی فرستاده نشده باشد، دریافت کننده نمی‌تواند آن‌را دریافت نماید. لازم است مشخص کنیم بعد از این‌که فرآیندی Send یا Receive را صادر کرد، چه اتفاقی برایش می‌افتد.

وقتی فرآیندی دستور Send را اجرا می‌کند دو امکان وجود دارد: یا فرآیند فرستنده تا دریافت پیام مسدود است یا نیست. همین طور هنگامی که فرآیندی دستور Receive را اجرا می‌کند یا تا رسیدن یک پیام مسدود می‌ماند و یا این‌که بدون توجه به تلاشی که برای دریافت پیام شده است، به اجرای خود ادامه می‌دهد.

بنابراین فرستنده و گیرنده هر دو می‌توانند مسدود شونده و یا غیرمسدود، باشند.

• همگام سازی و ارتباط بین فرآیندها با پیام‌ها:

تبادل پیام خالی بدون هیچ گونه فیلد داده مابین دو فرآیند معادل سمافور است. یک فرآیند با ارسال پیام، یک **Signal** را به فرآیند دریافت کننده می‌دهد، و همان اثری را دارد که یک فرآیند یک عمل Signal بر روی سمافور را اجرا می‌کند و فرآیند دیگر (دریافت کننده) عمل Wait را روی همان سمافور اجرا نماید. در این حالت صندوق پستی، متناظر با همان سمافور می‌باشد.

کد زیر یک راه استفاده از تبادل پیام برای اعمال انحصاری متقابل را نشان می‌دهد.

فرض بر این است که دستور Receive به صورت مسدود شونده و Send به صورت غیرمسدود استفاده شده اند. مجموعه‌ای از فرآیندهای هم‌زمان در صندوق پستی Mutex شریک‌اند و می‌توانند از این صندوق پستی برای ارسال پیام استفاده نمایند.

```
Repeat
Receive (Mutex,msg)
   Critical Section
Send ( Mutex , msg )
Remainder Section
Forever
```

صندوق پستی Mutex، حاوی تنها یک پیام تهی می‌باشد. فرآیندی که می خواهد وارد بخش بحرانی خود شود، ابتدا سعی می‌کند پیامی دریافت کند. اگر صندوق پستی خالی باشد، در انتظار قرار می‌گیرد. هنگامی که فرآیندی پیام را دریافت می‌نماید بخش بحرانی خود را انجام می‌دهد و سپس پیام را به صندوق پستی برمی گرداند. در نتیجه این پیام نقش نشانه‌ای را بازی می‌کندکه از فرآیندی به فرآیند دیگر منتقل می‌شود.

در این راه حل فرض بر این است که اگر بیش از یک فرآیند عمل دریافت را به صورت هم‌زمان انجام دهند، در این‌صورت اگر پیامی‌باشد، تنها به یک فرآیند داده می‌شود و بقیه فرآیندها در انتظار باقی می‌مانند.

اگر صندوق پستی خالی باشد، تمام فرآیندها در انتظار می‌مانند. موقعی که پیامی فراهم می‌شود، تنها یکی از فرآیندهای منتظر، فعال شده و پیام به همان فرآیند داده می‌شود.

۴ - ۲ مسایل کلاسیک هم‌زمانی :

در ذیل، تعدادی از مسایل گوناگون هم‌زمانی را مطرح می‌نماییم و با استفاده از سمافورها و مانیتورها نشان می‌دهیم که چگونه مسایل هم‌زمانی آن‌ها سنکرون می‌شوند.

۴-۲-۱- مساله تولید کننده و مصرف کننده:

فرض کنید که تولید کننده داده‌هایی را یک به یک تولید و در بافر قرار می‌دهد و مصرف کننده اقلام تولید شده را یکی یکی از بافر بر می‌دارد. در هر زمان تنها یک فرآیند (با مصرف کننده و یا تولید کننده) می‌تواند به بافر دسترسی داشته باشد. فرض بر این باشد که بافر محدود و شامل آرایه‌ای خطی از عناصر است. به سه طریق مساله فوق را در ذیل حل می‌نماییم.

۱ ـ روش سمافورها

از سه سمافور به شرح ذیل استفاده می‌نماییم.

سمافور شمارشی Full ـ تعداد خانه های پر بافر را می شمارد.

سمافور شمارشی Empty تعداد خانه های خالی بافر را می شمارد.

سمافور باینری Mutex ـ این سمافور تضمین می‌کند که در آن واحد مصرف کننده و تولیدکننده به بافر دسترسی نداشته باشند.

در آغاز Full برابر صفر و Empty برابر تعداد خانه های بافر (N) و Mutex نیز به ۱ مقدار دهی می‌شود.

کد مربوط به فرآیند تولید کننده و مصرف کننده به شکل ذیل نوشته می‌شوند:

کد مصرف کننده

```
Repeat
    Wait(Full)
    Wait(Mutex)
        Remove Item from Buffer
    Signal (Mutex)
    Signal (Empty)
        Consume Item
Forever
```

کد تولید کننده

```
Repeat
    Produce an item
    Wait(Empty)
    Wait(Mutex)
        Add Item to Buffer
    Signal (Mutex)
    Signal (Full)
Forever
```

۲ ـ روش مانیتور

از دو متغیر شرطی Full و Empty در مانیتور ذیل استفاده می‌شود.

زمانی که بافر خالی است، فرآیند فراخواننده مانیتور با دستور Wait بر روی متغیر شرطی Empty در انتظار باقی می‌ماند.

زمانی که بافر پر است، فرآیند فراخواننده مانیتور با دستور Wait بر روی متغیر شرطی Full در انتظار باقی می‌ماند.

متغیر صحیح و مثبت Counter به‌عنوان شمارنده تعداد داده‌های پر در بافر مورد استفاده قرار می گیرد و به مقدار اولیه صفر، مقدار دهی می‌گردد. حداکثر تعداد داده‌های داخل بافر N فرض می‌شود.

مانیتور مربوط به مساله تولید کننده و مصرف کننده به شکل ذیل کد می‌گردد:

```
Monitor ProducerandConsumer
   Var
      Counter : Integer
   Full, Empty : Condition

Procedure Append (Item)
   { If (Counter==N) Wait(Full);
     Add Item to Buffer
     Counter ++;
     If (Counter ==1) Signal (Empty);}

Procedure Remove (Item)
     { If (Counter ==0) Wait(Empty);
       Remove Item From Buffer
       Counter --;
       If (Counter == N-1) Signal(Full);
     }
  { Counter=0}
End Monitor
```

کد مربوط به فرآیند تولید کننده و مصرف کننده به‌شکل زیر خواهد بود :

فرآیند تولید کننده

```
Repeat
Produce Item
Producer and Consumer. Append (Item)
Forever
```

/* روتین Append در مانیتور مربوطه فراخوانده می‌شود. */

فرآیند مصرف کننده

```
Repeat
Producer and Consumer. Romove ( Item)
Consume Item
Forever
```

/* روتین Remove در مانیتور مربوطه فرا خوانده می‌شود. */

۳- روش پیام‌ها

در این راهکار ممکن است تولید کنندگان و مصرف کنندگان متعددی در یک سیستم توزیع شده موجود باشند.

تولید کننده با تولید داده‌ها، آن‌ها را به عنوان پیام به صندوق پستی مصرف کننده می فرستند تا هنگامی که حداقل یک پیام در صندوق وجود دارد، مصرف کننده می‌تواند مصرف نماید. بنابراین صندوق پستی مصرف کننده مثل یک بافر عمل می‌کند که داده‌ها در بافر به صورت صفی از پیام‌ها سازماندهی شده اند.

اندازه بافر یک تعداد محدود فرض می‌شود. در ابتدا صندوق پستی Producer با تعدادی پیام تهی معادل ظرفیت بافر پر شده است. تعداد پیام‌های Producer با هر تولید کاهش و با مصرف افزایش می‌یابد.

کد مربوط به رویه‌های تولید کننده و مصرف کننده به شکل ذیل می‌باشد:

```
Procedure Producer;
   Repeat
      Receive (Producer,Pmsg)  /× پیام تهی دریافت کن ×/
         Produce Item and Insert into Pmsg
      Send (Consumer,Pmsg)  /× داده تولید شده را به صندوق پستی مصرف کننده بفرست ×/
   Forever
Procedure Consumer
   Repeat
      Receive (Consumer,Cmsg)
         Consume Cmsg
      Send (Producer,Null)  /× پیام تهی به تولید کننده بفرست ×/
   Forever
```

در دستورالعمل Receive چنان‌چه پیام موجود نباشد، فرآیند در انتظار قرار می گیرد.

۴-۲-۲- مساله خوانندگان و نویسندگان

مساله خوانندگان و نویسندگان به صورت زیر تعریف شده است. پرونده ای وجود دارد که بین تعدادی از فرآیندها مشترک است. تعدادی از فرآیندها هستند که از این پرونده فقط می‌خوانند (خوانندگان) و تعدادی از فرآیندها در این پرونده فقط می نویسند (نویسندگان). شرایط زیر باید برقرار باشد:

۱- هر تعداد از خوانندگان می‌توانند به صورت هم‌زمان پرونده را بخوانند.

۲- در هر زمان تنها یک فرآیند ممکن است در این پرونده بنویسد.

۳- هنگامی که نویسنده ای در حال نوشتن است، هیچ خواننده‌ای نمی‌تواند پرونده را بخواند.

سه حالت زیر را برای این مساله مورد بررسی قرار می‌دهیم:

●خوانندگان اولویت دارند

در این حالت لازم می دارد که هیچ خواننده‌ای منتظر نماند، مگر این‌که نویسنده ای قبلاً به پرونده مشترک دسترسی مجاز داشته باشد. به عبارت دیگر هیچ خواننده‌ای منتظر خوانندگان دیگر نمی‌ماند، صرفاً به دلیل این‌که نویسنده ای در حال انتظار است. بنابراین نویسندگان ممکن است دچار محرومیت یا **قحطی زدگی** شوند.

با توجه به سمافورهای Mutex، Wrt، که به مقدار 1 و متغیر Integer، readcount که به مقدار صفر آغاز دهی می‌شوند، کد زیر را در نظر بگیرید.

سمافور Mutex، دسترسی به readcount را در موقع به هنگام سازی، انحصاری می‌سازد. متغیر readcount تعداد فرآیندهایی را که در حال حاضر مشغول خواندن پرونده هستند، می شمارد. سمافور Wrt به عنوان سمافور انحصار متقابل برای نویسندگان به کار می‌رود.

کد فرآیند نویسندگان

```
Wait (Wrt)
  Writing to file
Signal (Wrt)
```

کد فرآیند خوانندگان

```
Wait (Mutex)
readcount = readcount +1;
if (readcount=1) Wait (Wrt);
Signal (Mutex)
   reading from file
Wait (Mutex)
readcount = readcount-1;
if (readcount=0 ) Signal (Wrt);
Signal (Mutex)
```

- نویسندگان اولویت دارند

در این حالت لازم می دارد که به محض این‌که نویسنده ای آماده شد، عمل نوشتن خود را حتی المقدور هر چه زودتر به انجام رساند. به عبارت دیگر اگر نویسنده ای برای دسترسی به پرونده منتظر باشد خوانندگان جدید نباید شروع به خواندن کنند. در این حالت **خوانندگان قحطی زده** می‌شوند.

فرآیند نویسندگان و خوانندگان به شکل ذیل همگام می‌شوند:

برای **فرآیند نویسندگان** سمافورها و متغیرهای ذیل را تعریف می نماییم:

- سمافور باینری wsem برای اعمال انحصار متقابل در هنگام به هنگام سازی پرونده
- سمافور باینری rsem به منظور کنترل ورود خوانندگان در صورتی که حداقل یک نویسنده مشغول به هنگام سازی پرونده باشد.
- متغیر Writecount تعداد فرآیندهایی را که در حال حاضر مایل به دسترسی به پرونده برای به هنگام سازی می‌باشند و وضع rsem را کنترل می نمایند.
- سمافور باینری y به هنگام کردن Writecount را کنترل می نمایند.

برای **خوانندگان** یک سمافور باینری اضافی لازم است، زیرا نباید تشکیل صف طولانی روی rsem مجاز باشد، در غیر این‌صورت نویسندگان قادر به عبور از این صف نیستند. بنابراین تنها یک خواننده مجاز به تشکیل صف روی rsem است، خوانندگان دیگر روی سمافور باینری z (قبل از انتظار برای rsem) صف می بندند.

- سمافور باینری x به هنگام کردن readcount را کنترل می‌نماید.

کد فرآیند نویسندگان

```
Wait (y)
  Writecount ++;
   If (Writecount ==1) Wait(rsem);
Signal (y)
Wait (wsem)
   writing to file
Signal (wsem)
Wait (y)
    Writecount –;
     If (writecount==0 ) Signal (rsem);
Signal (y)
```

کد فرآیند خوانندگان

```
Wait (z)
   Wait (rsem)
     Wait(x)
        readcount= readcount+1;
        If (readcount==1)Wait(wsem);
     Signal(x)
   Signal (rsem)
Signal (z)
Reading from file
Wait(x)
  Readcount = readcount-1;
  If (readcount== 0 )Signal (wsem);
Signal (x)
```

- خوانندگان و نویسندگان بدون قحطی‌زدگی به کمک مانیتور

در این حالت موارد ذیل اعمال خواهد شد:

ـ در هر لحظه فقط یک نویسنده می‌تواند فعال باشد.

ـ هنگامی که یک نویسنده فعال باشد هیچ خواننده‌ای فعال نخواهد بود.

ـ در جایی که هیچ فرآیندی در حال نوشتن نباشد و هیچ فرآیند نویسنده ای برای نوشتن در انتظار به سر نبرد، خواننده جدیدی قادر است که به خواندن ادامه دهد که این شرط برای جلوگیری از تاخیر نامحدود خوانندگان لازم و از اهمیت خاص برخوردار است.

ـ چنان‌چه فرآیندی در حال خواندن باشد، فرآیندهای جدید برای خواندن فقط با شرط فوق الذکر هم‌زمان با فرآیند در حال خواندن می‌توانند از پرونده بخوانند. که این شرط برای جلوگیری از تاخیر نامحدود نویسندگان لازم می‌باشد.

راه حل مساله فوق را به کمک مانیتور در ذیل توضیح می‌دهیم.

مانیتور ReadersandWriters را که شامل۴ رویه می‌باشد، می‌تواند برای یک دسترسی کنترل شده به پرونده مشترک به کار گرفته شود.

چهار رویه به شرح ذیل می‌باشند:

۱- BeginReading

هنگامی که یک خواننده تمایل به خواندن داشته باشد، این رویه فرا خوانده می‌شود. در این رویه یک خواننده جدید قادر است به خواندن ادامه دهد تا جایی که هیچ فرآیندی در حال نوشتن نباشد و هیچ فرآیند نویسنده ای برای نوشتن در انتظار به سر نبرد. این رویه با اعلام کردن از طریق متغیر ReadingAllowed پایان کار خود را اعلام می‌کند و اجازه می‌دهد که خواننده منتظر دیگری، خواندن را آغاز نماید.

متغیر Integer، "Readers" تعداد خوانندگان را مشخص می‌کند. در این رویه زمانی که خواننده جدید مجوز ادامه خواندن بگیرد یک واحد به تعداد Readers اضافه می‌شود.

۲- FinishReading

خواننده‌ای که کار خواندن خود را به پایان رساند، این رویه را فرا می‌خواند. این رویه سبب می‌گردد که تعداد خواننده ها یک واحد کاهش یابد. سر انجام این کاهش، منجر به صفر شدن تعداد خواننده ها می‌شود که در این لحظه اعلان WritingAllowed اعلام می‌شود و به یک نویسنده منتظر، اجازه نوشتن می‌دهد.

۳- BeginWriting

هنگامی که نویسنده ای مایل به نوشتن باشد این رویه را فرا می‌خواند.

از آنجایی‌که یک نویسنده باید دارای یک دسترسی کاملاً منحصر به فرد باشد، اگر خواننده یا خوانندگانی موجود باشند یا نویسنده دیگری فعال باشد، این نویسنده باید منتظر متغیر شرطی WritingAllowed بماند. هنگامی که نویسنده امکان ادامه کار یابد، متغیر منطقی SomeoneWriting مقدار " True" می گیرد. که این امر سبب دور نگه داشتن سایر خوانندگان و نویسندگان خواهد بود.

۴- FinishWriting

هنگامی که نویسنده ای عمل به هنگام سازی پرونده را به اتمام می رساند، این رویه را فرا می‌خواند که سبب می‌شود متغیر منطقی SomeoneWriting به مقدار " False"، تغییر یابد تا به سایر فرآیندها امکان ورود دهد. سپس باید به فرآیند منتظر دیگری اعلام کند که به کار خود ادامه دهد.

آیا این فرآیند نویسنده باید اولویت را به یک خواننده بدهد یا به یک نویسنده؟

اگر اولویت به نویسنده منتظر داده شود این امکان وجود دارد که خواننده‌ای منتظر برای یک دنباله از نویسندگان با یک تاخیر نامحدود روبرو شود. بنابراین ابتدا بررسی می‌شود که آیا خواننده منتظری وجود دارد یا خیر. اگر وجود داشت، خواننده منتظر به کار خود ادامه می‌دهد. اگر خواننده منتظری موجود نباشد، در آن صورت نویسنده منتظر اجازه پیشروی خواهد گرفت.

حال کد مانیتور مربوطه همراه با چهار رویه مذکور در ذیل نمایش داده می‌شود.

```
Monitor  ReadersandWriters
Var
   Readers : Integer
   SomeoneWriting : Boolean
   ReadingAllowed , WritingAllowed : Condition

Procedure  BeginReading
   { If (SomeWriting  or Queue(WritingAllowed))
      Wait (ReadingAllowed);
   Readers = Readers+1;
   Signal (ReadingAllowed);
   }

Procedure  FinishReading
   {  Readers = Readers – 1;
      If (Readers==0)  Signal (WritingAllowed);
   }
Procedure  BeginWriting

      { If (Readers > 0 or  SomeoneWriting )
      Wait (WritingAllowed);
      SomeoneWriting = True;
   }
Procedure  FinishWriting
   {  SomeoneWriting =False;
      If (Queue (ReadingAllowed))    Signal (ReadingAllowed);
      Else  Signal (WritingAllowed);
   }
{Readers=0  ;   SomeoneWriting = False ; }
  End Monitor
```

• خوانندگان و نویسندگان بدون قحطی‌زدگی به کمک سمافور

در این حالت کد فرآیند خوانندگان و نویسندگان همانند حالتی است که فرآیند خوانندگان اولویت دارند، فقط با این اختلاف که ورود نویسندگان و خوانندگان با یک سمافور باینری Mutexi کنترل می‌شود.

بدین‌معنی که اگر خوانندگان مشغول خواندن باشند و نویسنده‌ای از راه برسد با اجرای wait (Mutexi) اجازه نخواهد داد که خوانندگان جدید وارد شوند و با اتمام کار خوانندگان قدیمی، نویسنده اجازه ورود برای نوشتن را پیدا می‌نماید.

در حالی که نویسنده‌ائی مشغول نوشتن در فایل باشد، چنان‌چه خواننده‌ایی از راه برسد در صف Mutexi قرار می‌گیرد wait (Mutexi) و بعد از اتمام کار نویسنده، کار خواننده منتظر با صدور دستور signal (Mutexi) شروع می‌شود، پس بدین‌ترتیب هیچ‌گاه مشکل قحطی‌زدگی برای نویسندگان و خوانندگان رخ نمی‌دهد.

کد فرآیند نویسندگان

```
Wait (Mutexi)
Wait (wrt)
   Writing to file
Signal (wrt)
Signal (Mutexi)
```

کد فرآیندگان خوانندگان

```
wait (Mutexi)
wait (Mutex)
readcount = readcount + 1
if (read count==1) wait (wrt);
signal (Mutex)
signal (Mutexi)
   Reading from file
Wait (Mutex)
   readcount = readcount-1
   If (readcount =0) signal (wrt);
   Signal (Mutex)
```

۴-۲-۳- مساله فیلسوفان خورنده

در سال ۱۹۶۵ دیجسترا یک مساله هم‌زمانی را طرح و حل کرد و نام آن را " **مساله فیلسوفان خورنده**" نامید. این مساله یک مساله کلاسیک هم‌زمانی محسوب می‌شود و به این صورت مطرح می‌شود:

۵ فیلسوف دور یک میز مدور نشسته اند، هر فیلسوف یک بشقاب اسپاگتی دارد. هر فیلسوف به دو چنگال برای خوردن آن نیاز دارد. بین هر دو بشقاب یک چنگال وجود دارد. زندگی یک فیلسوف از دو پریود متناوب خوردن و فکر کردن تشکیل شده است. وقتی که یک فیلسوف گرسنه می‌شود، سعی می‌کند که چنگال های سمت راست و چپش را یکی یکی و با هر ترتیب ممکن بردارد. اگر موفق شود که دو چنگال را به دست آورد، برای مدتی غذا می خورد و سپس چنگال‌ها را پایین می‌گذارد و به فکر کردن ادامه می‌دهد.

سوال اصلی این است که آیا می‌توانید برنامه ای برای هر فیلسوف طراحی کنید که کاری را که می خواهد بتواند انجام دهد و هیچ گاه به مشکلی برنخورد؟

یک راه حل ساده آن است که هر چنگال را با یک سمافور نشان دهیم. فیلسوف سعی دارد چنگال را با اجرای عمل Wait بر روی آن سمافور، به دست آورد و چنگال را با عمل Signal برروی سمافور مناسب، رها سازد.

کد زیر را در نظر بگیرید:

```
Repeat
Wait (Fork [i])
Wait (Fork [(i+1) mod 5])
.
eat
.
Signal (Fork [i])
Signal (Fork [(i+1) mod 5]
.
think
Forever
```

کد فوق برای فیلسوف i در نظر گرفته شده است که در این کد متغیر مشترک زیر

Var Fork : array [o:4] of Semaphore

تعریف شده است. هر عنصر آرایه Fork به مقدار "1" آغاز دهی می‌شود. متاسفانه این راه حل **غلط** است، زیرا فرض کنید کلیه فیلسوفان هم‌زمان گرسنه شوند و هر یک چنگال سمت چپشان را به‌دست آورند. کلیه عناصر آرایه Fork اکنون صفر می‌باشند. وقتی فیلسوفی سعی در برداشتن چنگال سمت راستش می‌نماید، برای همیشه به تاخیر می افتد و هیچ پیشرفتی در کارش حاصل نمی‌گردد و **بن‌بست** رخ می‌دهد.

یک راه حل بهبود یافته نسبت به کد فوق که منجر به بن‌بست نشود، این است که ۵ دستور اول را با یک سمافور باینری حفاظت نماییم. هر فیلسوف قبل از شروع به برداشتن چنگال ها، Wait را بر روی یک سمافور باینری (Mutex) انجام می‌دهد و بعد از زمین گذاشتن چنگال‌ها Signal را بر روی Mutex انجام می‌دهد. در این حالت فقط یک فیلسوف در هر لحظه می‌تواند مشغول غذا خوردن باشد. اما با پنج چنگال موجود باید بتوانیم به دو فیلسوف اجازه غذا خوردن هم‌زمان را بدهیم.

به عنوان رویکردی دیگر می‌توان فردی را در نظر گرفت که در هر زمان تنها 4 فیلسوف را به غذاخوری راه می‌دهد. اگر حداکثر 4 فیلسوف نشسته باشند، حداقل یک فیلسوف به دو چنگال دست می‌یابد. این راه‌حل بدون بن‌بست و گرسنگی است.

راه حلی را که در ذیل ارایه می‌دهیم، **صحیح است** و علاوه بر آن اجازه می‌دهد که حداکثر توازی را برای تعداد دلخواهی فیلسوف داشته باشیم. این راه حل از یک آرایه به نام state استفاده می‌کند که وضعیت جاری فیلسوف را در یکی از سه حالت خوردن، فکر کردن و گرسنگی، نگهداری می‌کند.

یک فیلسوف فقط در صورتی به حالت خوردن می‌رود که هیچ یک از همسایگانش در حال خوردن نباشد. علاوه بر این، برنامه از آرایه‌ای از سمافورها (S[i]) استفاده می‌کند (به ازای هر فیلسوف یک سمافور). بنابراین فیلسوف‌های گرسنه می‌توانند در صورت مشغول بودن چنگال‌های مورد نیازشان، به انتظار روند. توجه کنید که هر فرآیند (فیلسوف) رویه Philosepher را به عنوان کد برنامه اصلی خود اجرا می‌نمایند. رویه‌های دیگر رویه‌های معمولی می‌باشند که رویه Philosepher آن‌ها را فرا خوانند.

```
Procedure Philosepher(i)
  Repeat
     Take-Fork(i);   /× فراخوانی برای گرفتن چنگال‌ها ×/
     Eat
     Put-Fork(i);  /× فراخوانی برای زمین گذاشتن چنگال‌ها ×/
     Think
  Forever

Procedure Take-Fork(i)
{
Wait(mutex)
   State[i]=Hungry
   Test(i)
Signal(mutex)
Wait(S[i])
}
Procedure  Put-Fork(i)
{
Wait(mutex)

     State[i]=Thinking
   Test((i-1)mod 5);  /× آیا فیلسوف سمت چپ می‌تواند خوردن را شروع کند. ×/
   Test((i+1) mod 5);  /× آیا فیلسوف سمت راست می‌تواند خوردن را شروع نماید.؟ ×/
```

```
Signal(mutex)
}
Procedure Test(i)
{
If (State[i] == Hungry and State[(i-1) mod 5] <> Eating and
        State[(i+1)mod 5] <> Eating)
    { State[i] = Eating;  Signal(S[i]); }
}
```

راه حل مساله فیلسوف خورنده به کمک مانیتور همانند راه حل قبلی از آرایه‌ای به نام State که وضعیت جاری فیلسوف‌ها را در سه حالت خوردن، فکر کردن، گرسنگی نشان می‌دهد، استفاده می‌نماید و به جای سمافورهای S[i]در راه حل قبلی از متغیر شرطی S که به صورت آرایه تعریف می‌شود، کمک گرفته خواهد شد.

توزیع چنگال‌ها، توسط مانیتور Philosepher کنترل می‌گردد. هر فیلسوف قبل از شروع به خوردن، رویه Take-Fork را داخل مانیتور فراخوانی می‌نماید که ممکن است منتج به مسدود شدن فرآیند فیلسوف گردد. چنان‌چه هر دو چنگال به‌دست آید، فیلسوف می‌تواند بخورد. به دنبال آن بعد از اتمام عمل خوردن، فیلسوف رویه Put-Fork را فرا می‌خواند و می‌تواند شروع به اندیشیدن نماید.

بنابراین فیلسوف بایستی توالی زیر را انجام دهد.

```
Philosepher. Take-Fork(i)
  Eat
Philosepher. Put-Fork(i)
  Think
```

کد مانیتور Philosepher به قرار ذیل می‌باشد:

```
Monitor Philosepher
 Var
    State : array[0:4] of ( Thinking, Hungry, Eating)
    S     : array[0:4] of Condition
 Procedure Take-Fork(i)
   {
    State [i] =Hungry
    Test(i)
    If (State[i] <> Eating) Wait(S[i]);
   }
 Procedure Put-Fork(i)
   {
    State[i]=Thinking
    Test ((i-1) mod 5);
    Test ((i+1) mod 5);
   }
 Procedure Test(i)
   {
   If (State[i]== Hungry and State[(i-1) mod 5] <> Eating and
                State[(i+1) mod 5] <> Eating)
     { State[i]=Eating;  Signal(S[i]); }
   }
 { For i=0 to 4  ( State[i] = Thinking)  }
 End Monitor
```

۴ـ۲ـ۴ـ مساله آرایشگاه

یکی دیگر از مسایل کلاسیک، در آرایشگاه اتفاق می‌افتد. فرض کنید آرایشگاه دارای یک صندلی آرایشگری و n صندلی برای مشتریان منتظر و یک آرایشگر باشد. چنان‌چه مشتری وجود نداشته باشد، آرایشگر به خواب می‌رود (منتظر می‌ماند). وقتی مشتری از راه می‌رسد، آرایشگر را بیدار می‌نماید. چنان‌چه مشتری از راه برسد و آرایشگر مشغول کوتاه کردن سر مشتری دیگری باشد، در صورتی که صندلی خالی باشد مشتری بر روی صندلی می‌نشیند در غیر این‌صورت آرایشگاه را ترک می‌نماید.

در این مساله هماهنگی بین اعمال آرایشگر و مشتریان نشان داده می‌شود.

این راه‌حل از دو سمافور شمارشی Barbers, Customers و یک سمافور باینری Mutex و یک متغیر به نام waiting استفاده می‌نماید.

- سمافور شمارشی customers تعداد مشتریان منتظر را می‌شمارد که این تعداد در متغیر waiting قرار می‌گیرد.
- سمافور شمارشی Barbers تعداد آرایشگر منتظر مشتری را می‌شمارد که در این مساله 0 یا 1 می‌باشد.
- سمافور باینری Mutex برای ایجاد انحصار متقابل به‌کار گرفته می‌شود.

با توجه به کد ذیل دو رویه Customer , baber برای آرایشگر و مشتری نوشته شده است.

در ابتدا آرایشگر برای شروع به کار رویه barber را اجرا می‌نماید که بر روی سمافور customers به انتظار می‌رود تا مشتری از راه برسد. زمانی که مشتری از راه می‌رسد رویه customer اجرا می‌شود که ابتدا به‌صورت انحصاری تعداد مشتریان منتظر را چک می‌کند (آیا تعداد مشتریان کمتر از تعداد صندلی است)، چنان‌چه صندلی خالی نباشد، آرایشگاه را ترک می‌نماید اما در غیر این‌صورت نیز waiting را یک واحد افزایش می‌دهد و با signal بر روی سمافور customer آرایشگر را بیدار می‌نماید، (توجه کنید چنان‌چه آرایشگر مشغول کوتاه کردن سر مشتری دیگری باشد عمل signal باعث می‌شود که ارزش عددی customers یک واحد اضافه گردد) سپس Mutex را آزاد می‌نماید. چنان‌چه آرایشگر مشغول کوتاه کردن سر مشتری دیگری باشد، مشتری با اجرای دستور wait بر روی barber به انتظار می‌رود در غیر این‌صورت برای کوتاه کردن سر آماده می‌شود.

در این حالت توجه کنید که در رویه barber چنان‌چه barber منتظر باشد، هم اکنون بیدار می‌شود با یک واحد کاهش در تعداد مشتریان منتظر به‌صورت انحصاری آماده کوتاه کردن سر مشتری منتظر (و یا از راه رسیده) می‌گردد. وجود حلقه در ارتباط با رویه barber باعث می‌شود که مشتری بعدی را در صورت حاضر بودن بپذیرد.

در رویه customer حلقه‌ای وجود ندارد زیرا مشتری، پس از کوتاه شدن موی سرش از آرایشگاه خارج می‌شود.

کد مساله آرایشگاه

```
Semaphore customers=0;
Semaphore barbers =0;
Semaphore Mutex=1;
Int waiting = 0 ;
Proc barber ;
        While (TRUE)
        {Wait (customers);
            Wait (Mutex);
             Waiting = waiting-1;
             Signal (barbers);
             Signal (Mutex);
             Cut-hair;
        }
Proc customer;
            Wait (Mutex);
            If (waiting<n)
                {Waiting = waiting + 1 ;
                Signal (customers);
                Signal (Mutex);
                Wait (barbers);
                Get-haircut;}
}               Else Signal (Mutex);
```

تست‌های مربوط به نواحی بحرانی

۱ ـ آیا کد زیر می‌تواند راه حل مناسبی برای دو پردازش همروند باشد؟

```
{ Bool lock ;
  Bool key[2] ;
  PROC (i) ;
  Int (i) ;
 { While (true)
      { computation ;
        key [i] = true ;
      While (key[i])  swap  (key[i] ,lock) ;
        CS
        Lock= false;
        }
  }
 lock= false ;
 key[1] = false ;
 key[2] = false ;
}
```

الف) مشکل بن‌بست ایجاد می‌کند.

ب) مشکل ورود هم‌زمان به ناحیه بحرانی را باعث می‌شود.

ج) ممانعت دو جانبه ایجاد می‌شود و راه حل مناسبی برای دو پردازش همروند می‌باشد.

د) گاهی مشکل بن‌بست و گاهی مشکل ورود هم‌زمان دو پردازش ایجاد می‌شود.

۲ ـ آیا کد زیر می‌تواند راه حل مناسبی برای تولید کننده و مصرف کننده با بافر نا محدود محسوب شود؟

مقدار اولیه n، delay و ME به ترتیب صفر، صفرو یک می‌باشد.

```
Bsemaphore   ME , delay ;
  Int  n ;
```

کد تولید کننده

```
While (true)
   { P(ME)
     n=n+1

   Buff <=== item
   if(n==1)  v(delay)
   v(ME) }
```

کد مصرف کننده

```
 P (delay) ;
While (true)
   { P(ME) ;

   n=n-1
   item <==== Buff
   if (n==0)  p(delay);
   v(ME)   }
```

الف) کدهای فوق راه حل مناسبی برای مصرف کننده و تولید کننده می‌باشند.

ب) احتمال دارد مصرف کننده و تولید کننده هر دو در انتظار یکدیگر باقی بمانند.

ج) احتمال دارد ارزش n منفی شود.

د) احتمال دارد هر دو به ناحیه Buff هم‌زمان دسترسی یابند.

۳ـ آیا کد زیر می‌تواند راه حل مناسبی برای مشکل Race Condition باشد؟ چرا؟ (فقط بین دو فرآیند همروند)

```
int Turn ;
bool flag[2] ;
PROC (i)
int i ;
{ While (TRUE)
    { non Critical Section
      flag[i] =true ;
      While (Turn ==(i+1) mod 2)
        { Turn=i ;
          While (flag [(i+1) mod 2]);
        }
      Critical Section
      flag [i] = false;
    };
  };
Turn = 1 ;
flag[0] = false ;
flage [1] = false ;
}
```

الف) مشکل ورود هم‌زمان به ناحیه بحرانی را باعث می‌شود.

ب) ممانعت دوجانبه را تضمین می‌کند، اما احتمال بن‌بست دارد.

ج) مشکل ورود نوبتی به ناحیه بحرانی ایجاد می‌شود.

د) همیشه ممانعت دوجانبه ایجاد می‌گردد.

۴ـ آیا کد زیرپیاده سازی قابل قبولی از عمل P یا Down بر روی سمافورn تایی می‌تواند باشد یا خیر؟چرا؟
n عدد صحیح بزرگ‌تر یا مساوی صفر می‌باشدو روالSwap پارامترهای از نوع منطقی خود را به صورت بخش ناپذیر جابه‌جا می‌نماید.

```
a=0 ;
s=n ;
   P(s)
   {Bool b ;
    b=1 ;
    while (b)  swap(a,b) ;
    while (s<=0);
    s=s-1;
    swap (a,b);
   }
```

الف) در تمام مواقع صحیح است و کد قابل قبولی از عمل P بر روی سمافور n تایی می‌باشد.

ب) غیر از مشکل Busy Waiting، پیاده سازی فوق قابل قبول می‌باشد.

ج) مشکل بن‌بست ممکن است رخ دهد.

د) مشکل ورود به ناحیه بحرانی برای تغییر مقدار s ایجاد می‌شود.

۵ـ آیا کد زیر می‌تواند راه حل مناسبی برای ناحیه بحرانی باشد؟ مقدار اولیه lock= false

```
Bool lock ;
while TSL(lock) wait ;
        Critical Section
lock= false ;
Signal (Block process) ;
```

الف) احتمال دارد یک پردازش همیشه در انتظار ورود به ناحیه بحرانی قرار گیرد.

ب) در همه موارد کد فوق ممانعت دو جانبه ایجاد می‌نماید. و احتمالاً یک پردازش به بن‌بست می‌رود.

ج) در برخی از موارد کد فوق ممانعت دو جانبه ایجاد نمی‌کند.

د) می‌تواند راه حل مناسبی برای ناحیه بحرانی بین دو پردازش همروند باشد.

۶ـ آیا کد زیر می‌تواند راه حل مناسبی برای تولید کننده و مصرف کننده با بافر نامحدود محسوب شود؟ مقدار اولیه delay صفر ومقدار اولیه ME یک و مقدار اولیه n صفر می‌باشد.

```
Bsemaphore   ME,dely;
Int  n;
```

تولید کننده

```
while(true)
{ P(ME);
n=n+1;
Buff <===  item
If(n==1) V(delay);
V(ME)  }
```

مصرف کننده

```
P(delay);
while(true)
{P(ME);
 n=n-1;
 item  <===  Buff
V(ME) ;
If (n==0)  P(delay); }
```

الف) کدهای فوق برای تولید کننده و مصرف کننده صحیح می‌باشد.

ب) احتمال دارد ارزش n منفی گردد.

ج) احتمال دارد مصرف کننده در انتظارهمیشگی باقی بماند.

د) احتمال دارد تولید کننده سیگنالی به مصرف کننده که در انتظار نمی‌باشد ارسال نماید.

۷ـ کدام‌یک از موارد ذیل در مورد کدهای نوشته شده برای ایجاد ممانعت دو جانبه صحیح است؟

مقدار اولیه flag=1

```
while (read  and clear (flag) ==0)  wait();
        C. S
flag =1;
wake up   (یکی از پردازش‌های در انتظار)
```

الف) درهمه موارد این کدها جهت ایجاد ممانعت دوجانبه کافی است.

ب) در برخی موارد کدهای فوق ممانعت دوجانبه را ایجاد نمی‌کند.

ج) احتمال دارد که یک پردازش همیشه در انتظار ورود به ناحیه بحرانی باقی بماند.

د) مورد اول و سوم صحیح است.

۸ ـ برای توزیع ۶ امکان یکسان بین تعدادی فرآیند کدام یک از تصمیمات زیر صحیح تر است؟

الف) استفاده از یک سمافور با حداکثر مقدار ۶ ب) استفاده از یک سمافور با حداکثر مقدار ۵

ج) استفاده از دو سمافور با حداکثر مقدار ۲ د) استفاده از شش سمافور باینری(Binary)

۹ ـ مانیتور TEST به شکل زیر تعریف شده است.

```
MONITOR  TEST
   Lock  : Bool;
   Busy  : Condition;
    PROC   A
  { if(Lock)  wait (Busy)
    else Lock= true; }
    PROC    B
  { Lock = false ; signal (Busy);}
    Lock= false ;
END   MONITOR
```

فراخوانی رویه‌های A و B قبل و بعد از ناحیه بحرانی در فرآیندهای همروند:

الف) مشکل بن‌بست ایجاد می‌کند.

ب) مشکل ورود هم‌زمان به ناحیه بحرانی را باعث می‌شود.

ج) مشکل Race Condition به‌وجود نخواهد آمد، زیرا ممانعت ایجاد می‌کند.

د) گاهی اوقات مشکل بن‌بست و گاهی اوقات باعث ورود هم‌زمان به ناحیه بحرانی می‌شود.

۱۰ ـ اگر مقدار اولیه سمافورهای x و y به ترتیب 0 و 1 باشند با توجه به کد فرآیندهای P1 و P2 کدام گزینه صحیح است؟

```
P1  Code                    P2  Code
L1 :............            L2 :...............
     P(x)                        P(y)
   Print (A) ;                 Print (B) ;
     V(y)                        V(x)
    goto  L1                    goto  L2
```

الف) اجرای هم‌زمان P1 و P2 منجر به بن‌بست خواهد شد.

ب) خروجی نهایی *(BA) خواهد بود. (*(a) به معنی n بار (n>=0) تکرار a است.)

ج) امکان ندارد به فرآیند P1 قبل از فرآیند P2 وقت پردازنده تخصیص یابد.

د) عبارت Print (B) فقط یک بار اجرا خواهد شد.

۱۱ ـ آیا کد زیر یک راه حل نرم افزاری برای فرآیندهای PROC(0) و PROC(1) در ناحیه بحرانی می‌باشد؟

```
PROC(i)
    Int i ;
   {while (true)
     { compute ;
       flag [i] =1 ;
       turn= (i+1) mod 2 ;
      while (flag [(i+1) mod 2 ] && turn == i );
              C. S
      flag [i] =0;
      }
}
flag[1]=0  ;  flag[2] =0; turn=1 ;
```

الف) خیر، زیرا شرط ممانعت دو جانبه برقرار نیست. ب) بلی، زیرا شرط ممانعت دو جانبه برقرار است.

ج) خیر، زیرا بن‌بست وجود دارد. د) بلی، زیرا بن‌بست وجود ندارد.

۱۲ ـ فرض کنید مقدار اولیه سمافور x برابر k و سمافور y برابر m و سمافور z برابر L باشد. با توجه به قطعه کد زیرکه توسط n پردازنده اجرا می‌شود وبا فرض این‌که n>k>m>L>0 حداکثر تعداد پردازش‌های منتظر پشت سمافورهای x و y و z را تعیین نمائید.

```
P(x)
P(y)
P(z)
V(z)
V(y)
V(x)
```

الف) k تا پشت x و n-k تا پشت y و m تا پشت z ب) k تا پشت x و k-m تا پشت y و n-k تا پشت z

ج) n-k تا پشت x و k-m تا پشت y و m-L تا پشت z د) n-k تا پشت x و k تا پشت y و m-L تا پشت z

۱۳ ـ آیا کدهای زیر پیاده‌سازی مناسبی برای سمافورهای شمارشی محسوب می‌شوند.

(کد signal , wait مد نظر می‌باشد)

Wait (S)	Signal (S)
P(me)	P(me)
s=s-1	s=s+1
if (s<0)	if (s<=0)
{v(me);p(lock)}	v(lock);
else v(me)	v(me)

الف) کدهای پیاده‌سازی مناسبی می‌باشند.

ب) جمله "else v(me)" در wait (s) به v(me) و جمله v(me) در Signal(s) به "else v(me)" تبدیل می‌شوند.

ج) گزینه الف و ب هر دو صحیح است.

د) هیچ‌کدام

۱۴ ـ کدام‌یک از کدهای ذیل می‌تواند پیاده سازی مناسبی برای عمل wait، Signal سمافورشمارش با استفاده از باینری سمافورها باشد؟

متغیر ME، b باینری سمافور و مقادیر یک و صفر تعریف شده اند.

الف)

Signal (s)	Wait (s)
p(ME)	p(ME)
s=s+1	s=s-1
{if(s<=0) v(b);	if(s<0)
else v(ME)}	{v(ME);p(b)};
	v(ME)

ب)

```
Signal (s)
p(ME)
s=s+1
if (s<=0)   v(b)
v(ME)
```

```
Wait (s)
p(ME)
s=s-1
if (s<=0)
{v(ME);p(b);
else v(ME)}
```

ج)

```
Signal (s)
P(ME)
s=s+1
if (s<=0)   v(b);
v(ME)
```

```
Wait (s)
p(ME)
s=s-1
if(s<0)
{p(b) ;v(ME);
else v(ME)}
```

د) جواب الف و ج هر دو صحیح است.

۱۵ ـ آیا کد زیر یک راه حل نرم افزاری برای مساله ناحیه بحرانی می‌باشد؟

```
P 0
  f0=1
while (turn<>0)
{while (f1);
 turn=0}
C. S
f0=0;
```

```
P1
f1=1
while(turn<>1)
{while (f0);
 turn=1}
C. S
f1=0;
```

الف) مشکل ورود هم‌زمان به ناحیه بحرانی را باعث می‌شود.

ب) مشکل بن‌بست ایجاد می‌نماید.

ج) گاهی اوقات مشکل بن‌بست و گاهی اوقات باعث ورود هم‌زمان به ناحیه بحرانی می‌شود.

د) همیشه ممانعت دو جانبه ایجاد می‌نماید.

۱۶ ـ سه فرآیند p_3, p_2, p_1 در حالت اجرا هستند و طبق جدول زیر عملیات v,p روی سمافور s اجرا می‌شود. فرآیندی که شماره کوچک‌تری دارد برای راه‌اندازی اولویت دارد. حالت این 3 فرآیند پس از اجرای دستورات زیر چیست؟

زمان	فرآیند	دستور
100	P_1	P(S)
200	P_1	P(S)
300	P2	V(S)
400	P_3	P(S)
500	P_1	P(S)
600	P_2	V(S)
700	P_2	P(S)
800	P_1	V(S)
900	P_1	V(S)

الف) P_1, P_2 در حالت اجرا P_3 در حالت مسدود

ب) P_2, P_3 در حالت اجرا و P_1 در حالت مسدود

ج) P_1, P_3 در حالت اجرا و P_2 در حالت مسدود

د) P_1, P_2, P_3 در حالت اجرا

۱۷ ـ دو فرآیند P_1, P_2 به صورت همروند اجرا می‌شوند. امکان اجرای آن‌ها به صورت Interleaved نیز وجود دارد. در صورتی که مقدار اولیه متغیر سراسری a صفر باشد، بعد از اجرای کامل دو فرآیند کدام یک از گزینه‌های ذیل نادرست می‌باشد؟

کد P_1	کد P_2
a =1	b=a
	c=a

الف) مقادیر a و c هر کدام یک می‌باشد و مقدار b صفر است.

ب) مقادیر a , c صفر می‌باشد و مقدار a یک است.

ج) مقادیر a , b هر کدام یک می‌باشد و مقدار c صفر است.

د) هر یک از مقادیر a , b , c یک می‌باشد.

۱۸ ـ کدام یک از گزینه‌های زیر از دلایل استفاده از سمافور نمی‌باشد؟

الف) حل مشکل دوبه‌دو ناسازگاری

ب) حل مشکل بن‌بست

ج)هماهنگی بین فرآیندها

د) حل مشکل انتظار مشغول

۱۹ ـ با توجه به کد برنامه فرآیندهای P_1, P_2 در صورت اجرای همروند و موازی دو فرآیند زیر خروجی کدام یک از مقادیر زیر نمی‌تواند باشد؟

```
P1:                P2:
  Print(A);          Print(C)
  Print(C);          Print(B)
```

الف) CBCA

ب) ACCB

ج) ACBC

د) CABC

۲۰ ـ در یک سیستم تک پردازنده‌ای اشتراک زمانی سه برنامه P_1, P_2, P_3 با قطعه کدهای زیر مفروض است، در صورت اجرای هم‌زمان 3 برنامه کدام خروجی اصلاً رخ نمی‌دهد؟ سمافورهای A , B , C به ترتیب دارای مقدار اولیه 0 , 1 , 2 می‌باشند.

```
P1:                P2:                P3:
  While(1){          while(1){          while(1)
  Wait(A);           Wait(B);           {wait(C);
  Printf(A);         Printf(B);         Printf(C);
  Signal(C);}        Wait(B);           Signal(B);
                     Printf(B);         }
                     Signal(A);
                     }
```

الف) CBBA

ب) CCBB

ج) BCBA

د) BCCA

۲۱ ـ به منظور همگام‌سازی اجرای فرآیندها مطابق زیر از سمافورهای دودوئی با مقدار اولیه صفر استفاده شده است. کدام گزینه زیر نمی‌تواند ترتیب اجرا و اتمام فرآیندها باشد؟

P_1:	P_2:	P_3:	P_4:	P_5:	P_6:
$\vdots$	$P(S_{12})$	$P(S_{13})$	$P(S_{24})$	$P(S_{25})$	$P(S_{36})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$P(S_{35})$	$P(S_{46})$
$V(S_{12})$	$V(S_{24})$	$V(S_{35})$	$V(S_{46})$	$\vdots$	$\vdots$
$V(S_{13})$	$V(S_{25})$	$V(S_{36})$			

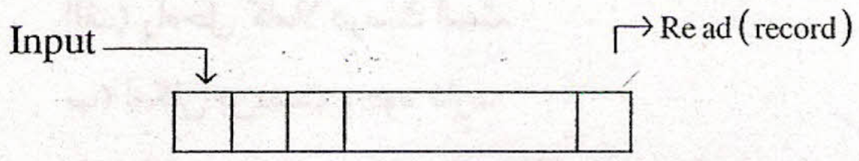
الف) $\rightarrow P_1, P_2, P_4, P_3, P_5, P_6$

ب) $\rightarrow P_1, P_2, P_3, P_4, P_6, P_5$

ج) $\rightarrow P_1, P_3, P_2, P_5, P_4, P_6$

د) $\rightarrow P_1, P_3, P_5, P_2, P_4, P_6$

۲۲ ـ مساله بافر محدود به صورت زیر مطرح شده است؟



Procedure read (record)

```
if not empty Buffer then
 remove (record, Buffer)
 start transfer if necessary;
 else
start transfer if necessary;
Wait;
```

روال وقفه

```
if not full Buffer then
 start transfer
if wait then
 restart reading
```

کدام اظهارنظر صحیح است؟

الف) بودن در روال وقفه به معنی شروع یک transfer است.
جمله restart reading را می‌توان از روال وقفه حذف نمود.

ب) بودن در روال وقفه به معنی شروع یک transfer است.
جمله wait در روال read (record) به این معنی است که بافر خالی است.

ج) بودن در روال وقفه بدین معنی است که transfer به پایان رسیده است.
جمله wait در روال read (record) به معنی این است که بافر خالی است.

د) جمله reastart reading برای شروع مجدد عمل خواندن بافر است که قبلاً ناتمام مانده بوده است. بودن در روال وقفه به معنی خالی بودن بافر است.

۲۳ ـ اگر مقادیر اولیه سمافورهای s , n به ترتیب یک و صفر باشد. چنان‌چه دو زیر روال به‌طور همروند اجرا شوند، کدام گزینه صحیح است؟

```
Proc Producer
Begin
    Repeat
        Procedure;
        Wait (S);
        append;
        Signal (n);
        Signal (S);
    Forever
End
```

```
Proc consumer
Begin
    Repeat
        Wait (S);
        Wait(n);
        Take;
        Signal (s);
        Signal(n);
    Forever
End
```

الف) راه‌حل کاملاً درست است.

ب) امکان بن‌بست وجود دارد.

ج) امکان عدم تامین انحصار متقابل وجود دارد.

د) امکان دارد که Consumer در حالت گرسنگی بماند و Producer فعال باشد.

۲۴ ـ الگوریتم زیر برای ایجاد ناحیه بحرانی بین دو پردازه P_j , P_i داده شده است. کدام گزینه با اجرای همروند این دو فرآیند صحیح می‌باشد؟

کد پردازه P_j

$f_j = true;$

$while(f_i \&\& turn == j);$

ناحیه بحرانی

$turn = j;$

$f_j = false$

الف) شرط انحصار متقابل را ندارد، شرط Progress را دارد.

ب) شرط انحصار متقابل را دارد، شرط Progress را دارد.

ج) شرط انحصار متقابل را دارد، شرط Progress را ندارد.

د) شرط انحصار متقابل را ندارد، شرط Progress را ندارد.

۲۵ ـ اگر دو پروسس P_1, P_2 آمده در ذیل به‌طور همروند اجرا شوند. کدام‌یک از موارد ذیل صحیح است؟

```
P1                                  P2
Legin                               Legin
Loop                                Loop
    non Critical sec.                   non critical sec.
    n1=1;                               n2=1;
    n1=n2+1;                            n2=n1+1;
    Loop exit                           Loop exit
    when(n2=0 or n1<=n2)                whan(n1=0 or n2<n1)
    end Loop;                           end Loop
    CS1                                 CS2
  end Loop;                           end Loop;
end                                 end
```

الف) انحصار متقابل تامین می‌شود.

ب) انحصار متقابل تامین نمی‌شود.

ج) انحصار متقابل تامین می‌شود، اما امکان بن‌بست وجود دارد.

د) انحصار متقابل تامین می‌شود، اما امکان گرسنگی وجود دارد.

۲۶ ـ در راه‌حل زیر برای شامخواران فیلسوف. فرض کنید، اجرای روال‌های برداشتن دو چنگال و گذاشتن هر چنگال از ابتدا تا انتهای روال با رعایت کامل Mutual Exclusion انجام می‌شود. روال take - forks(i) دو چنگال سمت راست و چپ را بررسی می‌کند و اگر هر دو موجود بودند بر می‌دارد وگرنه عمل بررسی را تکرار می‌نماید. روال put-fork(i) چنگال شماره i را می‌گذارد و از روال خارج می‌شود؟

الف) دارای بن‌بست

ب) فاقد بن‌بست و فاقد گرسنگی

ج) دارای بن‌بست و گرسنگی

د) فاقد بن‌بست اما دارای گرسنگی

۲۷ ـ به مانیتور زیر توجه کنید. این مانیتور می‌تواند توسط فرآیندها فراخوانده شود، این مانیتور چه عملی انجام می‌دهد؟

```
Monitor test
  Int n
Condition c
 Proc Arrie
  { n=n+1;
  if ( n≤1 ) wait(c)
  else signal(c)
  }
 n=0
END MONITOR
```

الف) فرآیندی که مانیتور را فرا می‌خواند در انتظار متغیر شرطی C در انتظار می‌ماند.

ب) همواره 10 فرآیند در این مانیتور منتظر هستند فرآیندهای بعدی در مانیتور منتظر نمی‌مانند.

ج) 10 فرآیند اول با فراخوانی مانیتور در انتظار باقی می‌مانند، سپس با فراخوانی فرآیندهای بعدی همه آن‌ها می‌توانند از مانیتور خارج شده به کار خود ادامه دهند.

د) هر فرآیند متغیر n را از صفر تا 10 افزایش می‌دهد و سپس از مانیتور خارج می‌گردد.

جواب‌های تشریحی تست‌های مربوط به مبحث نواحی بحرانی

۱ـ دستورالعمل swap روی دو متغیر منطقی به طور Atomic عمل می‌کند و مقدارمتغیرهای مربوطه را جابه‌جا می‌نماید(مقدار متغیر اول در دوم و دوم در اول قرار می‌گیرد.) با توجه به کد مفروض، هر فرآیندی زودتر دستور swap را اجرا نماید متغیر محلی key خود را false و lock را true می‌نماید و از حلقه while خارج شده به ناحیه بحرانی خود وارد می‌شود وتا زمانی‌که در ناحیه بحرانی باشد فرآیند دوم در حلقه while به علت true بودن key محلی خود و lock باقی می‌ماند.

به محض خروج فرآیند اول از ناحیه بحرانی فرآیند دوم ازحلقه while خارج شده و در ناحیه بحرانی خود وارد می‌شود. بنابراین کد فوق راه حل مناسبی برای ایجاد ممانعت دو جانبه برای دو فرآیند می‌باشد.

گزینه سوم صحیح است.

۲ـ حالتی را در نظر بگیرید که تولید کننده یک Item را در بافر قرارداده و دستورالعمل‌های V(delay),V(ME) را اجرا کرده باشد و مصرف کننده فرآیند را شروع نماید. دستور P(delay) مقدار delay را صفر و دستور P(ME) ورود به ناحیه بحرانی را انحصاری می‌نماید و اجازه نمی‌دهد که تولید کننده به بافر دسترسی داشته باشد و بعد از کسر یک واحدازn مقدار n صفر می‌شود و Item تولید شده از بافر برداشته شده و سپس دستورif باعث می‌شود که P(delay) اجرا گردد. این دستور فرآیند مصرف کننده را به wait می‌برد و چون این فرآیند اجازه دسترسی تولید کننده به بافر را نمی‌دهد، بنابراین تولید کننده بعد از اجرای دستور P(ME) به wait می‌رود و قادر به ادامه فرآیند نخواهد بود.

بنابراین هر دو فرآیند تولیدکننده و مصرف کننده در انتظار یکدیگر باقی می‌مانند.

گزینه دوم صحیح است.

۳ـ ابتدا با توجه به کد برنامه دو فرآیند P0 و P1 را به شکل زیر می نویسیم :

```
P0                          P1
f(0)=T                      f(1)=T
While(Turn==0)              While(Turn==1)
{ Turn= 0;                  { Turn=1;
While(f(1)); }              While(f(0);}
Critical Section            Critical Section
f(0)=F;                     f(1)=F;
```

فرض کنید Turn=1 باشد و فرآیند P0 اجرا شود.

f(0)، True می‌شود.

دستور While(Turn==1) باعث می‌شود کنترل وارد حلقه While شده و سپس Turn=0 گردد.

در این لحظه فرض کنید اینتراپت رخ دهد وکنترل به فرآیند P1 وارد شود. با اجرای فرآیند P1 دستورات ذیل اجرا می‌شود.

f(1)، True می‌شود.

کنترل وارد حلقه While می‌شود زیرا Turn=0 است.

Turn=1 می‌گردد و سپس حلقه While دوم اجرا می‌شود و در Loop باقی می‌ماند زیرا f(0)، True است. بنابراین فرآیند P1 پیشرفتی نخواهد داشت.

چنان‌چه اینتراپت اتفاق افتد و کنترل به فرآیند P0 برگردد ادامه کار در فرآیند P0 از سر گرفته می‌شود. بدین معنی که دستور While(f(1)) اجرا می‌شود و چون f(1) نیز True می‌باشد حلقه تکرار می‌شود. بنابراین فرآیند P0 نیز در این حلقه باقی می‌ماند و

پیشرفتی نخواهد داشت. نتیجتاً هر دو فرآیند در حلقه While باقی می‌مانند و هیچ گاه از حلقه خارج نخواهند شد. بدین معنی که امکان وجود بن‌بست رخ می‌دهد. از طرفی چون مقدار turn یا صفر است و یا یک بنابراین احتمال ورود هم‌زمان به ناحیه بحرانی وجود ندارد.

گزینه دوم صحیح است.

۴ـ دستور العمل p یا down بر روی سمافور شمارشی s به صورت زیر تعریف می‌شود:

```
p(s) :{  while (s<=0);
         s=s-1}
```

برای پیاده سازی دستورالعمل p باید حالت Atomic آن حفظ شود بدین معنی که دو دستور فوق همانند یک ناحیه بحرانی در نظر گرفته شوند.

به کمک دستور swap و دو متغیر a و b می‌توان این عمل را انجام داد. اگر به کد پیشنهادی مساله توجه شود در می یابیم که متغیر a سراسری و b محلی است و دستور swap(a,b) باعث می‌شود ناحیه بحرانی مذکور فقط توسط یک فرآیند قابل دسترسی باشد، زیرا هنگامی که اولین فرآیند swap را در دستور while اجرا می‌نماید، اجازه ورود به ناحیه بحرانی برای آن صادر می‌شود و بقیه فرآیندها در حلقه while در انتظار باقی می‌مانند. پس از خروج فرآیند اول از ناحیه بحرانی با اجرای مجدد swap فرآیند بعدی وارد ناحیه بحرانی می‌گردد، اما تنها مشکلی که ملاحظه می‌شود اجرای حلقه while (s<=0) می‌باشد، زیرا هنگامی‌که مقدار s=0 شود، کد اجرایی در loop باقی می‌ماند و مدام چک می‌شود تا s>0 شود از طرفی فرآیندهای بعدی نمی‌توانند وارد ناحیه بحرانی شوند، چون مقدار a=1 است، بنابراین مشکل بن‌بست ایجاد می‌شود.

گزینه سوم صحیح است.

۵ـ سناریو ذیل را برای دو فرآیند در نظر بگیرید:

چنان‌چه فرآیند اول کنترل پردازنده را در دست بگیرد:

۱- Lock=False

۲- شرط While که TSL(Lock) است چک می‌گردد و چون اجرای این دستور باعث می‌شود TSL، False و Lock برابر True شود، کنترل وارد ناحیه بحرانی می‌گردد و چنان‌چه در این لحظه کنترل به فرآیند دوم منتقل شود آن‌گاه:

۱- شرط While که TSL(Lock) است، چک می‌گردد و چون مقدار آن True می‌باشد، پس فرآیند باید دستور بعدی را که Wait است اجرا نماید، اما اگر قبل از ورود به Wait کنترل به فرآیند اول داده شود آن‌گاه:

۱- ناحیه بحرانی اجرا می‌شود.

۲- Lock=False می‌شود.

۳- Signal داده می‌شود، اما هیچ فرآیندی هنوز در Wait نمی‌باشد و Signal به هدر می‌رود.

بنابراین چنان‌چه کنترل به فرآیند ۲ برگردد فرآیند به Wait می‌رود و در انتظار ورود به ناحیه بحرانی قرار می‌گیرد. در صورتی‌که فرآیندی داخل ناحیه بحرانی نیست و این فرآیند می‌بایست وارد ناحیه بحرانی شود.

گزینه دوم صحیح است.

۶ـ فرض کنید تولید کننده کنترل پردازنده را در دست گیرد و یک Item داخل بافر قراردهد (n=1 و Delay=1 می‌شود) و با دستور V(me) اجازه دهد تا مصرف کننده به بافر دسترسی نماید. حال اگر کنترل به مصرف کننده منتقل شود با اجرای دستور P(delay) مقدار delay برابر صفر شده کنترل وارد حلقه While می‌شودو از n یک واحد کسر می‌گردد و Item تولید شده برداشته می‌شود(N=0) و با اجرای V(me) اجازه ورود به ناحیه بحرانی صادر می‌گردد. تصور نمایید که قبل از اجرای دستور If کنترل به

ـ چنان‌چه فرآیند 1,2 هر کدام پردازنده را در دست گیرند، دستور p(lock) را اجرا خواهند کرد و فقط یکی از آن‌ها به‌خاطر ارزش lock=1 کارش را ادامه می‌دهد و دیگری در انتظار lock باقی می‌ماند. در صورتی که اگر کد موردنظر درست عمل می‌کرد می‌بایست هر دوی آن‌ها کارشان را بدون انتظار ادامه می‌دادند بنابراین این سناریو صحت کد را نقض می‌نماید.

چنان‌چه کد به‌شکل ذیل تغییر نماید سناریوی فوق درست عمل می‌کند و هیچ سناریوی دیگری کد ذیل را نمی‌تواند رد کند.

Wait (S)	Signal(S)
p(me);	p(me)
s = s − 1	s = s + 1
if (s <= 0){v(me);p(lock)}	if (s < 0) v(lock)
v(me)	else v(me)

گزینه ۱ صحیح است.

۱۴ ـ برای پیاده سازی دو دستور Wait و Signal برروی سمافورشمارشی دو کار عمده باید صورت پذیرد.

کار اول : کد Wait و کدSignal طوری پیاده سازی شوند که حالت Atomic بودن خود را حفظ نماید.

کار دوم : کدهای زیر برای Wait و Signal تعریف شوند:

کد Signal(s)	کد Wait(s)
s=s+1	s=s-1
if (s<=0)Signal	if (s<0) Wait

برای پیاده سازی کار اول کافی است باینری سمافوری مثل ME با مقدار اولیه ۱ تعریف شود و قبل از ورود به کدهای فوق دستور p بر روی ME و بعد از خروج از کدها دستور v بر روی ME اعمال شوند.

دستور Wait و Signal درداخل کدها به کمک باینری سمافور b که به صفر مقدارگذاری شده باشد، پیاده سازی می‌شوند. بدین معنی که به‌جای Wait دستور p بر روی سمافور b و به‌جای Signal دستورv بر روی سمافور b جایگزین می‌گردد. توجه کنید که قبل از قراردادن دستور p بر روی سمافور b باید دستور v بر روی سمافور ME اعمال شود، زیرا در ابتدای ورود به کد دستور p برروی ME اعمال شده است. پس قبل از بلوکه شدن کد مربوطه با دستور p(b) عمل v بر روی سمافور ME باید صورت گیرد تا حالت بن‌بست ایجاد نگردد.

گزینه اول صحیح است.

۱۵ ـ فرض کنید $f_0=f_1=0$ و Turn=1 باشد و سناریوی ذیل برقرار گردد:

ابتدا کنترل به فرآیند P0 منتقل شود آن‌گاه:

۱- $f_0=1$ می‌شود.

۲-کنترل داخل حلقه While می‌شود (چون Turn≠0 است.)

۳- در حلقه While دوم وارد نمی‌شود (چون $f_1=0$ است.) به ابتدای دستور Turn=0 می‌رسد. چنان‌چه در همین لحظه کنترل به فرآیند p1 منتقل شود آن‌گاه:

۱- $f_1=1$ می‌شود.

۲- چون Turn=1 است وارد بدنه حلقه While نمی‌شود و وارد ناحیه بحرانی می‌گردد.

در همین لحظه کنترل پردازنده به فرآیند p0 واگذار گردد Turn=0 می‌شود و حلقه While(Turn≠0) مجدداً Test می‌گردد. این بار چون Turn=0 است کنترل وارد ناحیه بحرانی خواهد شد، بنابراین هر دو فرآیند p_0 و p_1 هم‌زمان وارد ناحیه بحرانی می‌شوند.

گزینه اول صحیح است.

۱۶ ـ گزینه چهارم صحیح می‌باشد.

مقدار اولیه سمافور یک می‌باشد.

فرآیند	P_1	P_1	P_2	P_3	P_1	P_2	P_2	P_1	P_1
دستور	P	P	V	P	P	V	P	V	V
مقدار سمافور پس از اجرای دستور	0	0	0	0	0	0	0	0	0
فرآیندهای منتظر	-	P_1	-	P_3	P_1, P_3	P_3	P_2, P_3	P_3	-

۱۷ ـ گزینه سوم صحیح می‌باشد.

چنان‌چه a , b هر دو یک باشند مقدار C نمی‌تواند صفر گردد، زیرا دستور c = a باعث می‌شود مقدار C یک شود.

۱۸ ـ گزینه دوم صحیح می‌باشد.

کاربرد سمافور برای هماهنگی بین فرآیندها و حل مشکل دوبه‌دو ناسازگاری و انتظار مشغول می‌باشد.

۱۹ ـ گزینه اول صحیح می‌باشد.

اجرای همروند دو فرآیند P_2, P_1 باعث نمی‌شود تا CBCA چاپ گردد زیرا بعد از چاپ CB توسط فرآیند P_2، فرآیند AC, P_1 را می‌تواند چاپ کند، حتماً باید اول A چاپ شود و سپس C

۲۰ ـ ترتیب اجرای P_1, P_2, P_3 خروجی گزینه اول را نتیجه می‌دهد.

ترتیب اجرای P_1, P_2, P_3, P_2 به ترتیب از راست به چپ خروجی گزینه سوم را نتیجه می‌دهد. خروجی گزینه دوم توسط اجرای فرآیندهای P_2, P_3, P_3 صورت می‌گیرد.

خروجی گزینه چهارم امکان ندارد، زیرا اگر بخواهیم خروجی مربوطه را داشته باشیم. پس از اجرای P_3, P_3, P_2 باید فرآیند P_1 اجرا شود و این امکان‌پذیر نمی‌باشد، زیرا مقدار سمافور A صفر است و فرآیند به انتظار می‌رود.

گزینه چهارم صحیح است.

۲۱ ـ گزینه چهارم صحیح می‌باشد.

اجرای فرآیند P_5 قبل از فرآیند P_2 نمی‌تواند پیشروی داشته باشد. زیرا در ابتدای فرآیند P_5، دستور $P(S_{25})$ باعث می‌شود فرآیند مربوطه به انتظار برود در صورتی که اگر اول فرآیند P_2 اجرا می‌شد دستور $V(S_{25})$ باعث می‌شد که فرآیند P_{25} به انتظار نرود.

۲۲ ـ گزینه دوم صحیح می‌باشد.

بودن در روال وقفه به معنی شروع یک transfer می‌باشد.

همان‌طور که مشاهده می‌کنید در روال وقفه چنان‌چه Buffer پر نباشد transfer شروع می‌شود و اگر قبلاً عمل خواندن از بافر ناتمام مانده باشد و در انتظار باشیم روتین خواندن دوباره شروع می‌شود.

جمله wait در روال (record) read به این معنی است که بافر خالی می‌باشد که با فراخوانی روال وقفه انتقال اطلاعات به بافر شروع می‌شود و سپس عمل خواندن از بافر از سر گرفته می‌شود.

۲۳ ـ گزینه دوم صحیح است.

حالتی را در نظر بگیرید که زیر روال Consumer دستور wait(n), wait (s) را اجرا نماید، مقدار سمافور S صفر می‌شود و زیر روال در wait(n) به حالت مسدود می‌رود. حال اگر زیر روال Producer دستور wait (s) را اجرا نماید به انتظار می‌رود. بنابراین هر دو زیر روال در حالت مسدود قرار می‌گیرد.

۲۴ ـ گزینه اول صحیح می‌باشد.

از فرآیند P_i شروع می‌کنیم فرض می‌کنیم که turn=i باشد، به دلیل این‌که $f_j=0$ است فرآیند P_i وارد ناحیه بحرانی می‌شود. حال اگر فرآیند P_j اجرا شود، چون turn = i می‌باشد پس این فرآیند هم وارد ناحیه بحرانی می‌گردد. هیچ‌گاه بن‌بست اتفاق نمی‌افتد، چون مقدار turn هم‌زمان نمی‌تواند i,j گردد.

۲۵ ـ فرآیند P_1, P_2 قبل از ورود به ناحیه بحرانی شماره‌ایی را دریافت می‌نماید که احتمال دارد برای هر دو مساوی باشد، زیرا دریافت شماره n_1, n_2 قبل از ورود به ناحیه بحرانی ممکن است منجر به حالتی گردد که ارزش n_1 و n_2 مساوی شود. چنان‌چه n_1 کوچک‌تر یا مساوی n2 شود، فرآیند P_1 وارد ناحیه بحرانی می‌گردد (توجه کنید که در جمله Loop exit در بدنه فرآیند P_1 چنان‌چه فرآیند P_2 تمایلی به ورود به ناحیه بحرانی نداشته باشد، فرآیند P_1 وارد ناحیه بحرانی می‌شود). برعکس چنان‌چه n_2 کوچک‌تر از n_1 باشد نوبت فرآیند P_2 است که وارد ناحیه بحرانی خود گردد، بنابراین به هیچ وجه مشکل ورد هم‌زمان و یا عدم پیشروی در فرآیندها دیده نمی‌شود. ضمناً قحطی‌زدگی هم به‌وجود نمی‌آید زیرا هر فرآیندی که وارد ناحیه بحرانی می‌شود بعد از خروج از ناحیه بحرانی و قبل از ورود مجدد به ناحیه بحرانی شماره جدیدی دریافت می‌کند که این شماره بزرگ‌تر از شماره فرآیند طرف مقابل می‌باشد و همین باعث می‌شود که خودش مجدداً وارد ناحیه بحرانی نگردد و اجازه دهد فرآیند طرف مقابل وارد ناحیه بحرانی خود گردد.

گزینه اول صحیح است.

۲۶ ـ هر فیلسوف قبل از اقدام به خوردن، سعی در بر داشتن دو چنگال می‌نماید. چنان‌چه یکی از دو چنگال و یا هر دو موجود نباشد از خوردن منصرف می‌شود و همین عمل دوباره تکرار می‌شود. چون معلوم نیست که چه زمانی دو چنگال موجود است ممکن است فیلسوف برای مدت نامعلومی موفق به برداشتن دو چنگال نشود بنابراین احتمال گرسنگی ایجاد می‌شود.

گزینه چهارم صحیح است.

۲۷ ـ هر فرآیند با فراخوانی مانیتور باعث می‌شود که به مقدار n یک واحد اضافه نماید و تا زمانی که n به عدد یازده نرسیده است فرآیند فراخواننده در انتظار متغیر شرطی باقی می‌ماند در غیر این‌صورت باعث می‌شود که با اجرای دستور Signal فرآیندی را که در ابتدای صف انتظار می‌باشد، بیدار نماید و از مانیتور خارج گردد.

گزینه ۳ صحیح می‌باشد.