موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

سیستم عامل

(مدیریت فرآیندها و نخ‌های هم‌روند)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

آبراهام سیلبرشاتز، ویلیام استالینگز و اندرو اس تنن‌بام

ارسطو خلیلی‌فر

کلیه‌ی حقوق مادی و معنوی این اثر در سازمان اسناد و کتابخانه‌ی ملی ایران به ثبت رسیده است.

۷ فرآیندهای همروند

مقدمه

در اغلب سیستم‌های امروزی، تعدادی از فرآیندها یا نخ‌ها به صورت همروند بر روی یک پردازنده و یا به صورت موازی بر روی چندین پردازنده اجرا می‌شوند. در سیستم‌های چند برنامگی و چند پردازنده‌ای، همروندی فرآیندها و نخ‌ها، یک پدیده‌ی عادی به شمار می‌آید.

فرآیندهای هم روند و همکار، به ارتباط با یکدیگر نیاز دارند. آن‌ها برای دستیابی به یک هدف مشترک، نیازمند همکاری، هماهنگی، تبادل داده و استفاده از داده‌ها و سایر منابع مشترک هستند. بنابراین مدیریت اجرای همروند چند فرآیند بر روی یک پردازنده و اجرای موازی چند فرآیند بر روی چندین پردازنده، حائز اهمیت فراوان می‌باشد. این مدیریت باید به گونه‌ای باشد که اجرای یک فرآیند آسیبی به اجرای فرآیند(های) همکار دیگر نرساند.

در هنگام طراحی سیستم عامل، در زمینه‌ی ارتباط بین فرآیندها با سه مسأله‌ی اساسی زیر مواجه هستیم:

۱ ـ تبادل داده

گاهی یک فرآیند، به نتیجه‌ی محاسبات یک فرآیند دیگر نیاز دارد، بنابراین به یک مکانیسم برای ارتباط بین فرآیندها نیاز است. انواع مکانیزم‌های تبادل داده بین فرآیندها به روش‌های زیر است:

- حافظه مشترک

- فایل مشترک
- تبادل پیام (در انتهای فصل تشریح خواهد شد)
- لوله (نوعی شبه فایل در سیستم عامل یونیکس)

تبادل داده برای نخ‌ها ساده می‌باشد، زیرا نخ‌ها یک فضای آدرس مشترک دارند. نخ‌های متعلق به فرآیندهای جداگانه که در فضای آدرس متفاوت قرار دارند، در صورت نیاز به ارتباط باید از مکانیسم‌های ارتباط فرآیندها، استفاده کنند که نیازمند فراخوان‌های سیستمی وقت‌گیر هستند.

ناحیه‌ی بحرانی

اگر چند فرآیند قصد دسترسی به یک منبع مشترک را داشته باشند، قطعه کدی از هر فرآیند را که در آن به دستکاری این منبع مشترک می‌پردازد، ناحیه‌ی بحرانی می‌گویند.

نکته : در همه‌ی نواحی بحرانی، دسترسی به منبع مشترک، وجود دارد، اما عکس آن همیشه صادق نیست و هرگونه دسترسی به منبع مشترک باعث رقابت و ایجاد ناحیه‌ی بحرانی نمی‌شود.

منبع بحرانی

منبعی که توسط ناحیه‌ی بحرانی مورد دستیابی قرار می‌گیرد، منبع بحرانی نام دارد، مانند متغیرهای مشترک رقابت‌زا.

۲ ـ شرایط رقابتی (مسابقه)

هرگاه دو یا چند فرآیند همزمان با هم وارد ناحیه‌ی بحرانی (منبع مشترک) شوند، شرایط رقابتی پیش می‌آید. در شرایط رقابتی، نتیجه‌ی نهایی بستگی به ترتیب دسترسی‌ها دارد. در واقع فرآیندهای همکار بر هم اثر دارند و اینکه پردازنده، به چه ترتیبی و در چه زمان‌هایی بین آنها تعویض متن انجام دهد در ایجاد پاسخ نهایی اثرگذار خواهد بود. بنابراین علت شرایط رقابت **تعویض متن پردازنده** بین فرآیندهای همکار است.

مانند زندگی عادی انسان‌ها، در فرآیندها نیز همیشه برخورد، تداخل و رقابت بر سر عوامل مشترک است. هرگز نمی‌توانیم رقابتی را فرض کنیم که هیچ عامل مشترکی در آن نباشد. تمام نزاع‌ها بر سر تصاحب یک «عامل مشترک» است که دو یا چند نفر یا قوم (یا فرآیند یا نخ) به طور همروند یا موازی تلاش می‌کنند تا آن را بدست آورند، یعنی بر روی عامل کاری انجام دهند.

مثال: شرایط رقابتی

دو فرآیند P_1 و P_2 را با کد زیر در نظر بگیرید که در یک سیستم اشتراک زمانی به صورت هم روند اجرا می‌شوند. فرض کنید متغیر a از نوع سراسری و مشترک است و مقدار اولیه آن نیز صفر است، بعد از اجرای کامل دو فرآیند، مقادیر a ، b و c چه خواهد شد؟

کد : $P_۲$	کد : $P_۱$
b=a	a=۱
c=a	

حالت اول:

کد : $P_۲$	کد : $P_۱$
② b=a	① a=۱
③ c=a	

خروجی : a=۱ , b=۱ , c=۱

حالت دوم:

کد : $P_۲$	کد : $P_۱$
① b=a	② a=۱
③ c=a	

خروجی : a=۱ , b=۰ , c=۱

حالت سوم:

کد : $P_۲$	کد : $P_۱$
① b=a	③ a=۱
②c=a	

خروجی : a=۱ , b=۰ , c=۰

توجه :اما مشکل اینجاست که مقدار نهایی متغیرهای a ، b و c، به نحوه‌ی **تعویض متن پردازنده** یا به عبارتی، به ترتیب اجرای دستورالعمل‌ها، بستگی دارد و می‌توانند مقادیر مختلفی را داشته باشند. این پدیده، حاصل رقابت بر سر تصاحب یک عامل مشترک (متغیر مشترک a) است.

مثال: شرایط رقابتی

دو فرآیند هم روند $P_۱$ و $P_۲$ در یک سیستم اشتراک زمانی که از متغیر مشترک سراسری S ، در بخشی از کد خود استفاده می‌کنند در نظر بگیرید، بعد از اجرای کامل دو فرآیند، مقدار نهایی S چه خواهد شد؟ (مقدار اولیه متغیر سراسری S برابر صفر است)

$P_۱$:	$P_۲$:
S=S+۱	S=S-۱

از آنجا که این فرآیندها به زبان اسمبلی یک ماشین فرضی در نظر گرفته می‌شوند، لذا در ادامه دستورات فوق را به صورت سطح غیرانتزاعی‌تر (نمایش جزئیات) و در سطح اسمبلی بازنویسی

می‌کنیم:

P_1:	P_2:
MOVE RGISTER , S	MOVE REGISTER , S
INC REGISTER	DEC REGISTER
MOVE S , REGISTER	MOVE S, REGISTER

حالت اول:

فرض کنید P_1 کامل اجرا شود و سپس P_2 کامل اجرا شود (یا برعکس).

P_1:	P_2:
① REGISTER ← ۰	④ REGISTER ← ۱
② REGISTER ← ۱	⑤ REGISTER ← ۰
③ S ← ۱	⑥ S ← ۰

بنابراین مقدار نهایی متغیر S برابر صفر خواهد بود. (S=۰)

حالت دوم:

فرض کنید ابتدا ترتیب زیر اجرا شود.

P_1:	P_2:
① REGISTER ← ۰	② REGISTER ← ۰
INC REGISTER	③ REGISTER ← - ۱
MOVE S, REGISTER	④ S ← - ۱

سپس پردازنده در اثر تعویض متن به فرآیند P_1 باز گردد.

توجه: هر فرآیند محتویات رجیسترهای خودش را قبل از تعویض متن در PCB ذخیره می‌کند. بنابراین در این لحظه مقدار رجیستر در PCB فرآیند P_1، برابر صفر است.

حال در ادامه دستورات باقی‌مانده فرآیند P_1 اجرا می‌شوند.

P_1:
⋮
⑤ REGISTER ← ۱
⑥ S ← ۱

بنابراین مقدار نهایی متغیر S برابر مثبت یک خواهد بود. (S=+۱)

حالت سوم:

فرض کنید ابتدا ترتیب زیر اجرا شود.

$P_۱$:	$P_۲$:
② REGISTER ← ۰	① REGISTER ← ۰
③ REGISTER ← ۱	DECREGISTER
④ S ← ۱	MOVES, REGISTER

سپس پردازنده در اثر تعویض متن به فرآیند $P_۲$ باز گردد.

توجه : هر فرآیند محتویات رجیسترهای خودش را قبل از تعویض متن در PCB ذخیره می‌کند. بنابراین در این لحظه مقدار رجیستر در PCB فرآیند $P_۲$ ، برابر صفر است.

حال در ادامه دستورات باقی‌مانده فرآیند $P_۲$ اجرا می شوند.

$P_۲$:
⋮
⑤ REGISTER ← - ۱
⑥ S ← - ۱

بنابراین مقدار نهایی متغیر S برابر منفی یک خواهد بود. (S=-۱)

توجه : اما مشکل اینجاست که مقدار نهایی متغیر S ، به نحوه‌ی **تعویض متن پردازنده** یا به عبارتی، به ترتیب اجرای دستورالعمل‌ها، بستگی دارد و می‌تواند مقادیر ۰، ۱ و ۱ـ را داشته باشد. این پدیده، حاصل رقابت بر سر تصاحب یک عامل مشترک (متغیر مشترک S) است.

توجه :وجود پدیده‌ی رقابت در دو مثال قبل در سیستم‌های تک پردازنده‌ای ناشی از وقفه‌ای است که می‌تواند اجرای دستورالعمل‌ها را در هر کجای فرآیند متوقف نماید. (پدیده‌ی تعویض متن). این وضعیت در سیستم‌های چند پردازنده‌ای نیز ممکن است پیش بیاید، به علاوه این که دو یا چند فرآیند می‌توانند به موازات هم اجرا شده و برای دسترسی به یک عامل مشترک در رقابت باشند.

برای کنترل شرایط رقابتی، باید راه حلی ارائه شود که سه شرط زیر را رعایت کند:

۱ ـ شرط انحصار متقابل

برای برقراری شرط انحصار متقابل، عامل مشترک را اسکورت کنید، مانند زمانی که وارد باجه‌ی تلفن همگانی (عامل مشترک) می‌شوید، در را می‌بندید تا مانع ورود شخص دیگری گردید! در عالم انسان‌ها، هیچ دو فردی نباید به طور همزمان وارد عامل مشترک شوند. در عالم فرآیندها نیز هیچ دو فرآیندی نباید به طور همزمان وارد عامل مشترک (ناحیه‌ی بحرانی) شوند. استفاده‌ی همزمان از عامل مشترک معنا ندارد! (اخلاقی نیست) بنابراین باید راهی را پیدا کنیم که از ورود همزمان دو یا چند فرآیند به ناحیه‌ی بحرانی جلوگیری کند. به عبارت دیگر، آنچه که ما به آن نیاز داریم، انحصار متقابل است که در متون فارسی به آن دو به دو ناسازگاری یا مانعةالجمعی نیز گفته می شود، یعنی اگر یکی از فرآیندها

در حال استفاده از حافظه‌ی اشتراکی، فایل اشتراکی و یا هر عامل اشتراکی رقابت‌زاست باید مطمئن باشیم که دیگر فرآیندها، در آن زمان از انجام همان کار محروم می‌باشند. در واقع از بین تمام فرآیندها، در هر لحظه تنها یک فرآیند مجاز است، در عامل مشترک باشد. بدین معنی که اگر فرآیندی در ناحیه‌ی بحرانی است، از ورود فرآیندهای دیگر به همان ناحیه‌ی بحرانی جلوگیری شود و تا خارج شدن فرآیند اول منتظر بمانند، زیرا هیچ دو فرآیندی نباید به طور همزمان وارد ناحیه‌ی بحرانی شوند. به یاد داشته باشید که استفاده‌ی همزمان از عامل مشترک معنا ندارد!

بنابراین برای برقراری شرط انحصار متقابل باید ساختاری را طراحی کنیم که در هر لحظه فقط یک فرآیند مجوز ورود به ناحیه‌ی بحرانی را داشته باشد. لذا هر فرآیند برای ورود به بخش بحرانی‌اش باید اجازه بگیرد. بخشی از کد فرآیند که این اجازه گرفتن را پیاده‌سازی می‌کند، بخش ورودی نام دارد. بخش بحرانی می‌تواند با بخش خروجی دنبال شود. این بخش خروجی کاری می‌کند که فرآیندهای دیگر بتوانند وارد ناحیه‌ی بحرانی‌شان بشوند. بقیه‌ی کد فرآیند را بخش باقی‌مانده می‌نامند. بنابراین ساختار کلی فرآیندها برای برقراری شرط انحصار متقابل به صورت زیر می‌باشد:

```
P (int i) {
   while (TRUE) {
      entry_section (); // تلاش برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی
      critical_section (); // ناحیه‌ی بحرانی
      exit_section (); // اعلام خروج از ناحیه‌ی بحرانی
      remainder_sectioln (); // ناحیه‌ی باقی‌مانده
      }
   }
```

۲ـ شرط پیشرفت

فرآیندی که داوطلب ورود به ناحیه‌ی بحرانی نیست و نیز در ناحیه‌ی بحرانی قرار ندارد، نباید در رقابت برای ورود سایر فرآیندها به ناحیه‌ی بحرانی شرکت کند، به عبارت دیگر، نباید مانع ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شود. در یک بیان ساده‌تر می‌توان گفت، فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، حق جلوگیری از ورود فرآیندهای دیگر به ناحیه‌ی بحرانی را ندارد. یعنی نباید در تصمیم‌گیری برای ورود فرآیندها به ناحیه‌ی بحرانی شرکت کند.

۳ـ شرط انتظار محدود

فرآیندهایی که نیاز به ورود به ناحیه‌ی بحرانی دارند، باید مدت انتظارشان محدود باشد، یعنی نباید به طور نامحدود در حالت انتظار باقی بمانند.

انتظار نامحدود بر دو دسته می‌باشد: (۱) قحطی، (۲) بن‌بست، بنابراین نباید در شرایط رقابتی بین فرآیندها، قحطی یا بن‌بست رخ دهد.

قحطی (گرسنگی)

در عالم زندگی قحطی زمانی رخ می‌دهد که عده‌ای مدام از منابع مشترک استفاده کنند، و عده‌ای دیگر قادر به استفاده از منابع مشترک نباشند. زیرا دسته‌ی اول از اختصاص منابع به دسته‌ی دوم به طور مداوم و بدون رعایت یک حد بالای مشخص جلوگیری می‌کنند. در عالم فرآیندها نیز هرگاه فرآیندی به مدت نامعلوم و بدون رعایت یک حد بالای مشخص در انتظار گرفتن یک منبع بحرانی یا دسترسی به یک عامل مشترک بماند و فرآیندی دیگر مدام در حال استفاده از منبع بحرانی باشد، در این حالت فرآیند اول دچار قحطی شده است. بنابراین در صورت اقدام یک فرآیند برای ورود به ناحیه‌ی بحرانی، باید محدودیتی برای تعداد دفعاتی که سایر فرآیندها می‌توانند وارد ناحیه‌ی بحرانی شوند، وجود داشته باشد تا قحطی رخ ندهد.

بن‌بست

به وضعیتی که در آن مجموعه‌ای متشکل از دو یا چند فرآیند برای همیشه منتظر یکدیگر بمانند (مسدود) و به عبارت دیگر دچار سیکل انتظار ابدی شوند، بن‌بست گفته می‌شود. به یاد آوردید که شرط انحصار متقابل شرط لازم، شرط انحصاری بودن و شرط نگهداری و انتظار شروط بدیهی و شرط سیکل انتظار (انتظار چرخشی) شرط کافی برای وقوع بن‌بست بود. برای مثال همه فرآیندها ممکن است، منتظر باشند تا فرآیند دیگری از همین مجموعه، یک منبع بحرانی را آزاد کند.

توجه : به تفاوت قحطی و بن‌بست دقت کنید، در قحطی فرآیندی مدام در حال کار و فرآیندی دیگر به مدت نامعلوم در انتظار است. اما در بن‌بست، مجموعه‌ای از فرآیندها در سیکل انتظار ابدی، گرفتار شده‌اند. نه راه پس دارند و نه راه پیش.

توجه : در کنترل شرایط رقابتی، رعایت شرط انحصار متقابل، شرط لازم و رعایت شروط پیشروی و انتظار محدود، شروط کافی برای ارائه‌ی یک راه حل جامع به شمار می‌آیند.

۴ـ همگام‌سازی

گاهی اوقات خواسته ما این است که فرآیندها به یک ترتیب مشخص و از قبل تعیین شده بر روی یک عامل مشترک (داده مشترک) عملیاتی را انجام دهند. واضح است که در این حالت تبادل داده به روش استفاده از حافظه‌ی مشترک است. همچنین از آنجا که پای یک عامل مشترک درمیان است، پس رقابت بر سر تصاحب این عامل مشترک هم در میان است. بنابراین راه حل همگام‌سازی باید به گونه‌ای باشد که فرآیندهای همکار دچار شرایط رقابتی نشوند. به بیان دیگر باید مانع اثر مخرب **تعویض متن**

پردازنده بین فرآیندهای همکار که عامل ایجاد شرایط رقابتی است، شد.
مثلاً اگر اول فرآیند P_1 داده‌ای را باید تولید کند و سپس فرآیند P_2 آن را مصرف کند، فرآیند P_2 باید منتظر باشد تا فرآیند P_1 ، داده مورد نیازش را آماده کند و بعد شروع به مصرف نماید.

مثال: همگام‌سازی دو فرآیند

دو فرآیند P_1 و P_2 را در نظر بگیرید که در یک سیستم تک پردازنده اشتراک زمانی به صورت هم روند اجرا می‌شوند. فرض کنید فرآیند P_1 در بخشی از کد خود مقدار متغیر x را می‌خواند و فرآیند P_2 نیز در بخشی از برنامه‌اش باید مقدار متغیر x خوانده شده توسط P_1 را چاپ کند. هدف مسأله، نوشتن این دو برنامه به صورتی است که P_2 در چاپ متغیر x بر P_1 پیشی نگیرد و اگر زودتر به بخش چاپ متغیر x رسید و هنوز مقدار متغیر x خوانده نشده بود، صبر کند تا P_1 متغیر x را مقداردهی کند. بنابراین، باید یک مسأله‌ی همگام‌سازی حل شود. واضح است که در این حالت تبادل داده به روش استفاده از حافظه‌ی مشترک است. یکی از راه حل‌های مناسب برای همگام‌سازی فرآیندهای همکار استفاده از سمافور می‌باشد که جلوتر شرح خواهیم داد.

راه حل‌های کنترل شرایط رقابتی

۱ـ راه حل‌های نرم‌افزاری (بر عهده‌ی برنامه‌نویس)
۲ـ راه حل‌های سخت‌افزاری
۳ـ راه حل‌های سیستم عامل (سمافور)
۴ـ راه حل‌های زبان‌های برنامه‌سازی (مانیتور)

توجه : راه حل‌های فوق، در سیستم‌های تک پردازنده و چند پردازنده با حافظه‌ی اشتراکی قابل استفاده‌اند.

راه حل‌های نرم‌افزاری

در این دسته راه حل‌ها، مسئولیت برقراری شرط انحصار متقابل بر عهده‌ی خود فرآیندها (کد نوشته شده توسط برنامه نویس) است و هیچ حمایتی از زبان برنامه‌سازی و سیستم عامل وجود ندارد.

۱ ـ راه حل متغیر قفل

یکی از راه حل‌های نرم‌افزاری کنترل شرایط رقابتی، استفاده از متغیر قفل می‌باشد. برای تجسم بهتر، یک باجه‌ی تلفن همگانی را به عنوان یک عامل مشترک در نظر بگیرید که قصد داریم شرط انحصار متقابل را برای آن برقرار کنیم. برای این کار، یک تابلوی اعلام وضعیت، در ورودی درب باجه نصب می‌کنیم. اگر تابلو عدد صفر را نمایش داد، بدین معنی است که فردی داخل نیست (باجه خالی است) و اگر تابلو عدد یک را نمایش داد، بدین معنی است که فردی داخل است (باجه پُر است).

اما در این راه حل ساده، شرط انحصار متقابل برقرار نیست، حالتی را در نظر بگیرید که تابلو به نشانه‌ی خالی بودن باجه، عدد صفر را نمایش می‌دهد، بنابراین ممکن است دو فرد، به شکل همزمان تابلوی وضعیت را به نشانه‌ی خالی بودن باجه مشاهده کنند و هر دو با هم اقدام به ورود به باجه تلفن (عامل مشترک) نمایند. بنابراین در این شرایط، شرط انحصار متقابل به عنوان شرط لازم شروط کنترل شرایط رقابتی نقض شده است.

شرح الگوریتم

فرض کنید یک متغیر قفل یکتای مشترک، با مقدار اولیه صفر موجود است. هنگامی که فرآیندی می‌خواهد وارد ناحیه‌ی بحرانی خود شود، ابتدا قفل را تست می‌کند، اگر قفل صفر باشد آن را برابر یک قرار می‌دهد و وارد ناحیه‌ی بحرانی خود می‌شود، ولی اگر قفل برابر یک بود، باید منتظر بماند تا قفل برابر صفر شود. بنابراین صفر به این معنی است که هیچ فرآیندی در ناحیه‌ی بحرانی قرار ندارد و یک به معنی این است که یک فرآیند در ناحیه‌ی بحرانی‌اش قرار دارد.

ساختار کلی این راه حل به صورت زیر می‌باشد:

P_0

```
P (int i) {
   while (TRUE) {
      while (lock ==1); /*loop*/
      lock=1;
      critical_section();
      lock=0;
      reminder_section();
      }
   }
```

P_1

```
P (int i) {
   while (TRUE) {
      while (lock ==1); /*loop*/
      lock=1;
      critical_section();
      lock=0;
      reminder_section();
      }
   }
```

توجه : قرار دادن کاراکتر؛ در انتهای حلقه while، سبب می‌شود حلقه تا زمانی که شرط برقرار است، در خود بچرخد.

اما این راه حل، با وجود اینکه خیلی ساده به نظر می‌رسد، ولی شرط انحصار متقابل را رعایت نمی‌کند. سناریوی زیر را در نظر بگیرید:

فرض کنید فرآیند P_0 ، در حلقه، متغیر قفل را می‌خواند و آن را برابر صفر می‌بیند، بنابراین شرط حلقه برقرار نیست و کنترل برنامه به خط بعد می‌رود، ولی قبل از آنکه بتواند مقدار یک را درون متغیر قفل قرار دهد، پردازنده به فرآیند P_1 تعویض متن می‌کند. حال فرآیند P_1 نیز متغیر قفل را برابر صفر

می‌بیند، بنابراین شرط حلقه برقرار نیست و کنترل برنامه به خط بعد می‌رود، در نتیجه مقدار متغیر قفل را برابر یک قرار می‌دهد و در ادامه وارد ناحیه‌ی بحرانی می‌شود. حال اگر دوباره در همین لحظه پردازنده به فرآیند P_0 تعویض متن کند، فرآیند P_0 نیز قفل را برابر یک قرار می‌دهد و وارد ناحیه‌ی بحرانی‌اش می‌شود، یعنی هر دو فرآیند P_0 و P_1 همزمان در عامل مشترک (ناحیه‌ی بحرانی) قرار دارند. این یعنی نقض شرط انحصار متقابل در شرایط رقابتی ما بین فرآیندها. پس شرط انحصار متقابل به عنوان شرط لازم در کنترل شرایط رقابتی برقرار نیست. بنابراین این راه حل مناسب نیست.

۲ ـ راه حل تناوب قطعی

یکی دیگر از راه حل‌های نرم‌افزاری کنترل شرایط رقابتی، استفاده از تناوب قطعی می‌باشد. برای تجسم بهتر، یک باجه‌ی تلفن را به عنوان یک عامل مشترک در نظر بگیرید که قصد داریم شرط انحصار متقابل را برای آن برقرار کنیم. برای این کار، این بار یک تابلوی اعلام نوبت برخلاف راه قبل که یک تابلوی اعلام وضعیت ناحیه‌ی بحرانی بود، در ورودی، درب باجه نصب می‌کنیم. اگر تابلو عدد صفر را نمایش داد، بدین معنی است که نوبت فرآیند P_0 است و اگر تابلو عدد یک را نمایش داد، بدین معنی است که نوبت فرآیند P_1 است. اما در این راه حل ساده، این بار شرط پیشروی برقرار نیست.

در شرط پیشرفت قرارمان این بود که اگر فردی داخل ناحیه‌ی باقی‌مانده بود، از ورود فرد دیگری به ناحیه‌ی بحرانی جلوگیری نکند، این همان تعریف شرط پیشرفت بود، اما این راه حل بر سر قرار پیشرفت نمانده است!

حالتی را در نظر بگیرید که تابلوی نوبت، عدد صفر را نشان می‌دهد، بنابراین در حال حاضر نوبت فرد P_0 برای ورود به باجه‌ی تلفن (عامل مشترک) است. پس P_0 وارد باجه‌ی تلفن می‌شود، و فرد دیگری حق ورود به باجه‌ی تلفن را ندارد، زیرا نوبتش نیست! این یعنی برقراری شرط انحصار متقابل. اکنون فرد P_0 داخل باجه قرار دارد و پس از آنکه تلفنش تمام شد، تابلوی نوبت را به یک مقداردهی می‌کند، از باجه خارج می‌شود و می‌رود دنبال کار کپی تا اسنادی را کپی نماید و مدت زیادی را مشغول این کار می‌ماند، در واقع P_0 در حال حاضر، در ناحیه‌ی باقی‌مانده کارهای خود قرار دارد.

در این لحظه فرد P_1 از راه می‌رسد، تابلوی نوبت را می‌بیند که مقدار آن برابر یک است، پس نوبتی هم که باشد این بار نوبت فرد P_1 است. بنابراین P_1 وارد باجه‌ی تلفن می‌شود، و فرد دیگری حق ورود به باجه‌ی تلفن را ندارد، زیرا نوبتش نیست! این یعنی برقراری انحصار متقابل. اکنون فرد P_1 داخل باجه قرار دارد و پس از آنکه تلفنش تمام شد، تابلوی نوبت را به صفر مقداردهی می‌کند و از باجه خارج می‌شود. حال دوباره نوبت فرد P_0 است. اما همانطور که گفتیم فرد P_0 در حال کپی اسنادی می‌باشد، همچنان هم کار کپی‌اش ادامه دارد...

یعنی فرد P_0 همچنان در ناحیه‌ی باقی‌مانده قرار دارد، در این لحظه فرد P_1 ناگهان تصمیم می‌گیرد تا

دوباره تلفن بزند، اما نمی‌تواند داخل باجه‌ی تلفن شود، چون نوبتش نیست، نوبتی هم که باشد این بار نوبت فرد P_0 است تا وارد باجه شود، پس فرد P_1 باید صبر پیشه کند، تا فرد P_0 کپی‌اش تمام شود و برود داخل باجه، تلفنش را بزند، کارش که تمام شد، تابلوی نوبت را به یک مقداردهی کند تا بالاخره نوبت فرد P_1 شود. نه، قرار این نبود، قرار این نبود که فردی که داخل ناحیه‌ی باقی‌مانده قرار دارد، از ورود فرد دیگری به باجه‌ی تلفن جلوگیری کند. دیدید که فرد P_0 با اینکه در ناحیه‌ی باقی‌مانده قرار داشت چگونه مانع ورود فرد P_1 به باجه‌ی تلفن شد. پس این راه حل هم مناسب نیست، به کار موقعیتی می‌آید، که مدام افراد نبوتی، تلفن می‌زنند و نوبت را سریع به طرف دیگر می‌سپارند، یعنی تناوب قطعی دارند.

شرح الگوریتم

P_0	P_1
P(int i){	P(int i){
while (TRUE)	while (TRUE)
{	{
① while (turn!=0); /*loop*/	while (turn!=1); /*loop*/
②/ critical_section ();	critical_section ();
③ turn=1;	turn=0;
④remainder_section ( );	remainder_section ( );
}	}
}	}

توجه : در این الگوریتم امکان ندارد دو فرآیند با هم وارد ناحیه‌ی بحرانی شوند، علت این امر به خاصیت الا کلنگی روش نوبت‌بندی مربوط می‌شود. بنابراین شرط انحصار متقابل برقرار است.

در این الگوریتم یک متغیر سراسری و مشترک به نام turn تعریف می‌شود که این متغیر، نوبت فرآیندها را برای ورود به ناحیه‌ی بحرانی نگه می‌دارد. زمانی که متغیر turn برابر صفر باشد، با اجرای فرآیند P_0، این فرآیند متغیر turn را بررسی و مقدار آن را مساوی صفر می‌بیند (خط ①)، بنابراین از حلقه‌ی while عبور کرده (چون شرط حلقه برقرار نیست) و وارد ناحیه‌ی بحرانی می‌گردد. حال اگر فرآیند P_1 نیز بخواهد به ناحیه‌ی بحرانی دسترسی داشته باشد با چک کردن مقدار این متغیر (در خط ①) منتظر مانده و مجوز ورود به ناحیه‌ی بحرانی به آن داده نمی‌شود، چون شرط حلقه در آن برقرار بوده و در اجرای دستور while می‌چرخد. بنابراین در داخل یک حلقه‌ی انتظار مشغول، بیکار می‌ماند و مرتباً turn را تست می‌کند تا ببیند چه موقع مقدار turn برابر یک می‌شود. زمانی که فرآیند P_0 از ناحیه‌ی بحرانی خارج می‌شود. مقدار turn را برابر یک قرار می‌دهد تا فرآیند P_1 بتواند وارد ناحیه‌ی بحرانی

شود.

مزایا

رعایت شرط انحصار متقابل

فرآیند P_0 وقتی وارد می‌شود که مقدار متغیر turn برابر صفر باشد و فرآیند P_1 وقتی وارد می‌شود که مقدار متغیر turn برابر یک باشد. بنابراین امکان ندارد فرآیندها با هم وارد ناحیه‌ی بحرانی شوند، زیرا مقدار متغیر turn یا صفر است و یا یک.

فرض کنید فرآیند P_0 در ناحیه‌ی بحرانی قرار دارد. یعنی مقدار متغیر turn برابر صفر است. از طرفی فرآیند P_1 تلاش می‌کند که به ناحیه‌ی بحرانی وارد شود. شرط ورود این فرآیند، یک شدن مقدار متغیر turn است، یعنی باید مقدار متغیر turn برابر یک شود، که این امر تنها در صورت خروج فرآیند P_0 از ناحیه‌ی بحرانی به وقوع می‌پیوندد. چنانچه فرآیند P_1 نیز در ناحیه‌ی بحرانی باشد، همین وضعیت برای فرآیند P_0 تکرار می‌شود. در نتیجه دو فرآیند هیچ‌گاه همزمان در ناحیه‌ی بحرانی قرار نمی‌گیرند، بنابراین شرط انحصار متقابل همواره برقرار است.

رعایت شرط انتظار محدود (عدم بن‌بست و قحطی)

در این الگوریتم، هر فرآیند پس از یک نوبت انتظار، می‌تواند وارد ناحیه‌ی بحرانی شود. در یک سناریو از اجرا، فرض کنید فرآیند P_1 منتظر ورود به ناحیه‌ی بحرانی است که چون اکنون P_0 در ناحیه‌ی بحرانی قرار دارد به آن اجازه‌ی ورود داده نمی‌شود. حال اگر فرایند P_0 از قسمت ناحیه‌ی بحرانی بگذرد و وارد بخش اعلام خروج شود، یعنی مقدار متغیر turn را یک کند و سپس بخواهد بدون توجه به درخواست فرآیند دیگر، مجدداً وارد ناحیه‌ی بحرانی گردد، به آن اجازه داده نمی‌شود و باید صبر کند تا P_1 یک بار اجرا شده و در انتهای کار خودش مقدار متغیر turn را صفر کند و مجوز ورود مجدد را به فرآیند P_0 بدهد. پس در این الگوریتم گرسنگی وجود ندارد. زیرا فرآیندها به صورت یک در میان و نوبتی وارد ناحیه‌ی بحرانی می‌شوند و نه تصادفی.

گرسنگی یعنی اینکه که یک فرآیند مدام کار کند و از کار کردن فرآیندی دیگر جلوگیری کند. در اینجا گرسنگی بدین معنی که یک فرآیند مدام داخل ناحیه‌ی بحرانی برود و از ورود فرآیندی دیگر جلوگیری کند وجود ندارد. از آن جا که در اجرای همروند و موازی این دو فرآیند، بن‌بست نیز وجود ندارد لذا شرط انتظار محدود برقرار است.

چگونگی امکان وقوع بن‌بست

هرگاه دو فرآیند متقاضی ورود به ناحیه‌ی بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه‌ی بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند، در این حالت بن‌بست رخ داده است.

معایب

عدم رعایت شرط پیشرفت

فرض کنید فرآیند P_1 یک باقی‌مانده‌ی سنگین دارد (مانند رسم یک نمودار طولانی) و مشغول انجام این کار در ناحیه‌ی باقی‌مانده است (خط ④). از طرفی P_0 در این لحظه از ناحیه‌ی بحرانی بیرون آمده و مقدار متغیر turn را در ناحیه خروج برابر یک قرار داده است و پس از مدت کمی، مجدداً درخواست ورود به ناحیه‌ی بحرانی را دارد. اما از آنجا که فرآیند P_1 هنوز در ناحیه‌ی باقی‌مانده خود به سرمی‌برد و نتوانسته است تاکنون با دستیابی مجدد به ناحیه‌ی بحرانی، مقدار متغیر turn را صفر کند (یعنی اجرای خط ③). بنابراین فرآیند P_0 مجبور است منتظر انجام بخش باقی‌مانده‌ی اجرای فرآیند مقدار P_1 که در ناحیه‌ی بحرانی قرار ندارد، شود. پس شرط پیشرفت در این الگوریتم نقض می‌گردد.

وجود پدیده‌ی گلوگاه

در این الگوریتم فرآیندها برای دستیابی به ناحیه‌ی بحرانی باید به صورت یک در میان عمل کنند، بنابراین سرعت اجرای عملیات توسط فرآیند کندتر هدایت می‌شود (گلوگاه).

مسأله انتظار مشغول

در این راه حل مشکل انتظار مشغول[1] وجود دارد، زیرا در زمانی که مثلاً فرآیند P_0 در ناحیه‌ی بحرانی قرار دارد، فرآیند P_1 زمان پردازنده را در چک کردن مقدار متغیر turn در خط ۱ ، در هر بار تعویض متن به فرآیند P_1 مصرف می‌کند و عملاً زمان پردازنده بیهوده هدر می‌رود.

توجه :روش نوبت گرفتن در مواقعی که یکی از فرآیندها خیلی کندتر است، ایده‌ی خوبی نیست.

توجه : در برخی از کتب مرجع و غیر مرجع، قبل از بیان راه حل پترسون، راه حل‌های ابتدایی‌تری موسوم به تلاش دوم، سوم و ... مطرح شده است، اما ما صلاح دیدیم ابتدا راه حل پترسون را معرفی کنیم و سپس قطعاتی از راه حل پترسون را برداریم تا به همان راه حل‌های ابتدایی قبل از پترسون برسیم، تا ارزش و قدرت راه حل پترسون هرچه بیشتر و بیشتر پر رنگ‌تر و با شکوه‌تر جلوه کند.

راه حل پترسون

این راه حل، ساده‌ترین و کوتاه‌ترین راه حل نرم‌افزاری است. در این راه حل از یک متغیر نوبت و یک تابلوی وضعیت دو یا چند حالته برای فرآیندها استفاده می‌گردد.

متغیر نوبت :برای نوبت گرفتن فرآیندها و ضایع نشدن حق یک فرآیند در ورود به ناحیه‌ی بحرانی مورد استفاده قرار می‌گیرد. بنابراین شرط انتظار محدود برقرار می‌شود.

1- Busy waiting

تابلوی وضعیت فرآیند: وضعیت فعلی یک فرآیند را برای فرآیند رقیب به نمایش می‌گذارد. تا اگر یکی از فرآیندها در ناحیه‌ی بحرانی نبود و یا قصد ورود به ناحیه‌ی بحرانی را نیز نداشت، یعنی در ناحیه‌ی باقی‌مانده قرار داشت، فرآیند دیگری بتواند به ناحیه‌ی بحرانی دسترسی داشته باشد. بنابراین شرط پیشروی برقرار می‌شود.

توجه : تابلوی وضعیت فرآیندها، سبب مستقل شدن فرآیندها از تناوب قطعی و خاصیت الاکلنگی می‌شود. این تابلو، دو وضعیت مختلف را برای یک فرآیند به نمایش می‌گذارد:

Flag[i] = FALSE : یعنی فرآیند مورد نظر در داخل ناحیه‌ی بحرانی قرار ندارد.

Flag[i] = TRUE : یعنی فرآیند مورد نظر علاقه‌مند است تا در صورت خالی بودن ناحیه‌ی بحرانی، وارد ناحیه‌ی بحرانی گردد. به عبارت دیگر علاقه‌مندی یک فرآیند برای ورود به ناحیه‌ی بحرانی را نشان می‌دهد.

شرط ورود به ناحیه‌ی بحرانی یک فرآیند در راه حل پترسون

شروط لازم

- فرآیند، علاقه‌مند به ورود به ناحیه‌ی بحرانی باشد، Flag [فرآیند علاقه‌مند] =TRUE
- فرآیند رقیب در ناحیه‌ی بحرانی نباشد، Flag [فرآیند رقیب] =FALSE

شرط کافی

- فرآیند مورد نظر، متغیر نوبت turn را زودتر مقداردهی کند. در واقع سرنوشت نهایی ورود فرآیندها به ناحیه‌ی بحرانی به متغیر نوبت turn گره خورده است. در یک قاعده‌ی کلی، در رقابت بر سر تصاحب عامل مشترک (ناحیه‌ی بحرانی) هر فرآیندی که علاقه‌مند به ورود به ناحیه‌ی بحرانی باشد (Flag [i]=TRUE) و سپس زودتر متغیر نوبت (turn=i) را مقداردهی کند، تحت هر شرایطی، تأکید می‌کنیم، تحت هر شرایطی زودتر وارد ناحیه‌ی بحرانی می‌شود و فرآیندی که دیرتر متغیر نوبت turn را مقدار دهی کند، پس از مقداردهی متغیر نوبت (turn=i)، باید بنشیند و در یک حلقه‌ی انتظار، صبر پیشه کند و مقدار متغیر نوبت turn را نگه‌داری کند تا فرآیند رقیب از ناحیه‌ی بحرانی خارج شود یعنی ناحیه‌ی بحرانی خالی گردد.

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
Boolean flag[2]= {FALSE, FALSE};
int turn;
```

P_0	P_1
P_0 (void) {	P_1 (void) {
while (TRUE)	while (TRUE)
{	{
①flag [0]= TRUE;	flag [1]= TRUE;
②turn = 0;	turn = 1;
③while (flag[1] && turn == 0); /*loop*/	while(flag[0] &&turn== 1); /*loop*/
④critical_section ();	critical_section ();
⑤flag [0] = FALSE;	flag [1] = FALSE;
remainder_section ();	remainder_section ();
}	}
}	}

مزایا

رعایت شرط انحصار متقابل

حالت اول (ورود غیرهمزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی باشد. یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی نباشد. پس در ابتدا Flag[0]=FALSE, Flag [1] = FALSE می‌باشد. فرض کنید فرآیند P_0 با مقداردهی Flag [0] = TRUE علاقه‌مندی خود را برای ورود به ناحیه‌ی بحرانی اعلام کند و در ادامه با مقداردهی متغیر turn برابر با صفر (turn=۰) به شکل یکه و تنها (اولی و آخری خودش است) پس از عبور از خط ③ به دلیل عدم برقراری شرط حلقه while (flag[1] && turn==0); وارد ناحیه‌ی بحرانی گردد.

FALSE TRUE

توجه کنید شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار بود. حال اگر پردازنده به فرآیند P_1 تعویض متن انجام دهد، فرآیند P_1 در حلقه‌ی انتظار مشغول خواهد ماند.

while (flag[0] & & turn = = 1);

TRUE TRUE

تا زمانی که فرآیند P_0 از ناحیه‌ی بحرانی خود خارج شود و Flag [0] = FALSE شود. توجه کنید که

شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_1 برقرار نبود. زیرا ناحیه‌ی بحرانی پُر است. بنابراین شرط انحصار متقابل برقرار است.

حالت دوم (ورود تقریباً همزمان فرآیندها)

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین تابلوی وضعیت فرآیندها هر دو Flag [0] = TRUE و Flag [1] = TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً عمزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. اما متغیر نوبت turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآیند، شماره‌ی فرآیند خود را در متغیر نوبت turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیره کند، فرآیندی است که شماره‌اش در متغیر نوبت turn باقی می‌ماند و دیگری اثرش در متغیر نوبت turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه‌ی بحرانی به متغیر نوبت turn گره خورده است (شرط کافی)، بنابراین فرآیندی که دیرتر متغیر نوبت turn را مقداردهی کرده است، باید صبر پیشه کند و متغیر نوبت turn را نگه‌داری کند و در یک حلقه‌ی انتظار بچرخد.

فرض کنید، فرآیند P_1 دیرتر اقدام به ورود به ناحیه‌ی بحرانی کند، بنابراین مقدار متغیر نوبت turn برابر با یک خواهد بود. $(turn = 1)$ وقتی که هر دو فرآیند به دستور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست.

```
while (flag [1] && turn == 0);
       TRUE          FALSE
```

بنابراین در حلقه نمی‌چرخد، پس فرآیند P_0 وارد ناحیه‌ی بحرانی می‌شود، اما خط ③ برای فرآیند P_1 برقرار است.

```
while (flag [0] && turn == 1);
       TRUE          TRUE
```

بنابراین در حلقه می‌چرخد و وارد ناحیه‌ی بحرانی نمی‌شود و می‌نشیند و صبر پیشه می‌کند و متغیر نوبت turn را نگه‌داری می‌کند. تا زمانی که فرآیند P_0 از ناحیه‌ی بحرانی خود خارج گردد و Flag [0] = FALSE شود. بنابراین شرط انحصار متقابل در این حالت هم برقرار است.

رعایت شرط پیشرفت

شرط پیشرفت می‌گفت، فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، در تصمیم‌گیری برای ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شرکت نکند. فرض کنید فرآیند P_1 در ناحیه‌ی باقی‌مانده‌ی سنگین خود قرار دارد (در حال حاضر در ناحیه‌ی بحرانی قرار ندارد و قصد ورود به ناحیه‌ی بحرانی را هم

ندارد)، مانند رسم یک نمودار طولانی، (بعد از خط ⑤) و در این زمان P_0 از ناحیه‌ی بحرانی خارج گردد و مجدداً درخواست این ناحیه را داشته باشد، اما فرآیند P_1 همچنان در ناحیه‌ی باقی‌مانده‌ی سنگین خود قرار دارد. از آنجا که فرآیند P_1 پس از خروج از ناحیه‌ی بحرانی و قبل از ورود به ناحیه‌ی باقی‌مانده‌ی سنگین خود مقدار Flag [1] را برابر FALSE قرار داده است (در خط ⑤)، پس باز هم فرآیند P_0 بدون وجود مانعی، وارد ناحیه‌ی بحرانی می‌شود. فرآیند P_0 داخل ناحیه‌ی بحرانی می‌گردد، در حالی که فرآیند P_1 در ناحیه‌ی باقی‌مانده قرار دارد، این یعنی پیشرفت. بنابراین شرط پیشرفت برقرار است.

توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار است.

while (flag[1] && turn == 0);

FALSE TRUE

رعایت شرط انتظار محدود

گرسنگی ندارد: گرسنگی یعنی اینکه یک فرآیند مدام کار کند و از کار کردن فرآیندی دیگر جلوگیری کند. در اینجا گرسنگی بدین معنی که یک فرآیند مدام داخل ناحیه‌ی بحرانی برود و از ورود فرآیندی دیگر به ناحیه‌ی بحرانی جلوگیری کند، وجود ندارد. سرنوشت ورود فرآیندها به متغیر نوبت turn گره خورده است، هر فرآیندی که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی، زودتر برایش برقرار باشد، زودتر وارد ناحیه‌ی بحرانی می‌گردد. بنابراین فرآیندی گرسنه نمی‌ماند.

فرض کنید فرآیند P_0 از ناحیه‌ی بحرانی خارج گردد و مقدار Flag [0] را برابر FALSE قرار دهد و برای اینکه فرآیند P_1 را گرسنه نگه دارد، مجدداً قصد ورود به ناحیه‌ی بحرانی را داشته باشد، در حالی که فرآیند P_1 در زمانی که فرآیند P_0 داخل ناحیه‌ی بحرانی بود، منتظر خالی شدن ناحیه‌ی بحرانی بود و نوبت گرفته بود، فرآیند P_0 می‌خواهد، فرآیند P_1 را گرسنه نگه دارد ولی نمی‌تواند، زیرا شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرایند P_0 برقرار نیست.

while (flag [0] & & turn == 0);

TRUE TRUE

زیرا P_1 زودتر از P_0 نوبت گرفته است و باعث می‌شود مقدار متغیر نوبت turn برابر صفر (turn = ۰) باشد. زیرا مقدار صفر مربوط به فرآیند P_0 که دیرتر آمده است در متغیر نوبت turn قرار می‌گیرد. بنابراین فرآیند P_1 وارد ناحیه‌ی بحرانی می‌گردد، زیرا شروط لازم و کافی برای فرآیند P_1 برقرار است.

while (flag [0] && turn = = 1);

FALSE FALSE

لذا این بار فرآیند P_0 باید بنشیند و صبر پیشه کند و مقدار متغیر نوبت turn را نگه‌داری کند تا فرآیند P_1 از ناحیه‌ی بحرانی خارج شود یعنی ناحیه‌ی بحرانی خالی گردد.

بن‌بست ندارد

زیرا پس از اجرای موازی یا همروند خطوط ① و ② ، با توجه به مقدار متغیر نوبت turn، در نهایت یکی از فرآیندها به طور قطع بسته به اینکه چه کسی متغیر turn را زودتر مقداردهی کرده است، وارد ناحیه‌ی بحرانی می‌گردد و هیچگاه دو فرآیند در خط ③ به طور همزمان مسدود نمی‌شوند، در نتیجه بن‌بست ندارد. بنابراین شرط انتظار محدود برقرار است.

معایب

۱ـ عدم رعایت شرط انتظار مشغول

در این راه حل مشکل انتظار مشغول وجود دارد، زیرا زمانی که مثلاً فرآیند P_0 در ناحیه‌ی بحرانی قرار دارد، فرآیند P_1 در یک حلقه‌ی انتظار زمان پردازنده را در بررسی برقراری شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی در خط ③ به هدر می‌دهد.

ناپترسون‌ها

در ادامه با تغییراتی در راه حل پترسون، راه حل‌هایی را ایجاد می‌کنیم که هیچ یک راه حل جامعی نخواهند بود، اما سبب می‌شود تا ارزش راه حل پترسون بیشتر شناخته شود.

ناپترسون اول

حذف متغیر نوبت turn از راه حل پترسون. این راه حل، به سومین تلاش نیز موسوم است.

در این راه حل تنها از یک تابلوی وضعیت دو یا چند حالته برای فرآیندها استفاده می‌گردد.

تابلوی وضعیت فرآیند: وضعیت فعلی یک فرآیند را برای فرآیند رقیب به نمایش می‌گذارد. تا اگر یکی از فرآیندها در ناحیه‌ی بحرانی نبود و یا قصد ورود به ناحیه‌ی بحرانی را نیز نداشت، یعنی در ناحیه‌ی باقی‌مانده قرار داشت، فرآیند دیگری بتواند به ناحیه‌ی بحرانی دسترسی داشته باشد. بنابراین شرط پیشروی برقرار است.

توجه : تابلوی وضعیت فرآیندها، سبب مستقل شدن فرآیندها از تناوب قطعی و خاصیت الاکلنگی می‌شود. این تابلو، دو وضعیت مختلف را برای یک فرآیند به نمایش می‌گذارد:

Flag [i] = FALSE: یعنی فرآیند مورد نظر در داخل ناحیه‌ی بحرانی قرار ندارد.

Flag [i] = TRUE: یعنی فرآیند مورد نظر علاقه‌مند است تا در صورت خالی بودن ناحیه‌ی بحرانی، وارد ناحیه‌ی بحرانی گردد. به عبارت دیگر علاقه‌مندی یک فرآیند برای ورود به ناحیه‌ی بحرانی را نشان می‌دهد.

شرط ورود به ناحیه‌ی بحرانی یک فرآیند در این راه حل

شرط لازم

• فرآیند، علاقه‌مند به ورود به ناحیه‌ی بحرانی باشد، Flag [فرآیند علاقه‌مند] = TRUE

شرط کافی

• فرآیند رقیب در ناحیه‌ی بحرانی نباشد، Flag [فرآیند رقیب] = FALSE

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
Boolean flag[2]= {FALSE, FALSE};
```

P۰

```
P۱(void){
   while (TRUE);
   {
   ①flag [0]= TRUE;
   ②while (flag [1]); /*loop*/
   ③critical_section ( );
   ④flag [0] = false;
   remainder_section ( );
   }
}
```

P۱

```
P۱(void){
while (TRUE);
{
flag [1]= TRUE;
while (flag [0]); /*loop*/
 critical_section ( );
flag [1] = false;
remainder_section ();
}
}
```

توجه : در این راه حل ابتدا فرآیند، علاقه‌مندی خود مبنی بر ورود به ناحیه‌ی بحرانی را با مقداردهی flag[i]=TRUE اعلام می‌کند، سپس وضعیت فرآیند مقابل، بررسی می‌شود.

مزایا

رعایت شرط انحصار متقابل

حالت اول (ورود غیرهمزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی باشد، یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی نباشد. پس در ابتدا flag [1] = FALSE و flag [0] = FALSE می‌باشد. فرض کنید، فرآیند P_0 با مقداردهی flag[0] = TRUE علاقه‌مندی خود را برای ورود به ناحیه‌ی بحرانی اعلام کند و در ادامه پس از عبور از خط ② به دلیل عدم برقراری شرط حلقه while (flag[1]); وارد ناحیه‌ی بحرانی گردد.

FALSE

توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار بود. حال اگر

پردازنده به فرآیند P_1 تعویض متن انجام دهد، فرآیند P_1 در حلقه‌ی انتظار مشغول خواهد ماند.

while (flag [0]);

TRUE

تا زمانی که فرآیند P_0 از ناحیه‌ی بحرانی خود خارج شود و flag [0] = FALSE شود. توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرایند P_1 برقرار نبود. زیرا ناحیه‌ی بحرانی پُر است. بنابراین شرط انحصار در حالت اول برقرار است.

حالت دوم (ورود تقریباً همزمان فرآیندها)

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین تابلوی وضعیت فرآیندها هر دو برابر Flag [0] = TRUE و Flag [1] = TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند.

وقتی که هر دو فرآیند به طور تقریباً همزمان به دستور while می‌رسند، شرط خط ②

برای فرآیند P_0

while (flag [0]);

TRUE

برای فرآیند P_1

while (flag [1]);

TRUE

برقرار است. بنابراین هر دو فرآیند P_0 و P_1 در حلقه دور می‌زنند و وارد ناحیه‌ی بحرانی نمی‌شوند. بنابراین شرط انحصار متقابل برقرار است.

اما هر دو فرآیند، تا ابد گرفتار حلقه‌ی انتظار مشغول شده‌اند، بنابراین بن‌بست رخ داده است و شرط انتظار محدود برقرار نیست.

رعایت شرط پیشرفت

شرط پیشرفت می‌گفت، فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، در تصمیم‌گیری برای ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شرکت نکند.

فرض کنید فرآیند P_1 در ناحیه‌ی باقی‌مانده‌ی سنگین خود قرار دارد (در حال حاضر در ناحیه‌ی بحرانی قرار ندارد و قصد ورود به ناحیه‌ی بحرانی را هم ندارد)، مانند رسم یک نمودار طولانی، (بعد از خط ④) و در این زمان P_0 از ناحیه‌ی بحرانی خود خارج گردد و مجدداً درخواست این ناحیه را داشته باشد، اما فرایند P_1 همچنان در ناحیه‌ی باقی‌مانده سنگین خود قرار دارد. از آنجا که فرآیند P_1 پس از خروج از ناحیه‌ی بحرانی و قبل از ورود به ناحیه‌ی باقی‌مانده‌ی سنگین خود، مقدار Flag [1] را

برابر FALSE قرار داده است (در خط ④)، پس باز هم فرآیند P_0 بدون وجود مانعی، وارد ناحیه‌ی بحرانی می‌شود. فرآیند P_0 داخل ناحیه‌ی بحرانی می‌گردد، در حالی که فرآیند P_1 در ناحیه‌ی باقی‌مانده قرار دارد، این یعنی پیشرفت. بنابراین شرط پیشرفت برقرار است.
توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار است.

while (flag [1]);
FALSE

معایب

عدم رعایت شرط انتظار محدود

بن‌بست دارد: زیرا در ورود تقریباً همزمان فرآیندها، بن‌بست وجود داشت.

ناپترسون دوم

حذف متغیر نوبت turn و جابه‌جا کردن خطوط ① و ③ از راه حل پترسون. این راه حل به دومین تلاش نیز موسوم است. در این راه حل تنها از یک تابلوی وضعیت دو یا چند حالته برای فرآیندها استفاده می‌گردد.
تابلوی وضعیت فرآیند: وضعیت فعلی یک فرآیند را برای فرآیند رقیب به نمایش می‌گذارد. تا اگر یکی از فرآیندها در ناحیه‌ی بحرانی نبود و یا قصد ورود به ناحیه‌ی بحرانی را نیز نداشت یعنی در ناحیه‌ی باقی‌مانده قرار داشت، فرآیند دیگری بتواند به ناحیه‌ی بحرانی دسترسی داشته باشد. بنابراین شرط پیشروی برقرار است.
توجه : تابلوی وضعیت فرآیندها، سبب مستقل شدن فرآیندها از تناوب قطعی و خاصیت الاکلنگی می‌شود. این تابلو، دو وضعیت مختلف را برای یک فرآیند به نمایش، می‌گذارد:
Flag [i] = FALSE: یعنی فرآیند مورد نظر در داخل ناحیه‌ی بحرانی قرار ندارد.
Flag [i] = TRUE: یعنی فرآیند مورد نظر علاقه‌مند است تا در صورت خالی بودن ناحیه‌ی بحرانی، وارد ناحیه‌ی بحرانی گردد. به عبارت دیگر علاقه‌مندی یک فرآیند برای ورود به ناحیه‌ی بحرانی را نشان می‌دهد.

شرط ورود به ناحیه‌ی بحرانی یک فرآیند در این راه حل

شرط لازم

فرآیند، علاقه‌مند به ورود به ناحیه‌ی بحرانی باشد، Flag [فرآیند علاقه‌مند] = TRUE

شرط کافی

فرآیند رقیب در ناحیه‌ی بحرانی نباشد، Flag [فرآیند رقیب] = FALSE

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
Boolean flag [2]= {FALSE, FALSE};
```

```
P۰
P۰(void){
   while (TRUE)
   {
   ①while (flag [1]);/*loop*/
   ②flag [0]= TRUE;
   ③ critical_section ();
   ④flag [0] = FALSE;
    remainder_section ();
   }
 }
```

```
P۱
P۱(void){
   while (TRUE)
   {
   while (flag [0]);/*loop*/
   flag [1]= TRUE;
    critical_section ();
   flag [1] = FALSE;
    remainder_section ();
   }
 }
```

توجه : در این راه حل ابتدا وضعیت فرآیند مقابل بررسی می‌شود و بعد از آن فرآیند، تابلوی وضعیت مربوط به خود را به نشانه‌ی علاقه‌مندی به ورود به ناحیه‌ی بحرانی برابر TRUE مقداردهی می‌کند.

مزایا

رعایت شرط پیشرفت

شرط پیشرفت می‌گفت، فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، در تصمیم‌گیری برای ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شرکت نکند.

فرض کنید فرآیند $P_۱$ در ناحیه‌ی باقی‌مانده‌ی سنگین خود قرار دارد (در حال حاضر در ناحیه‌ی بحرانی قرار ندارد و قصد ورود به ناحیه‌ی بحرانی را هم ندارد.) مانند رسم یک نمودار طولانی، (بعد از خط ④و در این زمان $P_۰$ از ناحیه‌ی بحرانی خود خارج گردد و مجدداً درخواست این ناحیه را داشته باشد، اما فرآیند $P_۱$ همچنان در ناحیه‌ی باقی‌مانده‌ی سنگین خود قرار دارد. از آنجا که فرآیند $P_۱$ پس از خروج از ناحیه‌ی بحرانی و قبل از ورود به ناحیه‌ی باقی‌مانده‌ی سنگین خود، مقدار Flag [1] را برابر FALSE قرار داده است (در خط ④، پس باز هم فرآیند $P_۰$ بدون وجود مانعی وارد ناحیه‌ی بحرانی می‌شود. فرآیند $P_۰$ داخل ناحیه‌ی بحرانی می‌گردد، در حالی که فرآیند $P_۱$ در ناحیه‌ی باقی‌مانده قرار دارد، این یعنی پیشرفت. بنابراین شرط پیشرفت برقرار است.

توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند $P_۰$ برقرار است.

while (flag [1]);
FALSE

معایب

عدم رعایت شرط انحصار متقابل

حالت اول (ورود غیرهمزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی باشد، یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی نباشد. پس در ابتدا Flag [1] = FALSE و Flag [0] = FALSE می‌باشد. فرض کنید، ابتدا پردازنده در اختیار فرآیند P_0 باشد، بنابراین فرآیند P_0 پس از عبور از خط ① به دلیل عدم برقراری شرط حلقه while (Flag [1]); وارد خط ② می‌شود و با مقداردهی Flag [0] = TRUE علاقه‌مندی خود را FALSE برای ورود به ناحیه‌ی بحرانی اعلام می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد.

توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار بود. حال اگر پردازنده به فرآیند P_1 تعویض متن انجام دهد، فرآیند P_1 در حلقه‌ی انتظار مشغول خواهد ماند.

while (flag [0]);
TRUE

تا زمانی که فرآیند P_0 از ناحیه‌ی بحرانی خود خارج شود و Flag [0] = FALSE شود. توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_1 برقرار نبود. زیرا ناحیه‌ی بحرانی پُر است.

بنابراین شرط انحصار متقابل در حالت اول برقرار است.

حالت دوم (ورود تقریباً همزمان فرآیندها)

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. به صورت پیش فرض تابلوی وضعیت هر دو فرآیند هر دو Flag [0] = FALSE و Flag [1] = FALSE می‌باشد.

وقتی که هر دو فرآیند به طور تقریباً همزمان به دستور while می‌رسند خط ① برای فرآیند P_0 while (Flag [0]); و برای فرآیند P_1، whlie (Flag [1]); برقرار نیست.

FALSE FALSE

بنابراین هر دو فرآیند P_0 و P_1 به طور تقریباً همزمان پس از اعلام علاقه‌مندی خود برای ورود به ناحیه‌ی بحرانی با مقداردهی Flag [0] = TRUE برای فرآیند P_0 و Flag [1] = TRUE برای فرآیند P_1، در خط ② در ادامه وارد ناحیه‌ی بحرانی می‌شوند. بنابراین شرط انحصار متقابل در حالت دوم برقرار نیست.

پس در کل، شرط انحصار متقابل برای این راه حل برقرار نیست.

عدم رعایت شرط انتظار محدود

گرسنگی دارد: گرسنگی یعنی اینکه یک فرآیند مدام کار کند و از کار کردن فرآیندهای دیگر جلوگیری کند. در اینجا گرسنگی بدین معنی که یک فرآیند مدام داخل ناحیه‌ی بحرانی برود و از ورود فرآیندی دیگر به ناحیه‌ی بحرانی جلوگیری کند، وجود دارد.

فرض کنید فرآیند P_0 در ناحیه‌ی بحرانی است و با انجام تعویض متن، پردازنده به P_1 داده شود. در این لحظه فرآیند P_1 مشغول بررسی Flag [0] برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی است که اجازه‌ی ورود به آن داده نمی‌شود و تا پایان کوانتوم خود می‌چرخد، چون مقدار Flag [0] برابر TRUE است. فرض کنید در پایان کوانتوم، با انجام تعویض متن مجدد، پردازنده از فرآیند P_1 گرفته شود و دوباره به فرآیند P_0 داده شود و فرآیند P_0 پس از اجرای ناحیه‌ی بحرانی‌اش، سریعاً ناحیه‌ی باقی‌مانده را پشت سرگذاشته و بخواهد مجدداً وارد ناحیه‌ی بحرانی شود. شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار است، بنابراین فرآیند P_0 مشکلی برای ورود مجدد ندارد، بنابراین مقدار Flag [0] توسط فرآیند P_0، مجدداً TRUE می‌شود. حال اگر در این لحظه، بر اثر تعویض متن پردازنده مجدداً به فرآیند P_1 برسد، از آنجا که Flag [0] قبلاً توسط فرآیند P_0 مقدارش TRUE شده است، پس شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_1 برقرار نیست، لذا به فرآیند P_1 باز هم اجازه‌ی ورود به ناحیه‌ی بحرانی داده نخواهد شد و باز هم P_0 ناحیه‌ی بحرانی را در اختیار دارد. در نتیجه در این راه حل، دستیابی به ناحیه‌ی بحرانی شانسی و تصادفی است و این سناریو می‌تواند بارها و بارها تکرار شود و در نهایت، سبب گرسنگی فرآیند P_1 گردد. بنابراین شرط انتظار محدود به دلیل گرسنگی برقرار نیست.

جهت درک بیشتر، یک بار دیگر به الگوریتم مطرح شده، نگاه کنید، واقعیت این است که هر فرآیند برای ورود به ناحیه‌ی بحرانی ابتدا وضعیت فرآیند مقابل را بررسی می‌کند و بعد از آن، فرآیند، مقدار تابلوی وضعیت مربوط به خود را به نشانه‌ی علاقه‌مندی به ورود به ناحیه‌ی بحرانی برابر TRUE قرار می‌دهد، بنابراین این الگوریتم، ابزاری برای جلوگیری و ممانعت از ورود فرآیندهای رقیب به ناحیه‌ی بحرانی که چابک‌تر هستند، در اختیار ندارد.

مثال واقعی این الگوریتم در عالم زندگی نیز وجود دارد، مانند فردی خجول که برای بدست آوردن عامل مشترک مورد علاقه‌ی خود، ابتدا وضعیت عامل مشترک را بررسی می‌کند و بعد علاقه‌مندی خود را اعلام می‌کند، این فرد در مواجه با افراد چابک‌تر مدام دچار گرسنگی می‌شود! در راه حل قبل ابتدا افراد علاقه‌مندی خود را اعلام می‌کردند و بعد وضعیت عامل مشترک را بررسی می‌کردند، که دیدید بن‌بست را به ارمغان می‌آورد! یکی برای همه به صورت همزمان، این یعنی بن‌بست، دیدید که

هیچ یک از این راه حل‌ها مناسب نبودند، هم باید علاقه‌مندی برای تصاحب عامل مشترک اعلام گردد و هم نوبت رعایت گردد. البته به شرطی که عامل مشترک خالی باشد. **راه حل پترسون همین راگفته بود.**
بن‌بست ندارد: زیرا در ورود تقریباً همزمان فرآیندها بن‌بست وجود نداشت.

راه حل‌های سخت‌افزاری

در این دسته راه حل‌ها، مسئولیت برقراری شرط انحصار متقابل بر عهده‌ی برنامه‌نویس و سخت‌افزار است.

دستورالعمل از کار انداختن وقفه‌ها

ساده‌ترین راه حل، آن است که هر فرآیند بلافاصله پس از ورود به ناحیه‌ی بحرانی‌اش، تمام وقفه‌ها را از کار بیندازد و دقیقاً قبل از خروج از ناحیه‌ی بحرانی همه‌ی آن‌ها را مجدداً فعال سازد. با متوقف کردن وقفه‌ها، دیگر وقفه‌های ساعت نیز رخ نخواهند داد. پردازنده فقط در نتیجه‌ی وقوع وقفه‌های ساعت و یا وقفه‌های ورودی و خروجی است که می‌تواند از یک فرآیند به فرآیندی دیگر تعویض متن کند. پس با از کار انداختن وقفه‌ها، پردازنده دیگر تحت هیچ شرایطی قادر نخواهد بود از فرآیندی به فرآیند دیگر تعویض متن کند. بنابراین هنگامی که یک فرآیند وقفه‌ها را غیرفعال می‌کند، می‌تواند بدون هیچ مشکلی و بدون ترس از مداخله‌ی دیگر فرآیندها به خواندن و نوشتن در حافظه‌ی مشترک بپردازد. پس برای رعایت شرط انحصار متقابل **در سیستم‌های تک پردازنده‌ای** کافی است، وقفه‌ها متوقف شوند.
ساختار کلی این راه حل به صورت زیر می‌باشد:

```
P (int i) {
   while (TRUE)
   {
   Disable_Interupts ();
   critcial_section ();
   Enable_Interupts );
   remainder_section ();
   }
}
```

معایب

۱ـ اگر کاربر وقفه‌ها را خاموش کند ولی بعد از اجرای ناحیه‌ی بحرانی، فراموش کند، مجدداً وقفه‌ها را فعال نماید، سبب از کار افتادن وقفه و در نتیجه کل سیستم می‌شود. پس اعطای قدرت متوقف کردن وقفه‌ها به فرآیند کاربر از نظر معیار امنیت سیستم، عاقلانه نیست.

توجه :از کار انداختن وقفه، در سیستم‌های تک پردازنده‌ای، اغلب در داخل خود سیستم عامل برای مدیریت نواحی بحرانی فرآیندهای سیستم عامل، تکنیک مفیدی است، اما برای مدیریت نواحی بحرانی فرآیندهای کاربر، به دلیل فراموش کار بودن کاربر، تکنیک مناسبی نیست.

۲ـ در سیستم‌های چند پردازنده‌ای، غیرفعال کردن وقفه‌ها، فقط در پردازنده‌ای اثر دارد که دستورالعمل از کار انداختن وقفه را اجرا می‌کند. پس پردازنده‌های دیگر به کارشان ادامه می‌دهند. از آنجایی که ممکن است بیش از یک فرآیند در هر لحظه در حال اجرا باشد، ممکن است فرآیندهای دیگر که روی پردازنده‌های دیگر در حال اجرا هستند نیز وارد ناحیه‌ی بحرانی شوند. به عبارت دیگر، در سیستم‌های چند پردازنده‌ای، این ذات توازی است که باعث ورود همزمان فرآیندها به ناحیه‌ی بحرانی می‌شود و نه وقوع وقفه، که با جلوگیری از آن، بخواهیم مشکل را حل کنیم. پس در سیستم‌های چند پردازنده‌های ممکن است شرط انحصار متقابل برقرار نباشد.

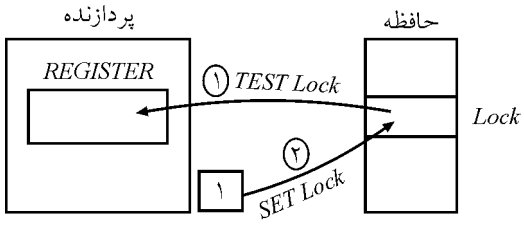
دستورالعمل TSL

راه حل متغیر قفل را به یادآورید، علت عدم موفقیت آن راه حل، عدم قدرت کافی، برای برقراری شرط انحصار متقابل بود و این عدم قدرت، ناشی از اتمیک نبودن (تجزیه‌پذیر بودن) دو دستور خواندن قفل (TEST Lock) و مقداردهی قفل (SET Lock) می‌باشد. در آن راه حل، این امکان وجود داشت که یک فرآیند زمانی که متغیر قفل را می‌خواند و آن را برابر صفر می‌بیند و با ناحیه‌ی بحرانی خالی مواجه می‌شود به جای آنکه فوراً متغیر قفل را به یک مقداردهی کند و سد راه ورود فرآیند رقیب به ناحیه‌ی بحرانی گردد، ممکن است این امکان فراهم شود که پردازنده قبل از آنکه فرآیند فعلی متغیر قفل را به نشانه‌ی تصاحب ناحیه‌ی بحرانی به یک مقداردهی کند، به فرآیند رقیب تعویض متن کند و باعث شود فرآیند رقیب نیز متغیر قفل را بخواند و مقدار آن را برابر صفر ببیند و آن هم با ناحیه‌ی بحرانی خالی مواجه شود و مطابق آنچه پیش از این نیز گفتیم، در ادامه هر دو فرآیند وارد ناحیه‌ی بحرانی شوند.

اگر بتوانیم کاری کنیم که دو عمل خواندن و مقداردهی متغیر قفل به صورت اتمیک (تجزیه‌ناپذیر) انجام شود، مسأله حل می‌شود، دیدید که از کار انداختن وقفه‌ها هم نتوانست یک راه حل عمومی باشد. بسیاری از پردازنده‌ها، دستورالعمل دو بخشی اما اتمیک خاصی دارند به نام TSL (Test and Set Lock)، بدین نحو که این دستورالعمل به شکل تجزیه‌ناپذیر، عملیات زیر را انجام می‌دهد:

- ابتدا محتویات یک کلمه از حافظه به نام Lock را می‌خواند و مقدار آن را در رجیستر قرار می‌دهد. (TEST Lock)
- سپس مقدار یک را در متغیر Lock قرار می‌دهد. (SET Lock)

به شکل زیر توجه کنید:



سخت‌افزار تضمین می‌کند که دو عمل خواندن و مقداردهی متغیر قفل به صورت اتمیک (تجزیه‌ناپذیر) انجام شود. بدین شکل که هیچ فرآیند و حتی پردازنده‌ی دیگری نتواند به این متغیر قفل دسترسی پیدا کند تا وقتی که اجرای دستورالعمل به پایان برسد. پردازنده‌ای که دستورالعمل TSL را اجرا می‌کند، گذرگاه حافظه را قفل می‌کند تا از دسترسی دیگر پردازنده‌ها به حافظه جلوگیری کند تا اینکه این دستورالعمل به پایان برسد.

ساختار کلی این راه حل، به صورت زیر می‌باشد:

```
enter_section ();          enter_section              exit_section
critical_section ();        TSL REGISTER , LOCK        MOVE LOCK , 0
exit_section ();           CMP REGISTER , 0           RET
remainder_section ();      JNE enter-section
                           RET
```

توجه مهم :در این راه حل شرط ورود به ناحیه‌ی بحرانی، اجرای زودتر دستور TSL است.

در این الگوریتم، فرآیندها برای کسب اجازه و ورود به ناحیه‌ی بحرانی از تابع enter_section و برای خروج از ناحیه‌ی بحرانی از تابع exit_section استفاده می‌کنند. اگر فرآیندی علاقه‌مند به ورود به ناحیه‌ی بحرانی باشد. ابتدا تابع enter_section را صدا می‌زند تا بررسی‌های لازم جهت فراهم بودن یا نبودن ورود به ناحیه‌ی بحرانی انجام گردد. بدین نحو که ابتدا، توسط دستور TSL و به شکل اتمیک مقدار متغیر Lock در رجیستر قرار داده می‌شود، سپس مقدار متغیر Lock برابر یک مقداردهی می‌شود. سپس مقدار رجیستر که حاوی مقدار متغیر Lock می‌باشد با صفر مقایسه می‌شود. اگر مقدار رجیستر برابر صفر باشد به معنی خالی بودن ناحیه‌ی بحرانی است و در ادامه دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن مقدار رجیستر با صفر، انجام نمی‌شود، زیرا مقدار رجیستر برابر صفر است. و در ادامه دستور RET (RETURN) باعث می‌شود تا تابع تمام شود. بنابراین فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، وارد ناحیه‌ی بحرانی می‌گردد. و پس از آنکه فرآیند کارش با ناحیه‌ی بحرانی تمام شد، تابع exit_section را به نشانه‌ی خروج از ناحیه‌ی بحرانی صدا می‌زند، اجرای این تابع سبب می‌شود تا مقدار متغیر Lock توسط دستور MOVE برابر صفر گردد، صفر بودن مقدار متغیر Lock به معنی خالی بودن ناحیه‌ی بحرانی است، بنابراین این امکان فراهم می‌شود تا

فرآیندهای دیگر بتوانند پس از کسب اجازه توسط تابع enter-section وارد ناحیه‌ی بحرانی شوند.
در طرف مقابل اگر در حین اجرای تابع enter_section، مقدار متغیر Lock برابر یک باشد، به معنی پُر بودن ناحیه‌ی بحرانی است و در ادامه شرط دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن رجیستر با مقدار صفر برقرار است، زیرا مقدار رجیستر برابر یک است. بنابراین یک حلقه‌ی انتظار مشغول تا خالی شدن ناحیه‌ی بحرانی ایجاد می‌گردد. این حلقه‌ی انتظار مانع ورود فرآیند رقیب به ناحیه‌ی بحرانی می‌گردد.

مزایا

رعایت شرط انحصار متقابل

حالت اول (ورود غیرهمزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی باشد، یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی نباشد. پس در ابتدا مقدار متغیر Lock برابر صفر است.
فرض کنید، فرآیند P_0 با فراخوانی تابع enter_section علاقه‌مندی خود را برای ورود به ناحیه‌ی بحرانی اعلام کند، این تابع دستور TSL (دو عمل خواندن و مقداردهی متغیر قفل) را به صورت اتمیک انجام می‌دهد، بنابراین فرآیند P_0، ناحیه‌ی بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد.
توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار بود.
حال اگر پردازنده به فرآیند P_1 تعویض متن کند، این فرآیند نیز تابع enter_section را فراخوانی می‌کند و در دستور CMP مقدار رجیستر را برابر یک می‌بیند و مطابق آنچه پیش از این نیز گفتیم شرایط برای فرآیند P_1 به گونه‌ای رقم می‌خورد که باید در حلقه‌ی انتظار مشغول، مشغول باشد. تا زمانی که فرآیند P_0 از ناحیه‌ی بحرانی خود خارج گردد و تابع exit_section را فراخوانی کند تا مقدار متغیر قفل برابر صفر گردد. (شرایط ورود به ناحیه‌ی بحرانی فراهم گردد) توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_1، برقرار نیست. (ناحیه‌ی بحرانی پُر است). بنابراین شرط انحصار متقابل برقرار است.

حالت دوم (ورود تقریباً همزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی است، یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی نباشد. پس در ابتدا مقدار متغیر Lock برابر صفر است. در این حالت هر دو فرآیند برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی تابع enter_section را به شکل همزمان فراخوانی می‌کنند. از آن جا که دستور TSL (دو عمل خواندن و مقداردهی قفل) به صورت اتمیک انجام می‌گردد. در نهایت یک فرآیند خوش شانس‌تر که کمی زودتر پردازنده را برای اجرای دستور TSL در اختیار بگیرد، سرانجام ناحیه‌ی

بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد و دیگری باید در حلقه‌ی انتظار مشغول، مشغول باشد. بنابراین شرط انحصار متقابل در حالت دوم نیز برقرار است.

رعایت شرط پیشرفت

شرط پیشرفت می‌گفت، فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، در تصمیم‌گیری برای ورود فرآیندهای دیگر به ناحیه‌ی بحرانی نباید شرکت کند در این راه حل یک فرآیند پس از خروج از ناحیه‌ی بحرانی و قبل از ورود به ناحیه‌ی باقی‌مانده، توسط فراخوانی تابع exit_section (مقداردهی متغیر قفل به صفر) ناحیه‌ی بحرانی را خالی اعلام می‌کند، در واقع مانعی که سد راه ورود فرآیندهای دیگر به ناحیه‌ی بحرانی بود، از روی ناحیه‌ی بحرانی برمی‌دارد و در ادامه برای رسیدگی به کارهای دیگرش، در ناحیه‌ی باقی‌مانده قرار می‌گیرد، بنابراین فرآیندهای رقیب می‌توانند پس از کسب اجازه‌ی ورود به ناحیه‌ی بحرانی توسط تابع enter_section و مستقل از فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، وارد ناحیه‌ی بحرانی شوند. بنابراین شرط پیشرفت برقرار است.

معایب

عدم رعایت شرط انتظار محدود

گرسنگی دارد: گرسنگی یعنی اینکه یک فرآیند مدام کار کند و از کار کردن فرآیندهای دیگر جلوگیری کند. در اینجا گرسنگی بدین معنی که یک فرآیند مدام داخل ناحیه‌ی بحرانی برود و از ورود فرآیندهای دیگر به ناحیه‌ی بحرانی جلوگیری کند، وجود دارد. از آن جا که این الگوریتم فاقد مکانیزم نوبت‌دهی است. بنابراین ورود به ناحیه‌ی بحرانی تصادفی خواهد بود، پس یک فرآیند خوش‌شانس‌تر و با بخت و اقبال بالاتر (منظور از شانس، چگونگی تعویض متن پردازنده است) می‌تواند، بارها و بارها پردازنده را بدست آورد و وارد ناحیه‌ی بحرانی گردد، بدون آنکه فرآیند رقیب بتواند کاری از پیش ببرد، این یعنی گرسنگی زیرا سرنوشت ورود فرآیندها به بخت و اقبال فرآیندها گره خورده است، و هیچ گونه نوبتی رعایت نمی‌شود، بنابراین شرط انتظار محدود به دلیل گرسنگی برقرار نیست.

بن‌بست ندارد:

زیرا پس از اجرای موازی یا تقریباً همزمان، هر دو فرآیند برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی تابع enter_section را به شکل همزمان فراخوانی می‌کنند. از آن جا که دستور TSL (دو عمل خواندن و مقداردهی قفل) به صورت اتمیک انجام می‌گردد. لذا یک فرآیند خوش‌شانس‌تر ناحیه‌ی بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد و دیگری باید در حلقه‌ی انتظار مشغول، مشغول باشد. پس هیچگاه هر دو فرآیند به طور همزمان مسدود نمی‌شوند، در نتیجه بن‌بست رخ نخواهد داد، بنابراین شرط انتظار محدود در این شرایط برقرار است.

عدم رعایت انتظار مشغول

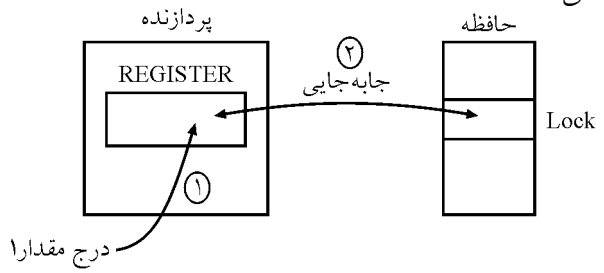
در این راه حل، مشکل انتظار مشغول وجود دارد، زیرا زمانی که مثلاً فرآیند P_0 در ناحیه‌ی بحرانی قرار دارد، فرآیند P_1 در یک حلقه‌ی انتظار زمان پردازنده را در بررسی برقراری شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی به هدر می‌دهد.

توجه: این راه حل، فقط در پردازنده‌هایی قابل اجرا است که دستورالعمل TSL را پشتیبانی می‌کنند.

توجه : پردازنده Intel، دستور TSL را پشتیبانی نمی‌کند و از دستور SWAP پشتیبانی می‌کند.

۳ـ دستورالعمل SWAP

این دستورالعمل محتوای یک رجیستر را با محتوای محلی از حافظه (متغیر Lock) به شکل اتمیک (تجزیه‌ناپذیر) جابه‌جا می‌کند.



سخت‌افزار تضمین می‌کند که عمل جابه‌جایی محتوای رجیستر و حافظه به صورت اتمیک (تجزیه‌ناپذیر) انجام شود. بدین شکل که هیچ فرآیند و حتی پردازنده دیگری نتواند به این متغیر قفل دسترسی پیدا کند تا وقتی که اجرای دستورالعمل به پایان برسد. پردازنده‌ای که دستورالعمل SWAP را اجرا می‌کند، گذرگاه حافظه را قفل می‌کند تا از دسترسی دیگر پردازنده‌ها به حافظه جلوگیری کند تا اینکه این دستورالعمل به پایان برسد.

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
enter_section ();
critical_section ();
exit_section();
remainder_section();
```

```
enter_section
MOVE REGITSTER, 1
SWAP RGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
```

```
exit_section ();
MOVE Lock, 0
RET
```

توجه مهم : در این راه حل شرط ورود به ناحیه‌ی بحرانی، اجرای زودتر دستور swap است.

در این الگوریتم، فرآیندها برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی از تابع enter_section و برای خروج از ناحیه‌ی بحرانی از تابع exit_section استفاده می‌کنند. اگر فرآیندی علاقه‌مند به ورود به

ناحیه‌ی بحرانی باشد، ابتدا تابع enter_section را صدا می‌زند تا بررسی‌های لازم جهت فراهم بودن یا نبودن ورود به ناحیه‌ی بحرانی انجام گردد. بدین نحو که ابتدا، مقدار رجیستر برابر یک مقداردهی می‌شود. سپس توسط دستور swap و به شکل اتمیک مقدار متغیر Lock و رجیستر جابه‌جا می‌شود، سپس مقدار رجیستر که حاوی مقدار متغیر Lock می‌باشد با صفر مقایسه می‌شود. اگر مقدار رجیستر برابر صفر باشد به معنی خالی بودن ناحیه‌ی بحرانی است و در ادامه دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن مقدار رجیستر با صفر، انجام نمی‌شود، زیرا مقدار رجیستر برابر صفر است و در ادامه دستور RET (RETURN) باعث می‌شود تا تابع تمام شود. بنابراین فرآیند علاقه‌مند به ورود به ناحیه بحرانی وارد ناحیه‌ی بحرانی می‌گردد. و پس از آنکه فرآیند کارش با ناحیه‌ی بحرانی تمام شد، تابع exit_section را به نشانه خروج از ناحیه‌ی بحرانی صدا می‌زند، اجرای این تابع سبب می‌شود تا مقدار متغیر Lock توسط دستور MOVE برابر صفر گردد، صفر بودن مقدار متغیر Lock به معنی خالی بودن ناحیه‌ی بحرانی است، بنابراین این امکان فراهم می‌شود تا فرآیندهای دیگر بتوانند پس از کسب اجازه توسط تابع enter_section وارد ناحیه‌ی بحرانی شوند.

در طرف مقابل اگر در حین اجرای تابع enter_section مقدار متغیر Lock برابر یک باشد، به معنی پُر بودن ناحیه‌ی بحرانی است و در ادامه شرط دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن رجیستر با مقدار صفر، برقرار است، زیرا مقدار رجیستر برابر یک است، بنابراین یک حلقه‌ی انتظار مشغول تا خالی شدن ناحیه‌ی بحرانی ایجاد می‌گردد. این حلقه‌ی انتظار مانع ورود فرآیند رقیب به ناحیه‌ی بحرانی می‌گردد.

مزایا

رعایت شرط انحصار متقابل

حالت اول (ورود غیرهمزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی باشد، یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی نباشد. پس در ابتدا مقدار متغیر Lock برابر صفر است.

فرض کنید، فرآیند P_0 با فراخوانی تابع enter_section علاقه‌مندی خود را برای ورود به ناحیه‌ی بحرانی اعلام کند، این تابع دستور swap (عمل جابه‌جایی مقدار متغیر قفل و رجیستر) را به صورت اتمیک انجام می‌دهد، بنابراین فرآیند P_0 ناحیه‌ی بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد.

توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار بود.

حال اگر پردازنده به فرآیند P_1 تعویض متن کند، این فرآیند نیز تابع enter_section را فراخوانی می‌کند و در دستور CMP مقدار رجیستر را برابر یک می‌بیند و مطابق آنچه پیش از این نیز گفتیم شرایط

برای فرآیند P_1 به گونه‌ای رقم می‌خورد که باید در حلقه‌ی انتظار مشغول، مشغول باشد. تا زمانی که فرآیند P_0 از ناحیه‌ی بحرانی خود خارج گردد و تابع exit_section را فراخوانی کند تا مقدار متغیر قفل برابر صفر گردد. (شرایط ورود به ناحیه‌ی بحرانی فراهم گردد) توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_1، برقرار نیست. (ناحیه‌ی بحرانی پُر است). بنابراین شرط انحصار متقابل برقرار است.

حالت دوم (ورود تقریباً همزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی است، یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی نباشد. پس در ابتدا مقدار متغیر Lock برابر صفر است. در این حالت هر دو فرآیند برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی تابع enter_section را به شکل همزمان فراخوانی می‌کنند. از آن جا که در دستور swap (عمل جابه‌جایی مقدار متغیر قفل و رجیستر) به صورت اتمیک انجام می‌گردد در نهایت یک فرآیند خوش‌شانس‌تر که کمی زودتر پردازنده را برای اجرای دستور swap در اختیار بگیرد، سرانجام ناحیه‌ی بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد و دیگری باید در حلقه‌ی انتظار مشغول، مشغول باشد. بنابراین شرط انحصار متقابل در حالت دوم نیز برقرار است.

رعایت شرط پیشرفت

شرط پیشرفت می‌گفت، فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، در تصمیم‌گیری برای ورود فرآیندهای دیگر به ناحیه‌ی بحرانی نباید شرکت کند. در این راه حل یک فرآیند پس از خروج از ناحیه‌ی بحرانی و قبل از ورود به ناحیه‌ی باقی‌مانده، توسط فراخوانی تابع exit_section (مقداردهی متغیر قفل به صفر) ناحیه‌ی بحرانی را خالی اعلام می‌کند، در واقع مانعی که سد راه ورود فرآیندهای دیگر به ناحیه‌ی بحرانی بود، از روی ناحیه‌ی بحرانی برمی دارد و در ادامه برای رسیدگی به کارهای دیگرش، در ناحیه‌ی باقی‌مانده قرار می‌گیرد، بنابراین فرآیندهای رقیب می‌توانند پس از کسب اجازه‌ی ورود به ناحیه‌ی بحرانی توسط تابع enter_section و مستقل از فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، وارد ناحیه‌ی بحرانی شوند. بنابراین شرط پیشرفت برقرار است.

معایب

عدم رعایت شرط انتظار محدود

گرسنگی دارد: گرسنگی یعنی اینکه یک فرآیند مدام کار کند و از کار کردن فرآیندهای دیگر جلوگیری کند. در اینجا گرسنگی بدین معنی که یک فرآیند مدام داخل ناحیه‌ی بحرانی برود و از ورود فرآیندهای دیگر به ناحیه بحران جلوگیری کند، وجود دارد، از آن‌جا که این الگوریتم فاقد مکانیزم نوبت‌دهی است، بنابراین ورود به ناحیه‌ی بحرانی تصادفی خواهد بود، پس یک فرآیند خوش‌شانس‌تر و با بخت

و اقبال بالاتر (منظور از شانس، چگونگی تعویض متن پردازنده است) می‌تواند، بارها و بارها پردازنده را بدست آورد و وارد ناحیه‌ی بحرانی گردد، بدون آنکه فرآیند رقیب بتواند کاری از پیش ببرد، این یعنی گرسنگی زیرا سرنوشت ورود فرآیندها به بخت و اقبال فرآیندها گره خورده است، و هیچ گونه نوبتی رعایت نمی‌شود، بنابراین شرط انتظار محدود به دلیل گرسنگی برقرار نیست.

بن‌بست ندارد:

زیرا پس از اجرای موازی یا تقریباً همزمان، هر دو فرآیند برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی تابع enter_section را به شکل همزمان فراخوانی می‌کنند. از آن جا که در دستور swap (عمل جابه‌جایی مقدار متغیر قفل و رجیستر) به صورت اتمیک انجام می‌گردد. لذا یک فرآیند خوش‌شانس‌تر ناحیه‌ی بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد و دیگری باید در حلقه‌ی انتظار مشغول، مشغول باشد. پس هیچ گاه هر دو فرآیند به طور همزمان مسدود نمی‌شوند، در نتیجه بن‌بست رخ نخواهد داد، بنابراین شرط انتظار محدود در این شرایط برقرار است.

عدم رعایت انتظار مشغول

در این راه حل، مشکل انتظار مشغول وجود دارد، زیرا زمانی که مثلاً فرآیند P_0 در ناحیه‌ی بحرانی قرار دارد، فرآیند P_1 در یک حلقه‌ی انتظار زمان پردازنده را در بررسی برقراری شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی به هدر می‌دهد.

این راه حل، فقط در پردازنده‌هایی قابل اجراست که دستور swap را پشتیبانی می‌کنند.

توجه : پردازنده Intel، از دستور swap، پشتیبانی می‌کند.

راه حل‌های سیستم عامل (سمافور ـ راهنما): Semaphore

از راه‌حل‌های نرم‌افزاری گفته شده الگوریتم پیترسون می‌تواند به عنوان راه حل صحیح به کار رود. همچنین در راه حل‌های سخت‌افزاری گفته شده استفاده از دستورالعمل‌های TSL و swap با صرف نظر کردن از مشکل گرسنگی‌شان، می‌توانند به عنوان یک راه حل استفاده شوند. اما مشکلی که وجود دارد، این است که این نوع راه حل‌ها دو مشکل اساسی زیر را دارند:

۱ ـ انتظار مشغول

همان طور که اشاره شد، راه حل‌های نرم‌افزاری و سخت‌افزاری، یک مشکل اساسی به نام انتظار مشغول دارند. در این دسته راه حل‌ها، زمانی که فرآیندی در ناحیه‌ی بحرانی به سر می‌برد. هر فرآیند دیگری که برای ورود به ناحیه‌ی بحرانی تلاش کند در هر بار تعویض متن پردازنده باید به طور پیوسته داخل یک حلقه، زمان را سپری کند. این حلقه‌های بیهوده یک مشکل محسوب می‌شود. چرا که در زمانی که انتظار مشغول وقت پردازنده را تلف می‌کند، ممکن است فرآیندهای دیگری وجود داشته

باشند که بتوانند از پردازنده استفاده مفید کنند.

۲ ـ اولویت معکوس

فرض کنید دو فرآیند، یکی با اولویت بالا و دیگری با اولویت پایین وجود دارد. اگر فرآیند با اولویت پایین در ناحیه‌ی بحرانی باشد و سپس فرآیند با اولویت بالا وارد حافظه‌ی اصلی شود و درخواست ناحیه‌ی بحرانی را داشته باشد، چون اولویت بالاتری دارد، زمان‌بند، پردازنده را در اختیار آن قرار می‌دهد و منتظر منبع بحرانی می‌ماند. در این حین، فرآیند با اولویت پایین پردازنده را در اختیار ندارد تا کار خود را به پایان رساند و ناحیه‌ی بحرانی را آزاد کند. در نتیجه‌ی این سناریو، فرآیند با اولویت بالا تابی نهایت بار در حلقه‌ی انتظار مشغول، دور می‌زند. لازم به ذکر است که به این وضعیت مشکل اولویت معکوس گفته می‌شود، که همان بن‌بست است.

چون فرآیند با اولویت پایین، منتظر فرآیند با اولویت بالا است تا پردازنده را رها کند و فرآیند با اولویت بالا، منتظر فرآیند با اولویت پایین است تا ناحیه‌ی بحرانی را آزاد کند شرط لازم (انحصار متقابل) برقرار است، شروط بدیهی (انحصاری بودن و نگهداری و انتظار) هم وجود دارد، شرط کافی، (سیکل انتظار چرخشی) هم برقرار است، پس وقوع بن‌بست حتمی است.)

با به کارگیری سمافور به عنوان راه حلی جهت کنترل شرایط رقابتی می‌توان از بروز چنین مشکلاتی جلوگیری کرد. راه حل سمافور در سال ۱۹۶۵ توسط Dijkstra پیشنهاد شده است.

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();
```

تذکر: mutex از عبارت Mutual Exclusion گرفته شده است.

توجه : دو تابع wait (mutex) و signal (mutex)، باید به صورت اتمیک (تجزیه‌ناپذیر) انجام گیرند. اتمیک بودن، تضمین می‌کند که از لحظه‌ای که یک عملیات بر روی شمارنده سمافور شروع می‌شود، هیچ فرآیند دیگری نتواند به سمافور دسترسی پیدا کند تا زمانی که آن عملیات به پایان برسد. اتمیک بودن این عملیات برای حل مسایل همگام‌سازی و کنترل شرایط رقابتی و به تبع برقراری شرط انحصار متقابل کاملاً لازم و ضروری است.

توجه : در مقاله Dijkstra، از نام‌های P و V (حرف اول کلمات آلمانی تست "probern" و افزایش "Verhogen") به ترتیب به جای wait و signal استفاده شده بود و همچنین در سایر متون، از نام‌های down و up به ترتیب برای این دو استفاده می‌کنند. در همان متون گاهاً به جای نام‌های down و up، به

ترتیب از عبارت‌های mutex_lock و mutex_unlock نیز استفاده شده است.
راه حل سمافور بر دو دسته کلی (۱) سمافور عمومی و (۲) سمافور دو دویی می‌باشد، که در ادامه به آن‌ها می‌پردازیم:

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.
ساختار کلی سمافور عمومی به صورت زیر است:

```
struct semaphore
{
int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

برای برقراری شرط انحصار متقابل از این شمارنده با مقدار اولیه یک استفاده می‌گردد.

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله انتظار مشغول و حل مسأله اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگه‌داری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.
بر روی سمافور عمومی s دو تابع wait (s) و signal (s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.
تابع wait (s): عملیات آن ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.
ساختار این تابع به صورت زیر است:

```
wait (semophore s)
   {
s.count = s.count - 1;
```

```
    if (s.count < 0)
    {
    add this process to s.queue;
    block ();
    }
}
```

توجه : راه حل سمافور و تابع اتمیک wait باید توسط سیستم عامل پشتیبانی گردد، در غیر این صورت می‌توان این راه حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، ابتدا یک واحد از شمارنده سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if (s.count < 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر این صورت، فرآیند، وارد ناحیه‌ی بحرانی می‌گردد.

توجه : در راه حل‌های قبل از سمافور، در صورتی که یک فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، با ناحیه‌ی بحرانی پُر مواجه می‌شد، پس از پایان هر برش زمانی از وضعیت اجرا به وضعیت آماده منتقل می‌گردید و مجدداً توسط زمان‌بند کوتاه مدت، همان فرآیند این شانس را داشت که مجدداً پردازنده را در اختیار بگیرد و مجدداً شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی را بررسی کند، در حالیکه همچنان ناحیه‌ی بحرانی پُر است که از این مسأله به انتظار مشغول یاد کردیم، همانطور که قبلاً هم گفتیم انتظار مشغول سبب هدر دادن وقت پردازنده می‌گردد، اما در راه حل سمافور در صورتی که یک فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، با ناحیه‌ی بحرانی پُر مواجه شود، در صف سمافور قرار داده می‌شود و حالت فرآیند، از وضعیت اجرا به وضعیت منتظر، تغییر می‌کند. در این شرایط فرآیندی که با ناحیه‌ی بحرانی پُر مواجه شده است، دیگر در صف آماده قرار ندارد و به تبع دیگر این شانس را هم نخواهد داشت تا توسط زمان‌بند کوتاه مدت مجدداً انتخاب گردد تا پردازنده را در اختیار بگیرد و مجدداً شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی را بررسی کند و سبب هدر دادن زمان پردازنده گردد. پس علاوه بر برقراری شروط انحصار متقابل، پیش‌روی و انتظار محدود، مسأله انتظار مشغول و اولویت معکوس نیز توسط راه سمافور حل شد.

توجه : در سمافور عمومی، وقتی شمارنده سمافور، مقدار منفی دارد، قدرمطلق این مقدار، معرف تعداد فرآیندهای بلوکه شده در صف سمافور است.

تابع signal (s) : عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
signal (semaphore s)
{
s.count = s.count + 1
   if (s.count < = 0)
   {
   remove a process from queue;
   wake up ();
   }
}
```

توجه : راه حل سمافور و تابع اتمیک signal باید توسط سیستم عامل پشتیبانی گردد، در غیر این صورت می‌توان این راه حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع signal (s) توسط یک فرآیند علاقه‌مند به خروج از ناحیه‌ی بحرانی، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count < = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به ناحیه‌ی بحرانی که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up () بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمان‌بند کوتاه مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند که از ناحیه‌ی بحرانی خارج شود، با اجرای تابع signal (s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

مزایا

رعایت شرط انحصار متقابل

حالت اول (ورود غیر همزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی است. یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی قرار ندارد. پس در ابتدا مقدار شمارنده سمافور برابر یک است.

فرض کنید، فرآیند P_0 با فراخوانی تابع wait (s) علاقه‌مندی خود را برای ورود به ناحیه‌ی بحرانی

اعلام کند، سیستم عامل اتمیک (تجزیه‌ناپذیر) بودن اجرای تابع wait (s) را تضمین می‌کند، هنگام اجرای تابع اتمیک wait (s) و پس از اجرای s.count = s.count - 1، مقدار شمارنده سمافور، برابر صفر می‌گردد.

سپس دستور if (s.count < 0) بررسی می‌گردد و چون شرط برقرار نیست تابع wait (s); خاتمه می‌یابد، بنابراین فرآیند P_0 ناحیه‌ی بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد. توجه کنید که شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی برای فرآیند P_0 برقرار بود. حال اگر پردازنده به فرآیند P_1 تعویض متن کند، این فرآیند نیز تابع wait (s) را فراخوانی می‌کند و پس از اجرای دستور s.count = s.count -1 مقدار شمارنده سمافور برابر ۱- می‌گردد. سپس دستور if (s.count < 0) بررسی می‌گردد و چون شرط برقرار است فرآیند P_1 شمارنده سمافور به صف سمافور می‌رود. حال اگر پردازنده به فرآیندهای بعدی علاقه‌مند ورود به ناحیه‌ی بحرانی نیز تعویض متن کند، تا زمانی که فرآیند P_0 در ناحیه‌ی بحرانی باشد، وضع به همین منوال خواهد بود.

حالت دوم (ورود تقریباً همزمان فرآیندها)

در شرایطی که ناحیه‌ی بحرانی خالی است، یعنی هیچ فرآیندی داخل ناحیه‌ی بحرانی قرار ندارد. پس در ابتدا مقدار شمارنده سمافور برابر یک است.

در این حالت فرآیندها برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی تابع wait (s) را به شکل همزمان فراخوانی می‌کنند. از آنجا که تابع wait (s) به صورت اتمیک انجام می‌گردد. یک فرآیند خوش‌شانس‌تر که زودتر پردازنده را بدست می‌آورد تابع wait (s) را زودتر فراخوانی کند، ناحیه‌ی بحرانی را تصاحب می‌کند و در ادامه وارد ناحیه‌ی بحرانی می‌گردد و فرآیندهای بعدی تا زمانی که فرآیند اول داخل ناحیه‌ی بحرانی قرار دارد، به ترتیب در صف سمافور قرار می‌گیرند. بنابراین شرط انحصار متقابل برقرار است.

توجه :اگر مقدار اولیه شمارنده سمافور برابر n می‌بود، همه n فرآیند قادر خواهند بود که همزمان وارد ناحیه‌ی بحرانی شوند و هیچ مشکلی حل نشده است! اگر مقدار اولیه شمارنده سمافور، برابر صفر می‌بود، هر فرآیندی که بخواهد وارد ناحیه‌ی بحرانی شود، بلوکه می‌شود، یعنی پس از مدتی همه فرایندها بلوکه می‌شوند و هیچگاه بیدار نخواهند شد زیرا دیگر کسی نیست که بخواهد آن‌ها را بیدار کند. اما اگر مطابق آنچه بیش از این نیز گفتیم، مقدار شمارنده سمافور را برابر یک در نظر بگیریم، تنها یک فرآیند می‌تواند وارد ناحیه‌ی بحرانی شود و پس از ورود فرآیند اول به ناحیه‌ی بحرانی، مقدار شمارنده سمافور صفر خواهد شد، بنابراین اگر در هنگامی که فرآیند اول در ناحیه‌ی بحرانی به سر می‌برد، یک یا چند فرآیند دیگر قصد ورود به ناحیه‌ی بحرانی را داشته باشند به ترتیب در صف سمافور خواهند خوابید. پس شرط انحصار متقابل برقرار است. در آخر اینکه هر فرآیند که از ناحیه‌ی

بحرانی خارج شود با اجرای تابع signal (s) فرآیند سر صف سمافور را بیدار می‌کند.

رعایت شرط پیشرفت

هرگاه فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، در تصمیم‌گیری برای ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شرکت کند، آن راه حل شرط پیشروی را رعایت نمی‌کند. در این راه حل یک فرآیند پس از خروج از ناحیه‌ی بحرانی و قبل از ورود به ناحیه‌ی باقی‌مانده، توسط فراخوانی تابع signal ابتدا ناحیه‌ی بحرانی را خالی اعلام می‌کند، سپس برای رسیدگی به ادامه کارهایش، در ناحیه‌ی باقی‌مانده قرار می‌گیرد. خالی اعلام کردن ناحیه‌ی بحرانی توسط یک فرآیند پس از خروج از ناحیه‌ی بحرانی و قبل از ورود به ناحیه‌ی باقی‌مانده باعث می‌شود تا این فرآیند زمانی که در ناحیه‌ی باقی‌مانده قرار دارد، در تصمیم‌گیری برای ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شرکت نکند و این یعنی برقراری شرط پیشرفت. فراخوانی تابع signal توسط یک فرآیند خارج شده از ناحیه‌ی بحرانی باعث می‌شود تا فرآیند ابتدای صف سمافور از خواب بیدار شود و از وضعیت منتظر به وضعیت آماده تغییر حالت دهد و در صف آماده پردازنده قرار گیرد. در واقع یک فرآیند پس از خروج از ناحیه‌ی بحرانی توسط فراخوانی تابع signal راه را برای ورود فرآیند ابتدای صف سمافور به صف آماده پردازنده فراهم می‌کند تا در صورتی که زمان‌بند کوتاه مدت، پردازنده را در اختیارش قرار داد، بتواند ناحیه‌ی بحرانی را تصاحب کند.

این روند یکی پس از دیگری و به صورت خروج به ترتیب ورود برای صف سمافور رخ می‌دهد، هر فرآیندی که از ناحیه‌ی بحرانی خارج شود، فرایند ابتدای صف سمافور را بیدار می‌کند و این روال ادامه پیدا می‌کند، بنابراین شرط پیشرفت برقرار است.

رعایت شرط انتظار محدود

گرسنگی ندارد: گرسنگی یعنی اینکه یک فرآیند مدام کار کند و از کار کردن فرآیندهای دیگر جلوگیری کند. در اینجا گرسنگی بدین معنی که یک فرآیند مدام داخل ناحیه‌ی بحرانی برود و از ورود فرآیندهای دیگر به ناحیه‌ی بحرانی جلوگیری کند، وجود ندارد. از آن جا که فرآیندهای بلوکه شده در صفی از فرآیندها که در آن ترتیب خروج به ترتیب ورود (FIFO) است، قرار می‌گیرند، پس هیچگاه گرسنگی بروز نمی‌کند.

بن‌بست ندارد: استفاده از سمافور مطابق آنچه در حالت دوم برقراری شرط انحصار متقابل گفتیم، ایجاد بن‌بست نمی‌کند. زیرا ورود به ناحیه‌ی بحرانی نوبتی انجام می‌شود.

توجه : استفاده نادرست از سمافور ممکن است، منجر به بن‌بست گردد.

مثال: دو فرآیند P_0 و P_1 را در نظر بگیرید که هر کدام به دو سمافور s و q با مقادیر اولیه یک برای

شمارنده سمافور دسترسی دارند، کد فرآیندها به صورت زیر است:

P_0	P_1
① wait (s);	② wait (q);
③ wait (q);	④ wait (s);
⋮ ⋮	
⑤ signal (s);	⑥ signal (q);
⑦ signal (q);	⑧ signal (s);

فرض کنید دستورات ①و ②با هم (در سیستم چند پردازنده‌ای و اجرای موازی) و یا یک در میان (در سیستم تک پردازنده‌ای و اجرای همروند) اجرا شوند. حال اجرای دستور ③باید منتظر اجرای دستور ⑥باشد و اجرای دستور ④ منتظر اجرای دستور ⑤باشد. این بدان معناست که بن‌بست رخ داده است.

توجه :در برخی کتب غیر مرجع، راه حل سمافور را دارای مشکل بن‌بست می‌دانند، حتی به غلط نتیجه گرفته‌اند که در راه حل سمافور شرط انتظار محدود به دلیل بن‌بست رعایت نمی‌شود. اما دیدید که کنترل شرایط رقابتی برای برقراری شروط انحصار متقابل، پیشروی و انتظار محدود به سادگی با یک سمافور بدون هیچ نقصی امکان‌پذیر است.

در واقع باید بگوییم که تنها مشکل سمافور در سیستمی که حافظه مشترک دارد، پیچیدگی استفاده از آن در حل مسایل است. اگر برنامه نویس دقت نکند، راه حل سمافور ممکن است مانند مثال قبل، دچار بن‌بست شود. در این‌صورت، نباید بگوییم سمافور مشکل بن بست دارد! یعنی ترجیح می‌دهیم این مشکل را به هوش برنامه‌نویس نسبت دهیم. این مشکل بر این نکته اشاره دارد که باید در هنگام استفاده از سمافورها دقیق باشید. یک خطای کوچک و ظریف ممکن است، همه چیز را متوقف کند.

سمافور دودویی

کلیه تعاریف، توضیحات و خصوصیاتی که برای سمافور عمومی گفته شد، برای سمافور دودویی نیز برقرار است. تنها تفاوت سمافور دودویی و سمافور عمومی در نحوه‌ی تعریف توابع wait و singal است.

تابع wati(s) :عملیات آن به ترتیب شامل، تست کردن مقدار شمارنده، کاهش مقدار شمارنده (اما در نهایت فقط تا مقدار صفر) و احیاناً خواباندن یک فرآیند.

ساختار این تابع به صورت زیر است:

```
wait (semaphore s)
```

```
{
    if (s.count == 1)
    s.count = 0
    else
    {
    add this process to s.queue;
    block ();
    }
}
```

توجه :به تفاوت ترتیب عملیات سمافور عمومی و دودویی در تابع wait دقت کنید.

تابع signal (s):عملیات آن به ترتیب شامل تست خالی بودن صف، افزایش مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

```
signal (semaphore s)
  {
    if (s.queue is empty)
      s.count = 1;
      else
      {
      remove a process from queue;
      wake up ();
      }
}
```

توجه :به تفاوت ترتیب عملیات سمافور عمومی و دودویی در تابع signal دقت کنید.

توجه : در سمافور دودویی، شمارنده سمافور، هیچگاه منفی نمی‌شود و در هر شرایطی فقط می‌تواند دو مقدار باینری یا دودویی ۰ یا ۱ را داشته باشد، برای اثبات، مجدداً به تعاریف توابع wait و signal در سمافور دودویی دقت کنید.

توجه : هر فرآیندی که از ناحیه‌ی بحرانی خارج شود با اجرای تابع signal (s)، فرآیند ابتدای صف را بیدار می‌کند (تغییری در شمارنده سمافور ایجاد نشده و برابر مقدار صفر باقی می‌ماند) و اگر هیچ فرآیند منتظری درون صف سمافور وجود نداشته باشد، مقدار شمارنده سمافور به یک مقداردهی می‌شود.

توجه مهم : در حل مسائل، سمافور عمومی و دودویی، یک پاسخ یکسان را تولید خواهند کرد، اما شیوه‌های رسیدن به پاسخ با توجه به متفاوت بودن توابع wait و signal، کمی تفاوت دارد، ما در این کتاب برای حل کلیه مسائل سمافور، از سمافور عمومی استفاده می‌کنیم، به شما نیز توصیه می‌کنیم، همین راه حل را در پیش بگیرید.

همگام سازی فرآیندها توسط سمافور

سمافور دو کاربرد دارد (۱) کنترل شرایط رقابتی (۲) همگام سازی فرآیندها
کاربرد سمافور در کنترل شرایط مسابقه بررسی شد.
حال با یک مثال به بیان کاربرد سمافور در همگام‌سازی فرآیندها می‌پردازیم.

مثال: دو فرآیند P_1 و P_2 را در نظر بگیرید که می‌خواهند به طور همروند اجرا گردند. P_1 دستور S_1 و P_2 دستور S_2 را اجرا می‌کند. فرض کنید دستور S_2 فقط باید بعد از دستور S_1 اجرا شود. برای این کار یک شمارنده سمافور مشترک به نام synch و با مقدار صفر را برای P_1 و P_2 در نظر گرفته و کدهای زیر را برای P_1 و P_2 می‌نویسیم.

P_1	P_2
S_1;	wait (synch);
signal (synch);	S_2;

در این پیاده‌سازی، پُر واضح است که تا زمانی که S_1 از فرآیند اول اجرا نشود، فرآیند P_2 اجازه اجرای خط S_2 را ندارد. چون مقدار اولیه synch صفر است، P_2 فقط هنگامی S_2 را اجرا می‌کند که P_1 قبلاً signal(synch) را صدا زده باشد و این کار نیز فقط بعد از دستور S_1 انجام می‌پذیرد.

نکته مهم: برای همگام سازی، معمولاً مقدار اولیه شمارنده سمافور برابر صفر و برای برقراری شرط انحصار متقابل مطابق آنچه پیش از این نیز گفتیم همیشه برابر یک است.
در ادامه حل مسائل کلاسیک با استفاده از راه حل سمافور مورد بررسی قرار می‌گیرد.

حل مسأله تولیدکننده و مصرف‌کننده توسط سمافور (با بافر محدود)

دو فرآیند که یکی تولیدکننده و دیگری مصرف‌کننده است، یک بافر با اندازه‌ی محدود را به اشتراک می‌گذارند. فرآیند تولیدکننده اطلاعات را در بافر قرار می‌دهد و فرآیند مصرف‌کننده اطلاعات را از بافر برمی‌دارد. مشکل زمانی به وجود می‌آید که تولیدکننده می‌خواهد اطلاعاتی را در بافر قرار دهد اما با بافر پُر مواجه می‌شود که در این حالت تولیدکننده باید بخوابد تا مصرف‌کننده توسط عمل برداشت اطلاعات، مقداری از بافر را خالی کند، تا شرایط بیدار شدن تولیدکننده فراهم گردد. به همین ترتیب اگر مصرف‌کننده بخواهد داده‌ای را از بافر بردارد ولی با بافر خالی مواجه شود، باید بخوابد تا تولیدکننده

توسط عمل درج اطلاعات، مقداری از بافر را پُر کند، تا شرایط بیدار شدن مصرف‌کننده فراهم گردد. در این راه حل از سه شمارنده سمافور استفاده می‌شود، اولی که full نامیده شده است برای شمردن تعداد خانه‌های پُر بافر و دومی که empty نامیده شده است برای شمارش خانه‌های خالی بافر و سومی هم که mutex نامگذاری شده است برای برقراری شرط انحصار متقابل به کار می‌رود، mutex به این منظور استفاده می‌شود که مطمئن شویم تولیدکننده و مصرف‌کننده به طور همزمان به بافر دسترسی ندارند. در ابتدا مقدار اولیه full=۰ و empty = N یعنی برابر تعداد کل خانه‌های بافر و mutex=۱ می‌باشد. شمارنده‌های سمافوری که مقدار اولیه آنها برابر یک است اغلب برای این استفاده می‌شوند که از ورود همزمان دو یا چند فرآیند به ناحیه‌ی بحرانی جلوگیری کنند. (برقراری شرط انحصار متقابل) اگر هر فرآیند، تابع wait را قبل از ورود به ناحیه‌ی بحرانی و تابع signal را دقیقاً پس از خروج از ناحیه‌ی بحرانی اجرا کند، شرط انحصار متقابل برقرار خواهد شد.

ساختار کلی این راه حل برای حل مسأله تولیدکننده و مصرف‌کننده به صورت زیر می‌باشد:

```
تولیدکننده (producde)
while (TRUE)
{
تولید قطعه ;
wait (empty);
wait (mutex);
درج قطعه تولید شده در بافر (ناحیه‌ی بحرانی)
signal (mutex);
signal (full);
}
```

```
مصرف‌کننده (consumer)
while (TRUE)
{
wait (full);
wait (mutex);
حذف قطعه از بافر (ناحیه‌ی بحرانی)
signal (mutex);
signal (empty);
مصرف قطعه حذف شده ;
}
```

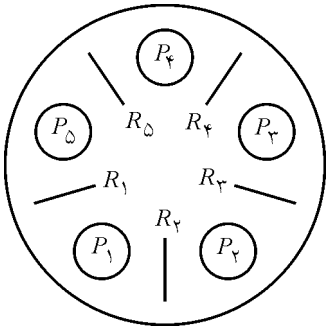
در قطعه کدهای فوق شمارنده‌های سمافور به دو روش و دو کاربرد کاملاً متفاوت استفاده شده‌اند، شمارنده سمافور mutex جهت برقراری شرط انحصار متقابل استفاده شده است تا تضمین کند در هر لحظه فقط یک فرآیند از بافر استفاده می‌کند (یا تولیدکننده یا مصرف‌کننده). ولی سمافورهای full و empty جهت همگام‌سازی فرآیندهای تولید کننده و مصرف کننده استفاده شده‌اند تا تضمین کنند، اتفاقات با ترتیب مشخص انجام می‌پذیرد. در این حالت، مطمئن می‌شویم که تولیدکننده در هنگامی که بافر پُر است، متوقف می‌شود و مصرف‌کننده نیز در هنگامی که بافر خالی است، متوقف می‌شود. این کاربرد کاملاً با انحصار متقابل متفاوت است.

حل مسأله تولید کننده و مصرف کننده توسط سمافور (با بافر نامحدود)

جهت حل مسأله تولیدکننده و مصرف‌کننده با بافر نامحدود توسط سمافور، کافی است شمارنده سمافور empty و به تبع توابع wait (empty) و signal (empty) را از راه حل فوق حذف کنید. در این شرایط تولید کننده، بدون مانع wait (empty)، تا هر چه قدر که می‌خواهد، می‌تواند اقدام به تولید قطعه نماید!

حل مسأله فیلسوفان خورنده توسط سمافور

۵ فیلسوف زندگی خود را صرف فکر کردن و خوردن کرده‌اند. آنها دور یک میز دایره‌ای با ۵ بشقاب و ۵ عدد چنگال نشسته‌اند. هر فیلسوف برای غذا خوردن حتماً باید دو چنگال در دست داشته باشد. بین هر جفت بشقاب روی میز، یک چنگال وجود دارد. هنگامی که فیلسوفی در حال فکر کردن است با بقیه هیچ ارتباطی ندارد. هر از گاهی فیلسوف احساس گرسنگی کرده و سعی می‌کند، چنگال‌های سمت راست و چپش را یکی یکی و با هر ترتیب ممکن بردارد، اگر موفق به برداشتن هر دو چنگال شود برای مدتی غذا خورده و دوباره چنگال‌ها را پایین گذاشته و به فکر کردن ادامه می‌دهد. فیلسوف مجاز است که در هر بار فقط یک چنگال را بردارد و همچنین نمی‌تواند چنگالی که دست فیلسوف دیگری است را به زور بگیرد.



راه حل اول

از آنجایی که هر فیلسوفی که در حال خوردن است، نمی‌تواند چنگال‌های در اختیارش را به فیلسوف مجاور بدهد، لازم است برای هر چنگال شرط انحصار متقابل برقرار باشد. لذا برای هر چنگال یک شمارنده سمافور تعریف می‌شود. مسأله فیلسوفان خورنده در راه حل اول بر دو نوع (۱) چپگرد و (۲) راستگرد می‌باشد.

راه حل چپگرد: فیلسوفان ابتدا چنگال سمت چپ را بر می‌دارند.

ساختار کلی این راه حل به صورت زیر می‌باشد:

تذکر: قطعه کد زیر را شبه کد فرض کنید.

```
# define N 5
semaphor fork[5] ={1};
void philosopher (int i)
{
   while (TRUE)
      {
         wait (fork [i]);
         wait (fork [(i+ 1) % N]);
         eat ();
         signal (fork [i]);
         signal (fork [(i+ 1) % N]);
         think ();
         }
}
```

برای هر چنگال (R_i) ، یک شمارنده سمافور با مقدار اولیه یک تعریف شده است، مطابق الگوریتم فوق هر فیلسوفی که می‌خواهد تغذیه کند، باید بتواند ابتدا چنگال چپ خود را بردارد، این کار با دستور wait (fork [i]) انجام می‌گیرد. سپس با اجرای دستور wait (fork [(i+ 1) % N]) باید بتواند چنگال راست خود را بردارد. اگر فیلسوفی موفق به انجام این دو عمل شد، می‌تواند خوردن را شروع کند.

اگر چه این راه حل تضمین می‌کند که هیچ دو همسایه‌ای همزمان غذا نخورند، ولی این روش ممکن است دچار بن‌بست شود.

فرض کنید که هر پنج فیلسوف همزمان گرسنه شده و هر کدام چنگال سمت چپ خود را بردارند (همه‌ی ۵ فیلسوف خط اول یعنی wait (fork [i]);) را اجرا کنند). بنابراین تمام عناصر آرایه fork[5] برابر با صفر می‌شوند. هنگامی که فیلسوفان سعی می‌کنند تا چنگال سمت راست خود را بردارند wait(fork[(i+ 1)%5);، یک بن‌بست به وجود می‌آید.

شرایط بن‌بست را به یادآورید:

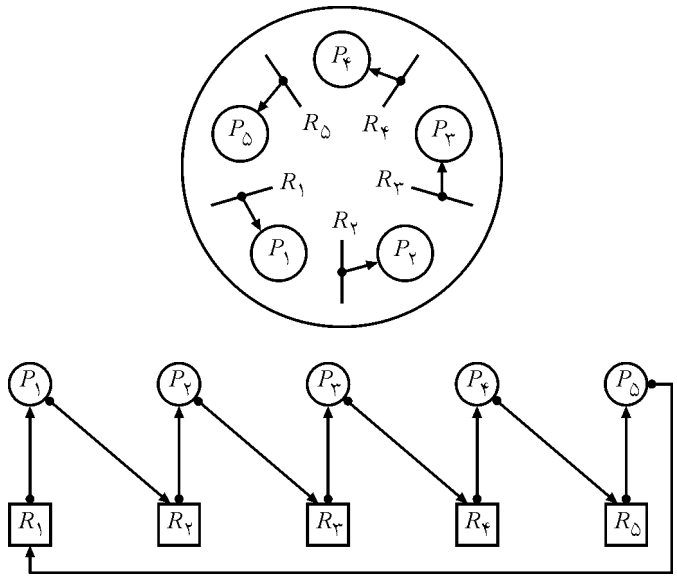
۱ـ انحصار متقابل (برقرار است، توسط تعریف شمارنده سمافور برای هر چنگال (منبع بحرانی))

۲ـ انحصاری بودن (برقرار است، نمی‌توان چنگال (منبع بحرانی) را به زور پس گرفت)

۳ـ نگهداری و انتظار (برقرار است)

۴ـ سیکل انتظار چرخشی (برقرار است)

به شکل بازسازی شده‌ی فیلسوفان خورنده در شرایط بن‌بست توجه کنید:



راه حل راستگرد: فیلسوفان ابتدا چنگال سمت راست را برمی‌دارند.

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
# define N 5
semaphore fork[5] = {1};
void philsopher (int i)
   {
      while (TRUE)
         {
         wait (fork [(i+ 1)%N]);
         wait (fork [i]);
         eat();
         signal (fork[(i+ 1)%N]);
         signal (fork[i]);
         think();
         }
}
```

توجه :کلیه تعارف، توضیحات و خصوصیاتی که برای راه حل چپگرد گفته شد، به شکل بالعکس برای راه حل راستگرد نیز برقرار است.

راه حل دوم: در این راه حل هر فیلسوف تنها زمانی می‌تواند تغذیه کند که هر دو چنگال موجود باشند. در واقع یک راه حل بهبود یافته نسبت به راه حل اول که منجر به بن‌بست نشود. این است که پنج با دستورالعمل قبل از ;()think را با یک شمارنده سمافور mutex با مقدار اولیه یک حفاظت کنیم.

ساختار کلی این راه حل به صورت زیر می‌باشد:

تذکر: قطعه کد زیر را شبه کد فرض کنید.

```
# define N 5
semaphore fork [5] = {1}
semaphore mutex =1;
void philsopher (int i)
{
while (TRUE)
   {
      wait (mutex);
      wait (fork [i]);
      wait (fork [(i+ 1) % N]);
      eat ();
      signal (fork [i]);
      signal (fork [(i+ 1) % N ])
      signal (mutex);
      think ();
   }
}
```

هر فیلسوف قبل از شروع به برداشتن چنگال‌ها تابع wait را بر روی یک شمارنده سمافور mutex انجام می‌دهد و بعد از پایین گذاشتن چنگال‌ها تابع signal را بر روی شمارنده سمافور mutex انجام می‌دهد. از نقطه نظر تئوری این راه حل مناسب است، اما کارایی را کاهش می‌دهد، زیرا در هر لحظه فقط یک فیلسوف می‌تواند مشغول غذا خوردن باشد. اما با ۵ چنگال موجود باید بتوانیم به دو فیلسوف اجازه غذا خوردن همزمان را بدهیم.

راه حل مانیتور

اگرچه راه حل سمافور، راه کار درستی برای کنترل شرایط رقابتی و همگام‌سازی می‌باشد ولی استفاده از آن دقت بسیار زیادی را می‌خواهد. مثلاً اگر برنامه‌نویس در نحوه‌ی استفاده و یا چیدمان توابع wait و signal حول ناحیه‌ی بحرانی برای برقراری شرط انحصار متقابل دقت نکند ممکن است شرط انحصار متقابل نقض گردد و یا پدیده بن‌بست آشکار گردد. همچنین اگر برنامه‌نویس در چیدمان این توابع برای همگام‌سازی فرآیندها دقت نکند، ممکن است باز هم پدیده‌ی بن‌بست ایجاد گردد و یا روالی غیر از آنچه مد نظر بوده است انجام گیرد.

برای اینکه نوشتن برنامه‌های درست، ساده‌تر شود، یک ابزار راحت‌تر برای کنترل شرایط رقابتی و همگام‌سازی به نام مانیتور ابداع شد. مانیتور یک راه حل مبتنی بر کامپایلر زبان برنامه‌نویسی است. توابع wait و signal در راه حل سمافور اغلب توسط سیستم عامل پشتیبانی می‌شود، حتی می‌توان این توابع را توسط زبان‌های برنامه‌نویسی شبیه سازی کرد و در کتابخانه توابع قرار داد تا هرگاه نیاز بود، مورد استفاده قرار گیرد. اما مانیتور ماهیتی دارد که باید توسط کامپایلر زبان برنامه‌نویسی پشتیبانی گردد.

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
monitor mon_name
Begin
i: integer;              variable declarations
c: contdition;           condition variable declarations
function f1()
begin
critical_section1 ();
end;
functionf2()
begin
critical_section2 ();
end;
function fn()
begin
critical_sectionn ();
end;
initialization code;
end monitor
```

کاربردهای مانیتور

مانیتور در دو کاربرد متفاوت می‌تواند مورد استفاده قرار گیرد:

۱ـ کنترل شرایط رقابتی (برقراری شرط انحصار متقابل، پیشرفت و انتظار محدود)

۲ـ همگام‌سازی (کنترل شرایط رقابتی + همگام سازی)

در ادامه به تشریح این دو مورد می‌پردازیم:

کنترل شرایط رقابتی با استفاده از مانیتور (برقراری شرط انحصار متقابل)

مانیتور برای کنترل شرایط رقابتی شامل مجموعه‌ای از متغیرهای داده‌ای و توابع می‌باشد که در یک ماژول بسته‌بندی شده است. متغیرهای داده‌ای تعریف شده در داخل مانیتور، فقط توسط توابع داخل همان مانیتور قابل دسترسی هستند. بنابراین فرآیندها برای دسترسی به متغیرهای داده‌ای داخل یک مانیتور، می‌بایست توابع داخل همان مانیتور را فراخوانی کنند.

مهمترین قانون مانیتور برای کنترل شرایط رقابتی، این است که هیچ دو فرآیندی نمی‌توانند به طور همزمان وارد مانیتور شوند، به بیان دیگر در هر لحظه فقط یک فرآیند می‌تواند داخل یک مانیتور فعال باشد. اگر یک فرآیند با فراخوانی یک تابع مانیتور، وارد آن مانیتور شود، هیچ فرآیند دیگری نمی‌تواند با فراخوانی همان تابع یا توابع دیگر، وارد آن مانیتور شود، مگر آنکه فرآیند اول با اتمام اجرای تابع، از مانیتور خارج گردد. (مانیتور خالی گردد) و یا درون مانیتور غیرفعال گردد (جلوتر شرح داده می‌شود) اینکه کامپایلر چگونه شرط انحصار متقابل را برقرار می‌کند، از نظر برنامه‌نویس ضروری نمی‌باشد. برنامه‌نویس فقط کافی است بداند با قرار دادن ناحیه‌ی بحرانی مورد نظر خود در داخل یکی از توابع مانیتور، هیچ وقت دو یا چند فرآیند، همزمان وارد این ناحیه‌ی بحرانی نخواهد شد. چرا که اگر فرآیند اولی یک تابع مانیتور را فراخوانی کرده باشد، اگر فرآیند دومی قبل از پایان یافتن کار فرآیند اول با تابع مانیتور، بخواهد همان تابع فراخوانی شده توسط فرآیند اول و یا حتی تابع دیگری از مانیتور را فراخوانی کند، اجازه‌ی چنین کاری به او داده نخواهد شد و این فرآیند دوم در وضعیت مسدود در صف مانیتور قرار داده می‌شود، به عبارت دیگر هنگامی که یکی از فرآیندها یک تابع مانیتور را اجرا می‌کند، اگر پردازنده بر اثر تعویض متن در اختیار فرآیند دیگری قرار گیرد و آن فرآیند تابع در اختیار فرآیند اول و یا حتی تابع از مانیتور دیگری از مانیتور را فراخوانی کند، فرآیند دوم به شکل مسدود در صف مانیتور قرار می‌گیرد.

هنگامی که یک فرآیند پس از اتمام کار با تابع موردنظر مانیتور، از مانیتور خارج می‌گردد، ساختار مانیتور به شکلی ایجاد شده است که باعث می‌شود یکی از فرآیندهای موجود در صف مانیتور به شکل خروج به ترتیب ورود از وضعیت مسدود به آماده منتقل گردد تا شرایط قرارگیری آن فرآیند در صف آماده پردازنده فراهم گردد.

کامپایلر یک زبان برنامه‌نویسی که مانیتور را پشتیبانی می‌کند، در ابتدا و انتهای توابع مانیتور تعدادی دستورالعمل کنترلی قرار می‌دهد. این دستورالعمل‌های کنترلی که از دید برنامه‌نویس پنهان هستند، این امکان را فراهم می‌کنند تا هر وقت یک فرآیند، یک تابع مانیتور را فراخوانی نمود، ابتدا بررسی شود که آیا اکنون فرآیند دیگری داخل مانیتور قرار دارد یا خیر. اگر فرآیندی داخل مانیتور قرار داشت، فرآیند فراخوانی کننده، مسدود و در داخل صف ورود به مانیتور قرار داده می‌شود، در غیر اینصورت وارد مانیتور می‌گردد. همچنین هرگاه اجرای یک تابع مانیتور متعلق به فرآیند فراخوانی کننده پایان یافت، دستورالعمل‌های پایانی و پنهان موجود در انتهای تابع مانیتور باعث می‌گردد، یک فرآیند از صف مانیتور به شکل خروج به ترتیب ورود از وضعیت مسدود به آماده منتقل گردد تا شرایط برقراری آن فرآیند در صف آماده پردازنده فراهم گردد.

کامپایلر برای پیاده‌سازی مانیتور و تحقق انحصار متقابل و نیز صف‌بندی فرآیندهای منتظر ورود به مانیتور، می‌تواند در ورودی مانیتور از سمافور mutex استفاده کند. مثلاً در ابتدای همه‌ی توابع مانیتور، یک تابع wait (mutex) و در انتهای همه توابع مانیتور، یک تابع signal (mutex) قرار دهد. بدیهی است این نحوه‌ی پیاده‌سازی و برقراری شرط انحصار متقابل درون مانیتور، توسط سمافور از دید برنامه‌نویس سطح کاربر پنهان است و توسط کامپایلر انجام می‌گردد.

توجه: شمارنده سمافور mutex در این حالت، برای همه توابع مانیتور به صورت مشترک مورد استفاده قرار می‌گیرد.

مانیتور مانند یک ساختمان چند طبقه می‌باشد، که در هر طبقه یک عامل مشترک خاص قرار داده شده است. مثلاً در طبقه‌ی اول استخر و سونا، در طبقه‌ی دوم امکانات ورزشی، در طبقه‌ی سوم سینما و در طبقات بعدی هم عامل‌های مشترک خاص دیگری قرار داده شده است. اما مطابق قوانین این ساختمان، در هر لحظه فقط یک نفر می‌تواند داخل این ساختمان حضور داشته باشد. تا با خیالی آسوده و راحت از عامل مشترک موجود در یک طبقه‌ی خاص استفاده کند. مثلاً اگر فرد اولی در طبقه‌ی سوم و در حال استفاده از عامل مشترک سینما باشد و افراد دیگری علاقه‌مندی خود را برای ورود به طبقه‌ی سوم و یا حتی طبقات دیگر اعلام کنند، نگهبان ساختمان جلوی این افراد را می‌گیرد و آن‌ها را به ترتیب به صف ورودی درب ساختمان هدایت می‌کند. تا وقتی که فرد اول از طبقه‌ی سوم و به تبع از ساختمان خارج گردد. نگهبان ساختمان پس از مشاهده‌ی خروج این فرد، یک نفر را به شکل خروج به ترتیب ورود از صف ورودی درب ساختمان انتخاب و به ساختمان راه می‌دهد!

مثال: کنترل شرایط رقابتی در مانیتور

دو فرآیند هم روند P_1 و P_2 در یک سیستم اشتراک زمانی که از متغیر مشترک سراسری s در بخشی از کد خود استفاده می‌کنند، در نظر بگیرید، شرط انحصار متقابل را توسط مانیتور حل کنید؛ (متغیر s

داخل ناحیه‌ی بحرانی استفاده می‌گردد)

P_1	P_2
⋮	⋮
critical_section	critical_section
⋮	⋮

برای حل این مسأله کافی است، نواحی بحرانی دو فرآیند داخل توابع مانیتور قرار گیرند:

```
Monitor cs
s: integer;
procedure prc1
begin
critcal_section;
end;
procedure prc2
begin
critical_section
end;
s: = 0;
end Monitor
```

حال استفاده از نواحی بحرانی فرآیندهای P_1 و P_2، پس از قرار دادن نواحی بحرانی داخل توابع مانیتور به شکل زیر بازنویسی می‌شود:

P_1	P_2
⋮	⋮
cs.prc1	cs.prc2
⋮	⋮

توجه: در واقع نواحی بحرانی مربوط به فرآیندهای P_1 و P_2 داخل توابع prc1 و prc2 موجود در مانیتور قرار گرفتند و مطابق قوانین مانیتور هیچ دو فرآیندی نمی‌توانند به طور همزمان وارد مانیتور شوند، به بیان دیگر در هر لحظه فقط یک فرآیند می‌تواند داخل یک مانیتور فعال باشد. بنابراین شرط انحصار متقابل برقرار است.

همگام سازی با استفاده از مانیتور (کنترل شرایط رقابتی + همگام‌سازی)

مانیتور برای کنترل شرایط رقابتی توام با همگام‌سازی، شامل مجموعه‌ای از متغیرهای داده‌ای، متغیرهای شرطی و توابع می‌باشد که در یک ماژول بسته‌بندی شده است. در برخی مسائل مانند تولیدکننده و مصرف‌کننده علاوه بر نیاز به کنترل شرایط رقابتی مطابق آنچه پیش از این در مورد مانیتور گفتیم، ما به راه حلی نیاز داریم که وقتی فرآیندها نمی‌توانند پیشروی کنند، داخل خود مانیتور مسدود شوند، در مسأله تولیدکننده و مصرف کننده قرار دادن تست‌هایی برای تشخیص پُر یا خالی بودن بافر در داخل تابع مانیتور ساده است. اما چگونه باید یک تابع را هنگامی که متوجه می‌شود بافر پُر است، مسدود کرد؟ راه حل را باید در متغیرهای شرطی مانیتور جستجو کرد. بنابراین ویژگی دیگر مانیتور این است که علاوه بر تعریف متغیرهای داده‌ای مثل، integer، می‌توان متغیرهای شرطی از نوع conditon تعریف کرد، به این‌ها متغیرهای شرطی (condition variables) گفته می‌شود.

بر روی متغیر شرطی x یک مانیتور، دو تابع خاص wait (x) و signal (x) عملیات همگام‌سازی بین دو فرآیند را کنترل می‌کنند:

تابع wait (x):

عملیات آن به ترتیب شامل وارد کردن یک فرآیند به صف متغیر شرطی مانیتور و خواباندن (تغییر وضعیت از اجرا به مسدود) همان فرآیند است.

ساختار این تابع به صورت زیر است:

```
wait (condition x)
{
   add this process to x.queue;
   block ();
}
```

توجه: به تفاوت تابع wait در سمافور و مانیتور دقت کنید، در تابع wait مانیتور، هیچ تغییری، بر روی مقدار متغیر شرطی x اعمال نمی‌گردد و فقط عمل درج یک فرآیند، داخل صف متغیر شرطی و خواباندن همان فرآیند انجام می‌گردد.

شرح تابع: فراخوانی تابع wait (x) متعلق به متغیر شرطی x، توسط یک تابع مانیتور مورد استفاده‌ی یک فرآیند موجود در مانیتور باعث می‌شود تا فرآیندی که در حال حاضر داخل مانیتور قرار دارد، داخل صف متغیر شرطی مانیتور قرار گرفته و توسط تابع ()blockr مسدود شود، یعنی از وضعیت اجرا به وضعیت مسدود منتقل گردد. دقت کنید که فرآیند مسدود شده است اما همچنان داخل مانیتور قرار دارد، اگر فرآیندی توسط تابع wait متعلق به یک متغیر شرطی داخل یک مانیتور مسدود گردد، در این

شرایط فرآیند رقیب می‌تواند وارد مانیتور گردد، زیرا فرآیند اول داخل مانیتور قرار دارد، اما فعال نیست! (خوابیده است). بنابراین ممکن است در یک لحظه بیش از یک فرآیند داخل مانیتور قرار بگیرد، اما فقط یکی از آن‌ها فعال باشد. یکی خواب و دیگری بیدار!

تابع signal (x):

عملیات آن به ترتیب شامل خارج کردن یک فرآیند به شکل خروج به ترتیب ورود از صف متغیر شرطی مانیتور و بیدار کردن (تغییر وضعیت از مسدود به آماده) همان فرآیند است.

ساختار این تابع به صورت زیر است:

```
siganl (condition x)
  {
    remove a process from queue;
    wakeup ();
  }
```

توجه : به تفاوت تابع signal در سمافور و مانیتور دقت کنید، در تابع signal مانیتور، هیچ تغییری، بروی مقدار متغیر شرطی x اعمال نمی‌گردد و فقط عمل حذف یک فرآیند، از صف متغیر شرطی موردنظر به شکل خروج به ترتیب ورود و بیدار کردن همان فرآیند، انجام می‌گردد.

شرح تابع: فراخوانی تابع signal(x) متعلق به متغیر شرطی x، توسط یک تابع مانیتور مورد استفاده‌ی یک فرآیند موجود در مانیتور باعث می‌شود یک فرآیند به شکل خروج به ترتیب ورود از صف متغیر شرطی مانیتور خارج شده و توسط تابع ()wakeup بیدار شود، یعنی از وضعیت منتظر به وضعیت آماده منتقل گردد.

نکته : هنگامی که یک فرآیند از ابتدای صف یک متغیر شرطی توسط تابع signal متعلق به یک متغیر شرطی، بیدار و فعال می‌گردد، قادر خواهد بود تا ادامه دستورات تابع مانیتور را که قبلاً به دلیل مسدود شدن انجام نشده بود، ادامه دهد و تابع مانیتور را تمام کند و از مانیتور خارج گردد.

توجه : تابع signal (x) متعلق به متغیر شرطی x، در صورت استفاده در یک تابع مانیتور باید آخرین دستور تابع مانیتور باشد. زیرا بر اثر اجرای تابع signal، یک فرآیند از ابتدای صف متغیر شرطی که قبلاً توسط تابع wait (x) متعلق به متغیر شرطی x مسدود شده بود، فعال می‌گردد. و اگر بعد از تابع signal باز هم دستور باشد، بدین ترتیب در یک زمان دو فرآیند فعال در مانیتور وجود خواهد داشت و این برخلاف قوانین مانیتور است. بنابراین برای برقراری قوانین داخلی مانیتور، تابع signal در صورت استفاده در توابع مانیتور باید آخرین دستور تابع مانیتور باشد.

در ادامه حل مسأله کلاسیک تولیدکننده و مصرف‌کننده با استفاده از راه حل مانیتور مورد بررسی قرار

می‌گیرد.

حل مسأله تولیدکننده و مصرف کننده توسط مانیتور (با بافر محدود)

```
Monitor ProducerConsumer;
count: integer;
full , empty: condition;
procedure insert (item: integer)
begin
   if count = N then wait (full);
   insert_item (item) درج قطعه تولید شده در بافر (ناحیه‌ی بحرانی)
   count : = count + 1 ;
   if count = 1 then signal (empty);
end;
function remove: integer
begin
   if count =  then wait (empty);
   remove: = remove_item;
   count: = count - 1;
   if count = N - 1 then signal (full)
end;
count : = 0 ;
end monitor

procedure producer
begin
   while true do
   begin
   item: = produce - item; تولید قطعه
   ProducerConsumer.insert (item) فراخوانی تابع درج مانیتور، برای درج قطعه تولید شده
   end
```

```
end;
Procedure consumer
begin
while true do
   begin
   item: = ProducerConsumer.remove; فراخوانی تابع حذف مانیتور برای حذف قطعه
   consume_item (item) مصرف قطعه
   end
End.
```

فرآیند تولیدکننده

وقتی فرآیند تولیدکننده توسط تابع insert مانیتور متوجه می‌شود که به دلیل پُر شدن بافر دیگر قادر به ادامه نیست، تابع wait (full) را بر روی متغیر شرطی full فراخوانی می‌کند، این کار باعث می‌شود، فرآیند تولیدکننده در صف متغیر شرطی full قرار گیرد و در داخل مانیتور مسدود گردد، در این لحظه به دلیل اینکه هیچ فرآیندی در داخل مانیتور فعال نیست، به فرآیند مصرف کننده که قبلاً درخواست ورود به مانیتور را داشته و در صف ورود به مانیتور به شکل مسدود قرار گرفته است. اجازه‌ی ورود به داخل مانیتور داده می‌شود. فرآیند مصرف‌کننده می‌تواند از طریق اجرای تابع signal بـر روی هـمان متغیر شرطی full که فرآیند تولیدکننده روی آن wait کرده بود، او را بیدار سازد.

توجه :انحصار متقابل خودکار در توابع مانیتور تضمین می‌کند که اگر مثلاً تـولیدکننده در درون تـابع insert متوجه شود که بافر پُر است، قادر خواهد بود عملیات wait را کامل کند و بخوابد و نگران این نباشد که مبادا زمان‌بند دقیقاً قبل از تکمیل wait به فرآیند مصرف‌کننده سوئیچ کند، زیرا در این صورت فرآیند مصرف کننده فعلاً اجازه‌ی ورود به مانیتور را پیدا نخواهد کرد و باید صبر کند تا تولیدکننده، فراخوان wait را تمام کند و بخوابد.

فرآیند مصرف کننده

به همین ترتیب وقتی فرآیند مصرف کننده توسط تابع remove مانیتور متوجه می‌شود که به دلیل خالی بودن بافر دیگر قادر به ادامه نیست، تابع wait (empty) را بر روی متغیر شرطی empty فراخوانـی می‌کند، این کار باعث می‌شود، فرآیند مصرف‌کننده در صف متغیر شرطی empty قرارگیرد و در داخل مانیتور مسدود گردد، در این لحظه به دلیل اینکه هیچ فرآیندی در داخل مانیتور فعال نیست، به فرآیند تولیدکننده که قبلاً درخواست ورود به مانیتور را داشته و در صف ورود به مانیتور به شکل مسدود قرار گرفته است، اجازه‌ی ورود به داخل مانیتور داده می‌شود. فرآیند تولیدکننده می‌تواند از طریق اجرای

تابع signal بر روی همان متغیر شرطی empty که فرآیند مصرف کننده روی آن wait کرده بود، او را بیدار سازد.

توجه :انحصار متقابل خودکار در توابع مانیتور تضمین می‌کند که اگر مثلاً مصرف کننده در درون تابع remove متوجه شود که بافر خالی است، قادر خواهد بود عملیات wait را کامل کند و بخوابد و نگران این نباشد که مبادا زمان‌بند دقیقاً قبل از تکمیل wait به فرآیند تولیدکننده سوئیچ کند، زیرا در این صورت فرآیند تولیدکننده فعلاً اجازه‌ی ورود به مانیتور را پیدا نخواهد کرد و باید صبر کند تا مصرف‌کننده فراخوان wait را تمام کند و بخوابد.

راه حل تبادل پیام

هدف از شبکه‌های کامپیوتری تبادل داده میان یک مبدأ و یک مقصد مورد نظر است. با قرار دادن داده (پیام) و آدرس مبدأ و مقصد داخل یک بسته، می‌توان این بسته را به سمت مقصد ارسال و هدایت نمود. بنابراین پیام یک مکانیزم ساده و مناسب جهت تبادل داده و همگام‌سازی و ارتباط بین فرآیندهاست که قابل استفاده در سیستم‌های تک پردازنده‌ای و چند پردازنده‌ای و همچنین سیستم‌های توزیع شده می‌باشد. ارتباط بین فرآیندها توسط دو تابع سیستمی send و receive انجام می‌گردد.

تابع (پیام و آدرس مقصد) send: ارسال پیام به مقصد مورد نظر

تابع (پیام و آدرس مبداء) receive: دریافت پیام از یک مبداء مورد نظر

توجه : یک پیام برای رسیدن به مقصد مورد نظر نیاز به سه آدرس PORT و IP و MAC دارد.

همگام سازی فرستنده و گیرنده

حالت‌های فرآیند فرستنده

حالت همگام: فرآیند فرستنده پس از ارسال پیام تا آمدن پیام تصدیق (ACK) مبنی بر درست رسیدن پیام مسدود می‌گردد. زیرا ممکن است پیام با خطا به مقصد برسد، یا اصلاً به مقصد نرسد، بنابراین پیام دوباره باید ارسال گردد، حال اگر فرآیند فرستنده پس از ارسال مسدود نشود، ممکن است فرآیند فرستنده، پیام بعدی را بر روی پیام فعلی روی نویسی کند. در حالی که شاید ارسال دوباره پیام فعلی لازم باشد.

حالت ناهمگام: فرآیند فرستنده، به شکل چند نخی می‌باشد و هر نخ وظیفه‌ی خاصی را بر عهده دارد. در این حالت نخی که وظیفه‌ی ارسال را بر عهده دارد، پس از ارسال پیام تا آمدن پیام تصدیق (ACK) مبنی بر درست رسیدن پیام مسدود می‌گردد. اما سایر نخ‌ها به وظایف خود می‌پردازند، مثلاً نخی که آماده‌سازی پیام بعدی را بر عهده دارد، به انجام وظایف خود می‌پردازد.

حالت‌های فرآیند گیرنده

حالت همگام: فرآیند گیرنده پس از بررسی بافر و مشاهده‌ی خالی بودن آن، تا دریافت پیام از سوی فرستنده، مسدود می‌گردد.

حالت ناهمگام: فرآیند گیرنده، به شکل چند نخی می‌باشد و هر نخ وظیفه‌ی خاصی را بر عهده دارد. در این حالت نخی که وظیفه‌ی دریافت را بر عهده دارد، تا دریافت پیام از سوی فرستنده، مسدود می‌گردد. اما سایر نخ‌ها به وظایف خود می‌پردازند.

قابلیت اطمینان

در حین تبادل داده ما بین فرستنده و گیرنده، ممکن است شبکه پیام‌ها را گم کند، برای حفاظت در مقابل گم شدن پیام‌ها، فرستنده و گیرنده می‌توانند با یکدیگر توافق کنند که به محض رسیدن پیام، گیرنده یک ACK (پیام تصدیق) مخصوص بفرستد (Acknowledgement). اگر فرستنده در یک فاصله زمانی مشخص و از قبل تعیین شده، ACK را دریافت نکرد. یعنی time out شد، پیام را دوباره ارسال می‌کند. در این حالت پیام اصلاً به مقصد نرسیده است که بخواهد ACK تولید شود حال فرض کنید که پیام درست برسد ولی ACK در راه بازگشت گم شود.

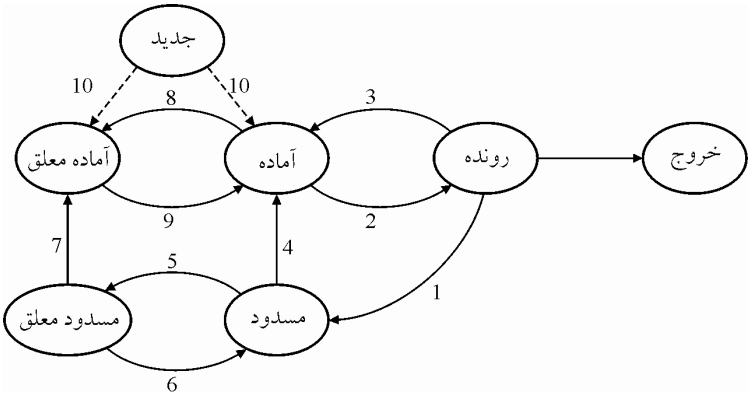
بنابراین باز هم time out می‌شود و فرستنده پیام را دوباره ارسال خواهد کرد. بنابراین گیرنده دوباره آن را دریافت می‌کند. فرض کنید دو بار پیام برداشت از حساب شما اجرا گردد، چه می‌شود؟ بنابراین این مسأله بسیار ضروری است که گیرنده بتواند یک پیام جدید را از ارسال مجدد یک پیام قدیمی تشخیص دهد. معمولاً این مسأله با قرار دادن شماره ترتیب در پیام‌ها حل می‌شود. اگر گیرنده یک پیام دریافت کند که شماره ترتیب آن مشابه شماره قبلی باشد، متوجه می‌شود که این پیام تکراری است و از آن چشم‌پوشی می‌کند.

لازم به ذکر است که پیام ACK که از سوی گیرنده به فرستنده ارسال می‌شود حامل دو پیام مهم است: (۱) درست رسیدن پیام فعلی (۲) اعلام شماره ترتیب پیام بعدی.

مثلاً اگر گیرنده یک پیام با شماره ترتیب ۱، را درست دریافت کند، در پیام ACK که به سمت فرستنده ارسال می‌کند، شماره ترتیب ۲ را از فرستنده درخواست می‌کند تا بفرستد. حال اگر این ACK گم شود، فرستنده همان پیام شماره ترتیب ۱ را دوباره ارسال می‌کند، اما گیرنده منتظر دریافت پیام شماره ۲ است، بنابراین متوجه می‌شود این پیام شماره با ترتیب ۱ تکراری است و آن را دور می‌اندازد و مجدداً ACK را ارسال می‌کند و به فرستنده می‌گوید منتظر پیامی با شماره ترتیب ۲ است.

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۱- شکل زیر تغییر حالت‌های یک فرآیند را نشان می‌دهد. تغییر حالت از رونده به خروج، امکان دارد باعث کدام تغییر حالت‌ها شود؟ (مهندسی IT- دولتی۸۹)



۱) 10,7,4
۲) 2,4,1
۳) 8,5,2
۴) 9,6

۲- پنج فرآیند (P_0 تا P_4)، با مشخصات زمان اجرای نشان داده شده در شکل، به ترتیب P_0 تا P_4 در صف آماده قرار دارند. هر فرآیند فقط در محل‌های مشخص شده در آن، عملیات P(s) یا (wait(s)) و V(s) یا (signal(s)) را بر روی سمافورs با مقدار اولیه یک اجرا می‌کند. اعداد نوشته شده در کنار آکولادها نشان‌دهنده طول زمان اجرای آن بخش از فرآیندها می‌باشند. برای زمان‌بندی آنها الگوریتم RR(Round Robin) با زمان کوانتوم q=5 استفاده می‌شود. متوسط زمان بازگشت (turn around) و متوسط زمان انتظار فرآیندهای فوق با زمان‌بندی مذکور به ترتیب چقدر است؟ توجه شود که فرآیندها براساس FIFO از حالت بلوکه شده (Blocked) خارج می‌شوند. ضمناً زمان اجرای P(s) و V(s) را ناچیز در نظر بگیرید. (مهندسی کامپیوتر – دولتی۹۰)

P_0 : code1( );}10 P_1 : code3( );}40

P(s);
code2( );}15 P_2 : code4( );}50
V(s);

P_3 : code5( );}10 P_4 : code7( );}15

P(s); P(s);
code6( );}15 code8( );}10
V(s); V(s);

۱) 141، 108
۲) 135، 102

۳) 133، 99

۴) بن‌بست رخ می‌دهد و اجرای فرآیندها به اتمام نمی‌رسد.

۳- آیا می‌توان یک مانیتور را از داخل یک مانیتور دیگر فراخوانی کرد؟ (مهندسی کامپیوتر - دولتی۹۰)

۱) امکان‌پذیر است ولی می‌تواند به بن‌بست بیانجامد.

۲) امکان‌پذیر است ولی نتیجه غلط خواهد داد.

۳) امکان‌پذیر است و مشکلی ندارد.

۴) امکان‌پذیر است به شرطی که زنجیره‌وار ادامه نیابد و فقط شامل دو مانیتور باشد.

۴- پیاده‌سازی زیر از عملیات تجزیه‌ناپذیر (atomic) روی سمافورها را در نظر بگیرید:

```
typedef struct{
  int value;
  struct process*List;
}semaphore;
wait(semaphore*S){
     S->value--;
     if(S->value<0){
     add this process to S->List;
      block(-(S->value));
   }
}
signal(semaphore*S){
   S->value++;
   if(S->value<=0){
      remove a process P from S->List;
      wakeup(P);
   }
}
```

مفروضات و تعاریف زیر را نیز داریم: (*initialization*) S-> value= 1

block(n): فرآیندهای بلوک شونده را به ترتیب n از کوچک به بزرگ مرتب کرده که بـه ایـن ترتیـب فرآیند اول (بلوک شونده) با block(1) در سر صف قرار می‌گیرد و به ترتیب n (از کوچک به بـزرگ) توسط wakeup(P) از حالت بلوک خارج می‌شوند. برنامه‌های سیستم به شکل زیر مفروضند:

```
mutex : semphore;
do{
    wait(mutex);
      Critical – sec tion
   si gnal (mutex);
     Re mainder – sec tion
} while (TRUE);
```

کدام گزینه صحیح است؟ (مهندسی IT- دولتی۹۰)

۱) خواص انحصار متقابل و انتظار محدود برقرارند ولی پیشرفت برقرار نیست.
۲) خواص انحصار متقابل و پیشرفت برقرارند ولی انتظار محدود برقرار نیست.
۳) خواص پیشرفت و انتظار محدود برقرارند ولی انحصار متقابل برقرار نیست.
۴) خاصیت انحصار متقابل برقرار است ولی پیشرفت و انتظار محدود برقرار نیستند.

۵- در فضاپیمای «راه‌یاب» ۳ کار مهم در نرم‌افزار آن تعبیه شده است که عبارتند از:
T_1: به صورت دوره‌ای سلامت سیستم‌ها و نرم‌افزار فضاپیما را چک می‌کند.
T_2: داده‌های تصویری را پردازش می‌کند.
T_3: هر از گاهی بر روی وضعیت تجهیزات آزمایش می‌کند.
اولویت سه کار به ترتیب T_1، T_2 و T_3 هستند. یعنی T_1 بالاترین اولویت و T_3 پایین‌ترین را دارند. هر کار که اولویت بالاتر داشته باشد و آماده باشد کار دیگر را قبضه (preempt) می‌کند. در هر بار اجرای T_1 یک تایمر به بالاترین مقدار خود مقداردهی می‌شود. اگر احیاناً زمان تایمر منقضی شود، فرض می‌شود که مشکلی در اجرای نرم‌افزار فضاپیما به وجود آمده است. در این حالت تمام پردازش‌ها متوقف می‌شوند و نرم‌افزار به طور کامل بار می‌شود و تمام سیستم‌ها آزمایش می‌شوند و همه چیز از نقطه شروع آغاز می‌شود. T_1 و T_3 در یک ساختار داده‌ای مشترک هستند و برای دسترسی به آن از سمافور باینری S استفاده می‌کنند. سناریوی زیر را در نظر بگیرید که به ترتیب پیش می‌رود.
۱- T_3 شروع به کار می‌کند.
۲- T_3 سمافور S را در اختیار می‌گیرد و وارد ناحیه بحرانی می‌شود.
۳- T_1 که دارای الویت بالاتری است T_3 را قبضه می‌کند و شروع به اجرا می‌کند.
۴- T_1 اقدام به ورود به ناحیه بحرانی می‌کند ولی بلوک می‌شود. T_3 کار خود در ناحیه بحرانی را پی می‌گیرد.
۵- T_2، T_3 را قبضه می‌کند و شروع به اجرا می‌کند.
۶- T_2 به دلیلی مستقل از T_1 و T_3، معلق می‌شود. T_3 دوباره ادامه می‌دهد.
۷- T_3 ناحیه بحرانی را ترک می‌کند و سمافور S آزاد می‌شود.
۸- T_1، T_3 را قبضه می‌کند و سمافور را در اختیار می‌گیرد و وارد ناحیه بحرانی می‌شود.

(مهندسی کامپیوتر – دولتی۹۱)

۱) در این سناریو ممکن است مشکل زمانی به وجود آید و اگر اولویت T_2 را کمتر از T_3 قرار دهیم مشکل حل می‌شود.

۲) در این سناریو اولویت داشتن T_1 نسبت به T_2 و T_3 خود را نشان می‌دهد و سیستم به درستی کار می‌کند.

۳) اگر بین کارها سهم زمانی برقرار کنیم زمان پاسخ تضمین می‌شود و مشکلات احتمالی زمانی از بین می‌روند.

۴) این سیستم به درستی کار نمی‌کند و می‌تواند شکست بخورد و تایمر منقضی شود.

۶- سه Thread زیر را در نظر بگیرید که به صورت همروند در سیستم اجرا می‌شوند.

(مهندسی IT- دولتی۹۱)

Fig A

```
=== Thread A ===
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock4);
...
pthread_mutex_unlock(&lock4);
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock1);

=== Thread B ===
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock3);
pthread_mutex_lock(&lock1);
...
pthread_mutex_unlock(&lock1);
pthread_mutex_unlock(&lock3);
pthread_mutex_unlock(&lock2);

=== Thread C ===
pthread_mutex_lock(&lock3);
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock4);
...
pthread_mutex_unlock(&lock4);
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock3);
```

Fig B

```
=== Thread A ===
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock4);
pthread_mutex_lock(&lock2);
...
pthread_mutex_unlock(&lock4);
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock1);

=== Thread B ===
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock3);
...
pthread_mutex_unlock(&lock1);
pthread_mutex_unlock(&lock3);
pthread_mutex_unlock(&lock2);

=== Thread C ===
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock3);
pthread_mutex_lock(&lock4);
...
pthread_mutex_unlock(&lock4);
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock3);
```

Fig C

```
=== Thread A ===
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock4);
...
pthread_mutex_unlock(&lock4);
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock1);

=== Thread B ===
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock3);
...
pthread_mutex_unlock(&lock3);
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock1);

=== Thread C ===
pthread_mutex_lock(&lock3);
pthread_mutex_lock(&lock2);
pthread_mutex_lock(&lock4);
...
pthread_mutex_unlock(&lock4);
pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock3);
```

Fig D

```
=== Thread A ===
pthread_mutex_lock(&lock1);
pthread_mutex_unlock(&lock4);
pthread_mutex_lock(&lock2);
pthread_mutex_unlock(&lock2);
pthread_mutex_lock(&lock4);
pthread_mutex_unlock(&lock1);
...

=== Thread B ===
pthread_mutex_lock(&lock2);
pthread_mutex_unlock(&lock1);
pthread_mutex_lock(&lock3);
pthread_mutex_unlock(&lock3);
pthread_mutex_lock(&lock1);
pthread_mutex_unlock(&lock2);
...

=== Thread C ===
pthread_mutex_lock(&lock3);
pthread_mutex_unlock(&lock4);
pthread_mutex_lock(&lock2);
pthread_mutex_unlock(&lock2);
pthread_mutex_lock(&lock4);
pthread_mutex_unlock(&lock3);
...
```

۱) کد Fig A می‌تواند منجر به بن‌بست شود و راه‌حل آن هر یک از کـدهای Fig B یـا Fig C می‌باشد.
۲) کدهای Fig A، Fig B، Fig C و Fig D همگی معادل هستند و بن‌بست ندارند.
۳) کد Fig A می‌تواند منجر به بن‌بست شود و راه‌حل آن فقط کد Fig B می‌باشد.
۴) کد Fig A می‌تواند منجر به بن‌بست شود و راه‌حل آن کد Fig D می‌باشد.

۷- آیا کد زیر می‌تواند راه حلی برای دو پردازش همروند باشد؟ (مهندسی IT- دولتی ۹۱)

```
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;
```

۱) راه حل صحیح نیست زیرا انحصار متقابل (mutual exclusion) رعایت نمی‌شود.
۲) راه حل صحیح نیست زیرا شرط پیشرفت برقرار نیست.
۳) راه حل صحیح نیست زیرا تضمینی برای محدودیت زمان انتظار ندارد.
۴) راه حل صحیح است.

۸- الگوریتم زیر یک راه‌حل نرم‌افزاری برای حل مسئله ناحیه بحرانی برای دو فرآیند است. در ایـن راه‌حل هر فرآیند تلاش می‌کند بی‌نهایت بار وارد ناحیه بحرانی شود. هر فرآینـد بـرای ورود بـه ناحیه بحرانی تابع Wait(i) و برای خروج از ناحیه بحرانی تابع Signal(i) را فراخوانی می‌کند کـه $i \in \{0,1\}$ شماره فرآیند است. c نیز یک آرایه با طول ۲ از متغیرهای دودویی است که با مقـدار true پر شده است. (مهندسی کامپیوتر – دولتی۹۲)

```
Wait(i){                        Signal(i){
    c[i] = false;                   c[i] = true;
    while(c[1 − i])do;              }
}
```

کدام یک از گزینه‌های زیر درست نیست؟

۱) این راه‌حل استفاده انحصاری از ناحیه بحرانی را برآورده می‌کند.

۲) این راه‌حل شرط انتظار محدود را برآورده می‌کند.
۳) این راه‌حل همه شرایط ناحیه بحرانی را برآورده می‌کند.
۴) این راه‌حل شریط پیشرفت را برآورده می‌کند.

۹- الگوریتم زیر یک راه حل نرم‌افزاری برای حل مسئله بحرانی برای دو فرآیند است. در این راه حل هر دو فرآیند تلاش می‌کنند بی‌نهایت بار وارد ناحیه بحرانی شوند. هر فرآیند برای ورود به ناحیه بحرانی تابع Wait (i) و برای خروج از ناحیه بحرانی تابع Signal را فراخوانی می‌نماید که $i \in \{0,1\}$ شماره فرآیند است. turn یک متغیر از نوع عدد صحیح و دارای مقدار اولیه یک و c یک آرایه با طول 2 از متغیرهای دودویی است که با مقدار ture پر شده است. (مهندسی IT- دولتی۹۲)

```
Wait(i){                          Signal(i){
  c[i] = true;                      c[i] = false :
  turn = 1 − i                    }
  while(c[i] & &turn = 1 − i) do;
}
```

کدام یک از گزینه‌های زیر درست نیست؟

۱) این راه حل همه شرایط ناحیه بحرانی را برآورده می‌کند.
۲) این راه حل تنها شرط انتظار محدود را برآورده می‌کند.
۳) این راه حل تنها استفاده انحصاری از ناحیه بحرانی را برآورده می‌کند.
۴) این راه حل تنها شرط پیشرفت را برآورده می‌کند.

۱۰- کدام یک از روش‌های زیر برای پیاده‌سازی سمافور در سیستم با چند پردازنده، مناسب است؟
(مهندسی IT- دولتی۹۳)

۱) غیرفعال نمودن وقفه‌ها
۲) استفاده از ویژگی‌های زبان سطح بالا
۳) استفاده از متغیر Flag و روش Busy Waiting
۴) استفاده از دستور Test and Set Lock با Busy Waiting

۱۱- کدامیک از روش‌های زیر برای پیاده‌سازی سمافور در سیستم با چند پردازنده، مناسب است؟
(مهندسی کامپیوتر-دولتی۹۴)

۱) با استفاده از دستور Test & Set و Busy Waiting.
۲) استفاده از Flag و Busy Waiting.
۳) با استفاده از ویژگی‌های زبان‌های سطح بالا.
۴) غیر فعال نمودن وقفه‌ها

۱۲- فرض کنید که دو ریسه (Thread) قطعه کدهای زیر را به صورت همروند اجرا نمایند. در این قطعه کدها، ریسه‌ها به متغیرهای مشترک a و b و c دسترسی دارند. مقادیر ممکن برای c پس از اجرای این قطعه کدها کدام است؟ (مهندسی کامپیوتر-دولتی۹۴)

```
Initialization      Thread 1              Thread 2
a=4;                if (a<b) then         b=10;
b=0;                    c=b-a;            c=-3
c=0;                else
                        c=b+a;
                    endif
```

۱) 4,7,6,-3

۲) 4,7,6,13,-3

۳) 4,1,6,-3,14

۴) 4,7,6,13,-3,14

۱۳- در مسئله غذا خوردن فیلسوف‌ها، 5 فیلسوف دور میزی نشسته‌اند و بین هر دو فیلسوف یک چنگال قرار دارد و هر فیلسوف برای غذا خوردن به دو چنگال نیاز دارد. فرض کنید دو نوع فیلسوف داریم. فیلسوفان چپ دست که ابتدا چنگال سمت چپ خود را برمی‌دارند و فیلسوفان راست دست که ابتدا چنگال سمت راست خود را برمی‌دارند. فرض کنید که در بین 5 فیلسوف، حداقل یک فیلسوف چپ دست و یک فیلسوف راست دست موجود است. با توجه به توضیحات فوق، کدام عبارت صحیح است؟ (مهندسی کامپیوتر-دولتی۹۴)

۱) اگر دو تا فیلسوف چپ دست یا دو فیلسوف راست دست کنار هم باشند بن بست رخ می‌دهد.

۲) مستقل از نحوه‌ی نشستن فیلسوفان چپ دست و راست دست، هیچگاه بن بست رخ نمی‌دهد.

۳) اگر از یک نوع فیلسوف، دو تا و از نوع دیگر سه تا داشته باشیم بن بست رخ می‌دهد.

۴) اگر همگی فیلسوف‌ها با هم همزمان اولین چنگال‌ها را بردارند، بن بست رخ می‌دهد.

۱۴- سه پردازه همزمان به صورت زیر در حال اجرا هستند. در این پردازه‌ها از سه سمافور باینری استفاده شده‌است که مقادیر اولیه آنها به ترتیب عبارتند از : S2=0، S1=0، S0=1

(مهندسی IT – دولتی ۹۴)

```
Proccess P0                Proccess P1          Proccess P2
while(true){               wait(S1);            wait(S2);
    wait(S0);              release(S0);         release(S0);
    print '0';
    release(S1);
    release(S2);
}
```

در این حالت، پردازه P0 چند بار مقدار 0 را چاپ می‌کند؟

۱) دقیقاً دو بار

۲) حداقل دو بار

۳) دقیقاً سه بار

۴) حداقل سه بار

۱۵- سه سمافور با مقدار اولیه $x=1$، $y=5$ و $z=10$ در نظر بگیرید. قطعه کد زیر توسط 20 پردازه (process) اجرا می‌شود. حداکثر طول صفی که برای سمافور y تشکیل می‌شود، چقدر است؟ (مهندسی کامپیوتر- دولتی ۹۵)

```
…
z.wait();
…
y.wait();
...
x.wait();
...
x.signal();
…
z.signal();
…
y.signal();
```

۱) 4 ۲) 5 ۳) 9 ۴) 10

۱۶- کدام مورد، درباره ایمن بودن یک تابع در سطح نخ (Thread Safe) درست است؟

(مهندسی IT- دولتی ۹۶)

۱) یک تابع در سطح نخ ایمن است و اگر و فقط اگر از نخ‌های مختلف فراخوانی شود همیشه نتیجه درست را برگرداند.

۲) یک تابع در سطح نخ ایمن است اگر و فقط اگر از نخ‌های همروند فراخوانی شود همیشه نتیجه درست را برگرداند.

۳) یک تابع در سطح نخ ایمن است اگر از نخ‌های همروند فراخوانی شود همیشه نتیجه درست را برگرداند.

۴) یک تابع در سطح نخ ایمن است اگر از نخ‌های مختلف فراخوانی شود همیشه نتیجه درست را برگرداند.

پاسخ تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۱- گزینه (۴) صحیح است.

طراح محترم این سؤال، دقت لازم و کافی را برای طرح گزینه‌ها نداشته است و علت می‌تواند به این دلیل باشد که طراح محترم، فرایندهای مستقل از هم را در نظر داشته است و به مسأله ناحیه بحرانی فرآیندهای هم‌روند توجهی نداشته است.

البته این بدین معنی نیست که تفکر طراح درست بوده است، بلکه به عکس، طراح محترم همه جوانب مسأله را در نظر نگرفته است و دچار خطای فکری شده‌اند، بهتر بود، شرایط و فرضیات فکری خود را مطرح می‌نمودند که آن هم مطرح نشده است.

سازمان سنجش آموزش کشور گزینه چهارم را به عنوان پاسخ نهایی اعلام کرده است.

به طور کلی هنگامی که یک فرآیند از حالت اجرا (رونده) به حالت خروج می‌رود و به تبع از سیستم خارج می‌گردد، دو نتیجه ابتدایی زیر حاصل می‌گردد:

اول اینکه، پردازنده آزاد می‌گردد.

دوم اینکه، بلوک‌هایی از حافظه آزاد می‌گردد.

حال اگر برای پاسخ به این سؤال، مانند طراح فکر کنیم و فرآیندها را مستقل از هم در نظر بگیریم، آنگاه گزینه اول، دوم و سوم نادرست و گزینه چهارم درست خواهد بود. در این شرایط تغییر حالت 4 و 7 امکان پذیر نیست، چون فرآیندها مستقل از یکدیگر هستند. مثلاً مطابق مباحث فرآیندهای هم‌روند، یک فرآیند رقیب علاقه‌مند به ورود به ناحیه بحرانی، به دلیل حضور یک فرآیند دیگر در ناحیه بحرانی توسط راه حل سمافور مسدود نشده است، که حال بخواهد به محض خروج فرآیند داخل ناحیه بحرانی از سیستم به واسطه عمل بیدارسازی، از حالت مسدود به آماده یا از حالت مسدود و معلق به آماده و معلق تغییر وضعیت بدهد. در فرآیندهای مستقل، اگر فرآیندی در حالت مسدود یا مسدود و معلق باشد، زمانی به حالت آماده یا آماده و معلق تغییر وضعیت می‌دهد که کارهای ورودی و خروجی شخصی خودش تمام شود و نه بیرون رفتن یک فرآیند دیگر از سیستم!

تغییر حالت 10، امکان‌پذیر است، زیرا با خروج یک فرآیند، ممکن است، حافظه کافی برای پذیرش یک فرآیند جدید، ایجاد گردد.

بنابراین گزینه اول نادرست است، زیرا حالت‌های 4 و 7 امکان‌پذیر نیست.

تغییر حالت‌های 1 و 3 امکان‌پذیر نیست، زیرا با خروج یک فرآیند از سیستم و عدم حضور آن در سیستم، چگونه می‌تواند به حالت مسدود یا آماده تغییر وضعیت بدهد!

توجه داشته باشید که در هر لحظه فقط یک فرآیند می‌تواند در حالت اجرا (رونده) باشد. تغییر حالت 2 امکان‌پذیر است، زیرا پردازنده آزاد شده و یک فرآیند آماده باید جهت اجرا انتخاب و سپس اجرا گردد. بنابراین گزینه دوم نادرست است. زیرا حالت‌های 1 و 4 امکان‌پذیر نیست.

تغییر حالت‌های 5 و 8 امکان‌پذیر نیست، زیرا این تغییر وضعیت‌ها زمانی رخ می‌دهد که یک فرآیند برای تکمیل کارهای خود با کمبود حافظه مواجه شده باشد که سیستم عامل موظف است

برای تأمین این حافظه ابتدا فرآیندهای مسدود را راهی دیسک کند و به وضعیت مسدود و معلق ببرد و در غیر اینصورت فرآیندهای آماده را راهی دیسک کند و به وضعیت آماده و معلق ببرد. که در شرایط مسأله با خروج یک فرآیند از سیستم نه تنها با کمبود حافظه مواجه نشده‌ایم که رهاسازی حافظه نیز داشته‌ایم. بنابراین گزینه سوم نادرست است. زیرا حالت‌های 5 و 8 امکان‌پذیر نیست. تغییر حالت‌های 6 و 9 امکان‌پذیر است، زیرا با خروج فرآیندی از سیستم، بلوک‌هایی از حافظه آزاد می‌گردند و ممکن است بخش مدیریت حافظه تصمیم بگیرد که یک فرآیند موجود بر روی دیسک را که اولویت بیشتری از فرآیندهای موجود در صف آماده دارد را از حالت آماده و معلق به حالت آماده و یا از حالت مسدود و معلق به حالت مسدود منتقل کند. بنابراین گزینه چهارم درست است.

حال اگر در نگاهی دیگر، برای پاسخ به این سؤال، مانند طراح فکر نکنیم، و کامل‌تر فکر کنیم و علاوه بر فرآیندهای مستقل، مسأله ناحیه بحرانی فرآیندهای هم‌روند را در سیستم در نظر بگیریم، آنگاه گزینه‌های دوم و سوم نادرست و گزینه‌های اول و چهارم درست خواهند بود.

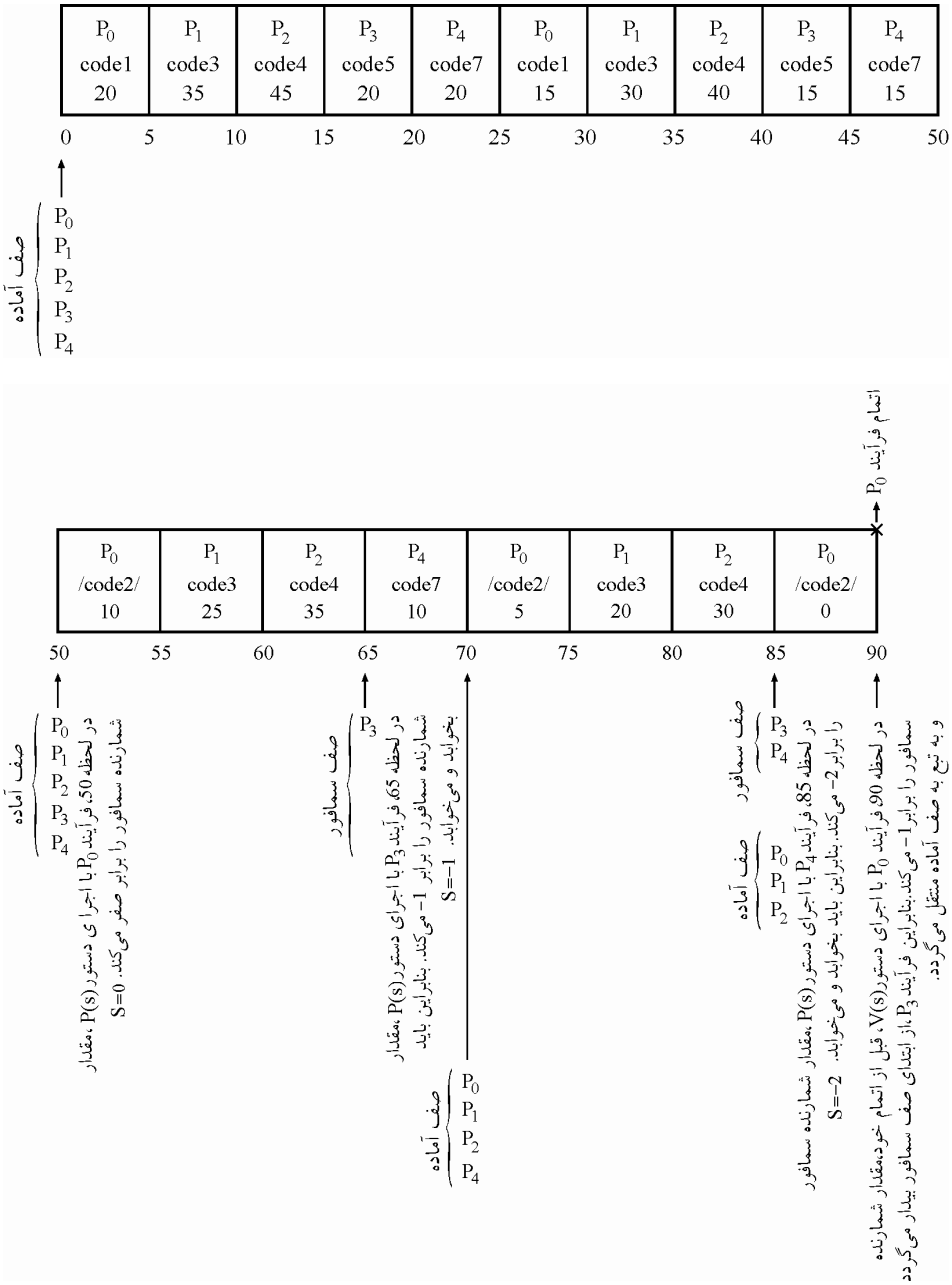
حالت‌های 1، 2، 5، 6، 8، 9 و 10 به مانند فرض قبل می‌باشد. اما حالت‌های 4 و 7 در این فرض کامل، امکان‌پذیر است، که طراح محترم در نظر نگرفته است. در مبحث ناحیه بحرانی فرآیندهای هم‌روند، هرگاه یک فرآیند، داخل ناحیه بحرانی باشد و فرآیند دیگری قصد ورود به ناحیه بحرانی را داشته باشد و در صورتی که مثلاً از راه حل سمافور استفاده گردد، فرآیند رقیب با ناحیه بحرانی پُر مواجه می‌گردد و برای اینکه فرآیند رقیب، مسأله انتظار مشغول را ایجاد نکند. راه حل سمافور فرآیند رقیب را مسدود می‌کند تا در صف آماده قرار نگیرد. همچنین ممکن است کمی بعدتر، همین فرآیند مسدود به دلیل کمبود حافظه به حالت مسدود و معلق درآید. حال اگر فرآیند موجود در ناحیه بحرانی از ناحیه بحرانی خارج گردد و قبل از خروج خود از سیستم، توسط عمل بیدارسازی، فرآیند رقیب خود را بیدار سازد که یا در حالت مسدود است و یا مسدود و معلق، آنگاه فرآیند رقیب می‌تواند از حالت مسدود به آماده و یا از حالت مسدود و معلق به آماده و معلق تغییر وضعیت بدهد.

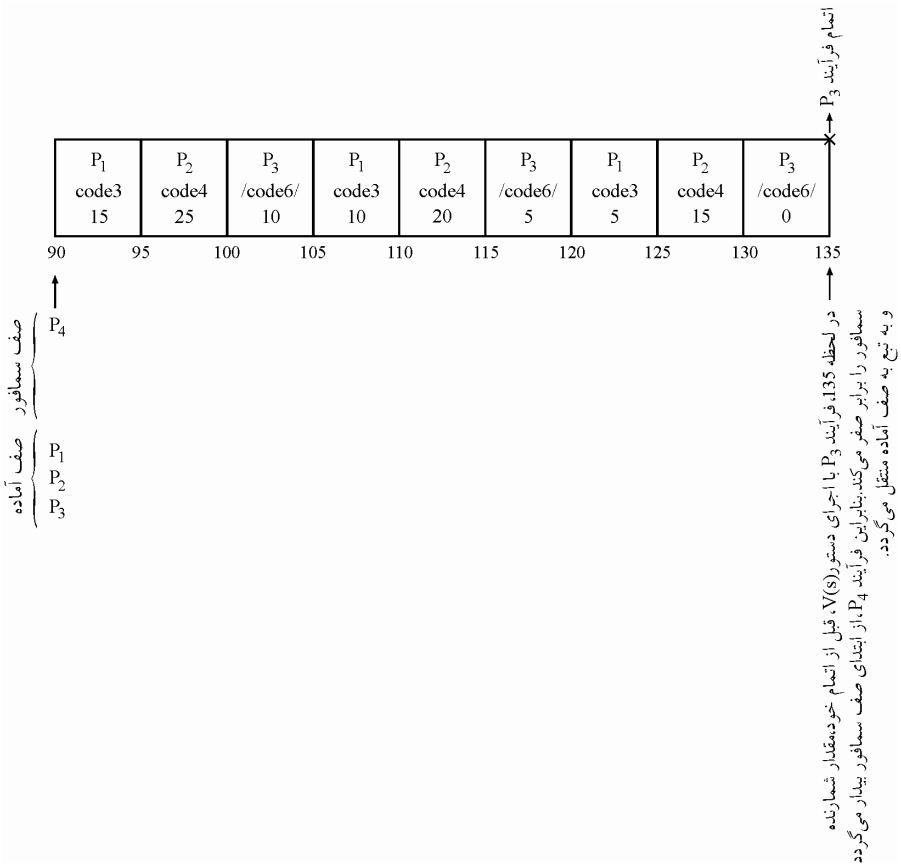
۲- گزینه ( ) صحیح است.

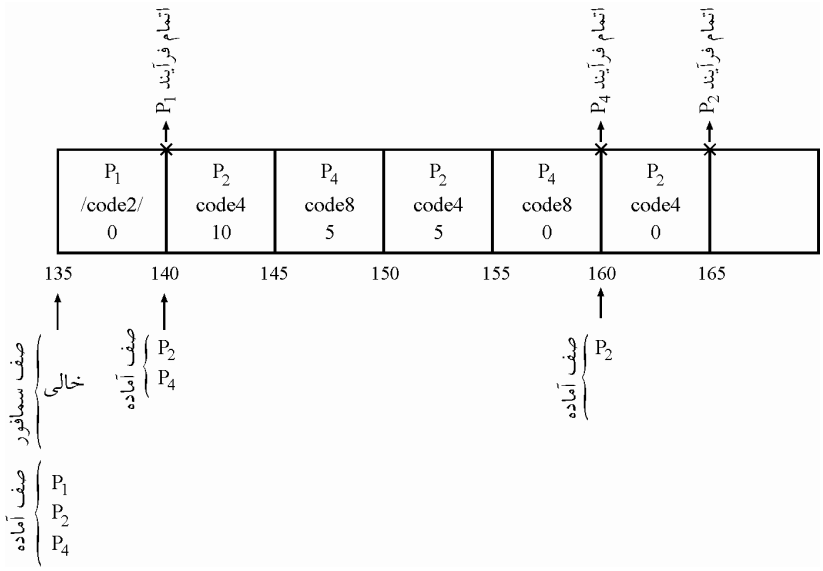
فرآیند	زمان ورود	قطعه کد	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_0	0	Code1 /Code2/	10 15		
P_1	0	Code3	40		
P_2	0	Code4	50		
P_3	0	Code5 /Code6/	10 15		
P_4	0	Code7 /Code8/	15 10		

راه حل اول:

اگر دستور P(s) ابتدای کدهای /Code2/ ، /Code6/ و/Code8/ اجرا گردد، داریم:

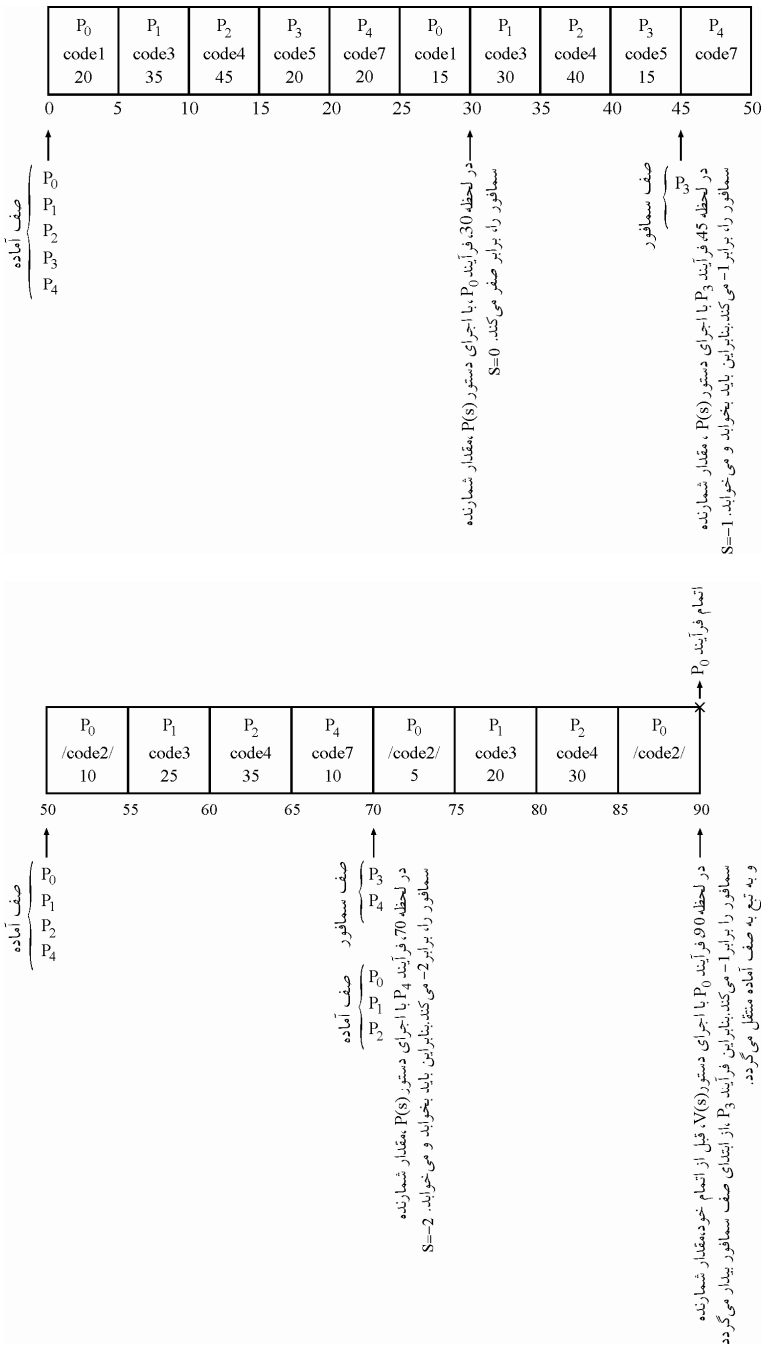


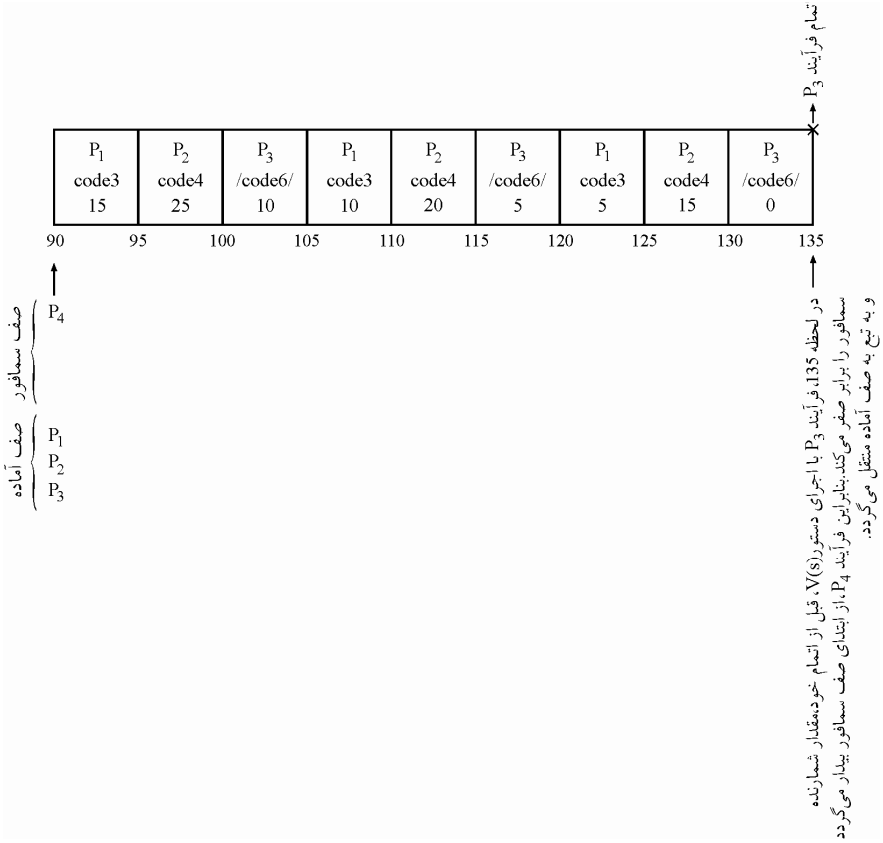

P1 code3 15
P2 code4 25
P3 /code6/ 10
P1 code3 10
P2 code4 20
P3 /code6/ 5
P1 code3 5
P2 code4 15
P3 /code6/ 0
90 95 100 105 110 115 120 125 130 135
اتمام فرآیند P3
در لحظه 135، فرآیند P3 با اجرای دستور V(s)، قبل از اتمام خود مقدار شمارنده سمافور را برابر صفر می‌کند. بنابراین فرآیند P4، از ابتدای صف سمافور بیدار می‌گردد و به تبع به صف آماده منتقل می‌گردد.
صف سمافور
P4
صف آماده
P1
P2
P3

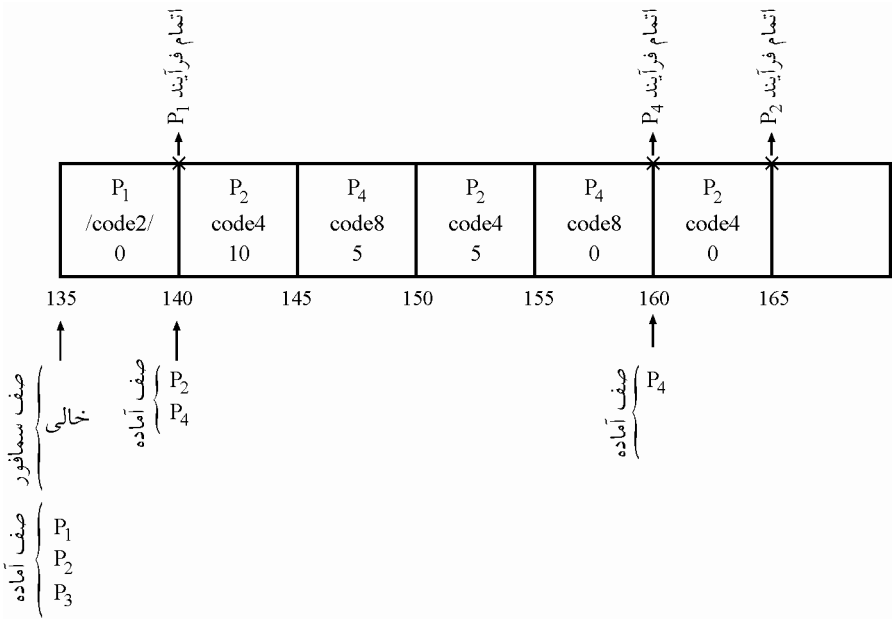

اتمام فرآیند P1
اتمام فرآیند P4
اتمام فرآیند P2
P1 /code2/ 0
P2 code4 10
P4 code8 5
P2 code4 5
P4 code8 0
P2 code4 0
135 140 145 150 155 160 165
صف سمافور
خالی
صف آماده
P1
P2
P4
صف آماده
P2
P4
صف آماده
P2

راه حل دوم:

اگر دستور P(s) انتهای کدهای Code1، Code5 و Code7 اجرا گردد، داریم:




P1 code3 15
P2 code4 25
P3 /code6/ 10
P1 code3 10
P2 code4 20
P3 /code6/ 5
P1 code3 5
P2 code4 15
P3 /code6/ 0
90 95 100 105 110 115 120 125 130 135
تمام فرآیند P3
در لحظه 135، فرآیند P3 با اجرای دستور V(s)، قبل از اتمام خود، مقدار شمارنده سمافور را برابر صفر می‌کند. بنابراین فرآیند P4، از ابتدای صف سمافور بیدار می‌گردد و به تبع به صف آماده منتقل می‌گردد.
صف سمافور
P4
صف آماده
P1
P2
P3


اتمام فرآیند P1
اتمام فرآیند P4
اتمام فرآیند P2
P1 /code2/ 0
P2 code4 10
P4 code8 5
P2 code4 5
P4 code8 0
P2 code4 0
135 140 145 150 155 160 165
صف سمافور
خالی
صف آماده
P1
P2
P3
صف آماده
P2
P4
صف آماده
P4

در هر دو صورت حل، داریم:

زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_0 زمان بازگشت = $90 - 0 = 90$

P_1 زمان بازگشت = $140 - 0 = 140$

P_2 زمان بازگشت = $165 - 0 = 165$

P_3 زمان بازگشت = $135 - 0 = 135$

P_4 زمان بازگشت = $160 - 0 = 160$

$$\text{میانگین زمان بازگشت} = \frac{90+140+165+135+160}{5} = \frac{690}{5} = 138$$

زمان اجرای فرآیند– زمان بازگشت فرآیند = زمان انتظار فرآیند

P_0 زمان انتظار = $90 - 25 = 65$

P_1 زمان انتظار = $140 - 40 = 100$

P_2 زمان انتظار = $165 - 50 = 115$

P_3 زمان انتظار = $135 - 25 = 110$

P_4 زمان انتظار = $160 - 25 = 135$

$$\text{میانگین زمان انتظار} = \frac{65+100+115+110+135}{5} = \frac{525}{5} = 105$$

$$\text{میانگین زمان اجرا} = \frac{25+40+50+25+25}{5} = \frac{165}{5} = 33$$

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

138=33+105

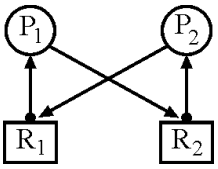
توجه: برای مجموعه فرآیندهای P_1 و P_2 بن بست رخ نمی‌دهد، زیرا فقط دستورات پردازش دارند و براساس الگوریتم نوبت چرخشی، در هر تکرار پردازنده را در اختیار می‌گیرند، تا به اتمام برسند.

توجه: برای مجموعه فرآیندهای P_0، P_3 و P_4 نیز هیچگاه بن بست رخ نمی‌دهد، برای این مجموعه فرآیندها زمانی بن بست رخ می‌دهد که هر سه فرآیند بخوابند و دیگر کسی نباشد که آنها را بیدار کند، یعنی به خواب ابدی بروند، اما مطابق آنچه در اجرای فرآیندها دیدیم، همه فرآیندها پس از اجرای کامل، از سیستم خارج شدند و این به معنی عدم وقوع بن بست در سیستم است، بنابراین گزینه چهارم نادرست است.

توجه: مطابق مقدار میانگین زمان اجرای برابر با 33، باید تفاضل میانگین زمان بازگشت و میانگین زمان انتظار برابر میانگین زمان اجرا و برابر مقدار 33 باشد که فقط در گزینه اول و دوم این مسأله رعایت شده است، بنابراین گزینه سوم نادرست است.

توجه: پاسخ سؤال، در هیچ یک از گزینه‌های باقی مانده، نمی‌باشد.
توجه: سازمان سنجش آموزش کشور، در کلید اولیه و نهایی خود، گزینه اول را به عنوان پاسخ اعلام کرده بود.
سخنی با مسئولین سازمان سنجش آموزش کشور و طراحان محترم: برای ارج نهادن به تلاش‌های با ارزش داوطلبان گرامی، بهتر است مسئولین محترم سازمان سنجش آموزش کشور و طراحان محترم، دقت بیشتری را در طرح سؤال داشته باشند. امید است، این مسائل به زودی مرتفع گردد.

۳- گزینه (۱) صحیح است.
مهمترین خاصیت مانیتور آن است که هیچ دو فرآیندی بطور همزمان نمی‌توانند در یک مانیتور فعال باشند، این یعنی برقراری انحصار متقابل و غیر اشتراکی شدن منبع.
در یک سناریو فرض کنید، فرآیند P_1 وارد مانیتور خود به نام R_1 می‌شود و فرآیند P_2 نیز وارد مانیتور خود به نام R_2 می‌شود. اگر در این لحظه فرآیند P_1 خواهان مانیتور R_2 (منبع غیر اشتراکی R_2) باشد، چون مانیتور R_2 در اختیار فرآیند P_2 می‌باشد، فرآیند P_1 در صف این مانیتور می‌خوابد. حال اگر فرآیند P_2 هم خواهان مانیتور R_1 (منبع غیر اشتراکی R_1) باشد، چون مانیتور R_1 در اختیار فرآیند P_1 می‌باشد، فرآیند P_2 هم در صف این مانیتور می‌خوابد.
در این شرایط نمودار زیر را داریم:



مطابق نمودار فوق، واضح است که هر چهار شرط کافمن برقرار است.
شرط انحصار متقابل و اشتراکی نبودن منابع توسط مانیتور برقرار شده است. شرط انحصاری بودن هم بدیهی و برقرار است. نمی‌توان مانیتورها را به زور پس گرفت. شرط نگهداری و انتظار هم مطابق سناریوی فوق می‌تواند برقرار باشد. یعنی در عین حال اینکه یک منبع را فرآیندها در اختیار دارند، منبع دوم را نیز درخواست کرده‌اند، این یعنی نگهداری و انتظار. شرط سیکل یا انتظار چرخشی هم که مطابق نمودار، واضح است که برقرار است. بنابراین با توجه به سناریوی فوق وقوع بن بست حتمی است.

۴- گزینه (۲) صحیح است.
به صورت سؤال توجه کنید.

$S \rightarrow$ Value =1 (مقداردهی اولیه)

توابع wait و signal تعریف شده در صورت سؤال، مطابق تعریف سمافور عمومی (منفی شونده) می باشد. 5 فرآیند P_1، P_2، P_3، P_4 و P_5 را در نظر بگیرید. سناریوی زیر را در نظر بگیرید:

ابتدا P_1 درخواست ورود می‌دهد و از آنجایی که مقدار اولیه شمارنده سمافور 1 است، به آن مجوز ورود داده می‌شود. در زمانی که فرآیند P_1 مشغول اجرای ناحیه بحرانی است، فرآیند P_2 درخواست استفاده از ناحیه بحرانی را دارد که چون با عملیات wait، مقدار شمارنده سمافورs، 1- می‌شود، این فرآیند به صف سمافور اضافه شده و بلوکه می‌گردد (چون تنها فرآیند موجود در صف سمافور، P_2 است، این فرآیند در ابتدای صف سمافور قرار می‌گیرد).

پس از مدتی فرآیند P_3 نیز درخواست ناحیه بحرانی را صادر می‌کند که با اجرای دستور wait، مقدار شمارنده سمافور 2- شده و این فرآیند نیز به لیست فرآیندهای بلوکه شده اضافه می‌گردد. از آنجایی که در دستور $\text{block}(-(s \rightarrow \text{value}))$ (-2 / +2) موجود در تابع wait، فرآیند P_3 رتبه 2 را دارد، این فرآیند بعد از فرآیند P_2 قرار خواهد گرفت. $(\overrightarrow{P_3, P_2})$

پس از مدتی فرآیند P_4 نیز درخواست ورود به ناحیه بحرانی را می‌دهد که مطابق آنچه در مورد فرآیندهای قبلی گفته شد مقدار شمارنده سمافور 3- شده و فرآیند P_4 در رتبه 3+ قرار می‌گیرد. $(\overrightarrow{P_4, P_3, P_2})$ حال فرض کنید کار فرآیند P_1 با ناحیه بحرانی به اتمام برسد. در اینصورت مقدار سمافور 2- شده و فرآیند ابتدای لیست (یعنی P_2) بیدار و برای اجرا انتخاب می‌شود. پس از مدتی کار فرآیند P_2 نیز با ناحیه بحرانی به اتمام رسیده و مقدار سمافور 1- شده و فرآیند بعدی از ابتدای لیست (یعنی P_3) بیدار و برای اجرا انتخاب می‌شود.

در حال حاضر فقط P_3 در حال اجرا است و فرآیند P_4 (با رتبه 3) در صف انتظار قرار دارد.

حال اگر در این لحظه فرآیند جدیدی به نام P_5 درخواست ورود به ناحیه بحرانی را دهد، با اجرای دستور wait، مقدار سمافور را 2- کرده و به صف سمافور اضافه می‌شود. از آنجایی که با اجرای $\text{block}(-(s \not\rightarrow \text{value}))$ (-2 / +2) فرآیند P_5 رتبه 2+ را دارد، جلوتر از فرآیند P_4 (که رتبه 3 را داشت) قرار می‌گیرد و اولویت اجرایش بالاتر از P_4 می‌شود. $(\overrightarrow{P_4, P_5})$ سناریوی فوق می‌تواند بارها و بارها تکرار شده و تخصیص ناحیه بحرانی به فرآیندهای جدید مانند P_6، P_7 و غیره، با رتبه بالاتری نسبت به P_4 انجام شود.

مثلاً در ادامه، کار فرآیند P_3 با ناحیه بحرانی تمام می‌شود و مقدار سمافور 1- شده و فرآیند بعدی از ابتدای لیست (یعنی P_5) بیدار و برای اجرا انتخاب می‌شود. (درحالی که دیرتر از P_4 آمده بود اما به دلیل اولویت‌بندی، زودتر اجرا می‌شود) حال اگر در این لحظه فرآیند جدیدی بنام P_6

درخواست ورود به ناحیه بحرانی را بدهد، با اجرای دستور wait، مقدار سمافور را 2- کرده و به صف سمافور اضافه می‌شود. از آنجاییکه با اجرای block (-(s ← value)) ، فرآیند P_6 رتبه 2 را دارد، مجدداً جلوتر از فرآیند P_4 (که رتبه 3 را دارد) قرار می‌گیرد و اولویت اجرایش بالاتر از P_4 می‌شود. $(\overrightarrow{P_4, P_6})$

این سناریو می‌تواند بارها و بارها تکرار شود.

این مشکل سبب بروز گرسنگی (starvation) یعنی عدم برقراری شرط انتظار محدود خواهد شد. برقراری شرط انحصار متقابل و پیشرفت در سمافور بدیهی است.

توجه: اگر روش نوبت‌دهی صف سمافور، مطابق سمافور استاندارد، روش FIFO می‌بود، این راه حل درست بود و تمامی شرایط انحصار متقابل، پیشرفت و انتظار محدود برقرار می‌شد.

توجه: در سمافورها صفی برای نگهداری فرآیندهای منتظر استفاده می‌شود. اگر آزاد شدن یا خروج از این صف به ترتیب ورود (FIFO) باشد، به آن سمافور **قوی** می‌گویند، و اگر آزاد شدن یا خروج از این صف به ترتیب ورود نباشد، به آن سمافور **ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را گارانتی می‌کنند، اما در سمافورهای ضعیف امکان بروز پدیده گرسنگی وجود دارد.

توجه: نکته اصلی این سؤال وجود تابع block(n) در تابع wait است که فرآیند مسدود شده را به ترتیب از کوچک به بزرگ مرتب می‌کند. اگر این مرتب‌سازی صورت نمی‌گرفت فرآیندها براساس الگوریتم FIFO وارد ناحیه بحرانی می‌شدند، اما بر اثر این مرتب‌سازی فرآیندی که شماره بزرگتری به آن داده می‌شود، به انتهای صف می‌رود و ممکن است دچار گرسنگی شود و باعث شود شرط انتظار محدود برقرار نباشد.

۵- گزینه (۴) صحیح است.

با وجود آنکه صورت تست طولانی است، اما مسأله مطرح شده بسیار ساده است. با توجه به صورت سوال، با اجرای T_1 تایمر با یک مقدار بالا مقدار دهی می‌شود و بعد از اتمام زمان آن، سیستم بطور کامل بار می‌گردد. در سناریوی ارائه شده در مرحله‌ی 3، تایمر T_1 مقداردهی و شروع به شمارش معکوس می‌کند. در مرحله‌ی 4، T_1 به دلیل حضور T_3 در ناحیه بحرانی مسدود می‌گردد. در مرحله 5، T_2 به دلیل داشتن اولویت بالاتر نسبت به T_3، T_3 را قبضه می‌کند، بنابراین پردازنده در اختیار T_2 قرار می‌گیرد. در مرحله 6، T_2 به دلیلی مستقل از T_1و T_3 معلق می‌شود، بدین معنی که پردازنده را واگذار می‌کند، حالا به هر دلیلی (مثل عملیات ورودی و خروجی و قرار گرفتن در وضعیت منتظر). در حال حاضر پردازنده آزاد است و فقط T_3 در صف آماده پردازنده قرار دارد، زیرا T_1 مسدود و در صف سمافور قرار دارد و T_2 هم معلق است، بنابراین

پردازنده در اختیار T_3 قرار داده می‌شود. بنابراین T_3 می‌تواند ادامه کار خود در ناحیه بحرانی را انجام و تمام کند و T_1 را از ابتدای صف سمافور بیدار کند تا در صف آماده پردازنده قرار گیرد. حال اگر دقیقا در همین لحظه هم تعلیق T_2 تمام شود و T_2 هم در صف آماده پردازنده قرار بگیرد، از آنجا که T_1 اولویت بیشتری نسبت به T_2 دارد، T_1 می‌تواند پردازنده را در اختیار بگیرد و وارد ناحیه بحرانی گردد. و قبل از Timeout تمام گردد و T_2 هم پس از مدتی تمام گردد. در واقع اگر همه‌ی شرایط فوق به موقع رخ دهد، سیستم به درستی کار می‌کند و کارهای T_1، T_2 و T_3 تمام می‌شوند و سناریوی مطرح شده می‌تواند به طور دوره‌ای در این نرم افزار تعبیه شده تکرار گردد.

اما نکته در اینجاست که اگر تعلیق یاد شده در مرحله 6، به موقع رخ ندهد و دیر اتفاق بیافتد، باعث می‌شود پردازنده در اختیار T_2 باقی بماند، بنابراین ادامه کار T_3 در ناحیه بحرانی به تعویق می‌افتد. بنابراین صدور مجوز ورود T_1 به ناحیه بحرانی به تعویق می‌افتد. زیرا صدور این مجوز مستلزم اتمام کار T_3 در ناحیه بحرانی است که در حال حاضر پردازنده را در اختیار ندارد و درگیر تاخیر ناشی از کار طولانی T_2 است، در این شرایط زمان در حال سپری شدن است، در حالیکه از مدت‌ها قبل شمارنده تایمر T_1، شمارش معکوس خود را آغاز کرده است. زمان می‌گذرد ولی همچنان T_2 معلق نشده است، زمان به سرعت در حال سپری شدن است، T_2 آنقدر تعلل و تأخیر ایجاد می‌کند که تا سرانجام شمارنده T_1، Timeout کند. در نتیجه سیستم مجدداً راه‌اندازی می‌گردد. مطابق فرض مسأله منقضی‌شدن تایمر (Timeout)، نشانه وجود مشکل در اجرای نرم‌افزار فضاپیما است که باید تمام پردازش‌ها متوقف شوند و نرم‌افزار به طور کامل بار شود و تمام سیستم‌ها آزمایش شوند و همه چیز از نقطه شروع آغاز گردد. بنابراین از آنجا که احتمال وقوع Timeout مطابق سناریوی فوق در این سیستم وجود دارد، بنابراین ممکن است این سیستم به درستی کار نکند، بنابراین گزینه دوم نادرست و گزینه چهارم درست است.

گزینه اول نیز نادرست است، زیرا با افزایش اولویت T_3 نسبت به T_2، باز هم این سیستم ممکن است به درستی کار نکند، ممکن است کار T_3 در ناحیه بحرانی آنقدر طولانی گردد که باز هم تایمر منقضی گردد.

گزینه سوم نیز نادرست است، زیرا وجود سهم زمانی برای خروج سریع یک کار موجود در ناحیه بحرانی کاری نمی‌تواند بکند، یک کار حاضر در ناحیه بحرانی، خودش باید از ناحیه بحرانی خارج گردد. برش زمانی فقط سبب انتقال پردازنده به کارها می‌گردد، حال اگر مدت زمان فعالیت یک کار در ناحیه بحرانی زیاد باشد، به تبع تعداد سهم زمانی بیشتری را مصرف می‌کند، که باز هم منجر به ایجاد تأخیر می‌گردد.

توجه: مشکل اساسی در این سیستم این است که کاری روی شمارنده تایمر نقش دارد که خود در شرایط رقابتی ممکن است در مدت زمانی که تایمر، شمارش معکوس خود را آغاز کرده است، وارد ناحیه بحرانی نشود.

۶- گزینه (۱) صحیح است.

بهترین راه کشف بن‌بست، موازی جلو بردن خطوط کدها است.

توجه: فرض می‌شود مقدار شمارنده سمافور برابر یک است.

Fig A:

با اجرای سناریوی زیر، کد Fig A می‌تواند منجر به بن بست شود:

فرض کنید ابتدا خط اول از هر کدام از Threadهای A، B و C اجرا شود. در این صورت مقدار شمارنده‌ی سمافورهای Lock1، Lock2 و Lock3 برابر صفر می‌شود. حال فرض کنید نوبت اجرای خط دوم از هر کدام از Thread های A، B و C برسد. در این صورت Thread A روی سمافور Lock2 با مقدار 1-، Thread B روی سمافور Lock3 با مقدار 1- و Thread C نیز روی سمافور Lock2 با مقدار 2- به خواب می‌رود. پس در این حالت از اجرا، سیستم دچار بن بست می‌شود.

Fig B:

با اجرای سناریوی زیر Fig B می‌تواند منجر به بن‌بست شود:

فرض کنید ابتدا خط اول از هر کدام از Thread های A، B و C اجرا شوند. در این صورت مقدار شمارنده‌ی سمافورهای Lock1 به ازای دو بار اجرا در Thread A و Thread B برابر 1- و Lock2 برابر صفر می‌شود.

توجه: Thread B دیگر جلو نمی‌رود، زیرا در صف سمافور Lock1 با مقدار 1- خوابیده است.

حال فرض کنید نوبت اجرای خط دوم از هر کدام از Threadهای A و C برسد. در این صورت مقدار شمارنده‌ی سمافورهای Lock4 و Lock3 برابر صفر می‌شود. حال در ادامه فرض کنید نوبت اجرای خط سوم از هر کدام ازThreadهای A و C برسد. Thread A روی سمافور Lock2 با مقدار 1- و Thread C روی سمافور Lock4 با مقدار 1- به خواب می‌رود. Thread B نیز که قبلاً به خواب رفته است. پس در این حالت از اجرا، سیستم دچار بن بست می‌شود.

Fig C:

کد Fig C بن بست ندارد، بنابراین راه‌حل درستی است.

فرض کنید ابتدا خط اول از هر کدام از Thread های A، B و C اجرا شوند، در این صورت مقدار شمارنده سمافورهای Lock1 به ازای دو بار اجرا در Thread A و Thread B برابر 1- و Lock3 برابر صفر می‌شود.

توجه: Thread B دیگر جلو نمی‌رود، زیرا در صف سمافور Lock1 با مقدار 1- خوابیده است.

حال فرض کنید نوبت اجرای خط دوم از هر کدام از Threadهای A و C برسد. در این صورت مقدار شمارنده سمافور Lock2 به ازای دو بار اجرا در Thread A و Thread C برابر 1- می‌شود.

توجه: Thread C دیگر جلو نمی‌رود، زیرا در صف سمافورLock2 با مقدار 1- خوابیده است، حال در ادامه نوبت اجرای خط سوم از Thread A می‌رسد. در این صورت مقدار شمارنده سمافور Lock4 برابر صفر می‌شود و در ادامه Thread A وارد ناحیه بحرانی می‌گردد. پس از خروج Thread A از ناحیه بحرانی، به تریتب مقدار شمارنده .سمافور Lock4 برابر 1، Lock2 برابر صفر و Lock1 برابر صفر می‌شود. بنابراین Thread B و Thread C از خواب بیدار می‌شوند و در صف آماده‌ی پردازنده قرار می‌گیرند. حال در ادامه، خط دوم از Thread B اجرا می‌گردد، در این صورت مقدار شمارنده سمافور Lock2 برابر 1- می‌شود.

توجه: Thread B دیگر جلو نمی‌رود، زیرا در صف سمافور Lock2 با مقدار 1- خوابیده است. حال در ادامه نوبت اجرای خط سوم از Thread C می‌رسد، در این صورت مقدار شمارنده سمافور Lock4 برابر صفر می‌شود و در ادامه Thread C وارد ناحیه بحرانی می‌گردد. پس از خروج Thread C از ناحیه بحرانی، به ترتیب مقدار شمارنده سمافور Lock4 برابر 1، Lock2 برابر صفر و Lock3 برابر 1 می‌شود. بنابراین Thread B از خواب بیدار می‌شود و در صف آماده پردازنده قرار می‌گیرد. حال در ادامه خط سوم از Thread B اجرا می‌گردد، در این صورت مقدار شمارنده سمافور Lock3 برابر صفر می‌شود و در ادامه Thread B نیز وارد ناحیه بحرانی می‌گردد. پس از خروج Thread B از ناحیه بحرانی، به ترتیب مقدار شمارنده سمافور Lock3 برابر 1، Lock2 برابر 1 و Lock1 نیز برابر 1 می‌شود.

:Fig D

کد Fig D شرط انحصار متقابل را رعایت نمی‌کند. زیرا UnLock در Threadها قبل از ناحیه بحرانی قرار داده شده است. در حالیکه در راه‌حل سمافور، کسب اجازه برای ورود به ناحیه بحرانی باید با Lock(wait) باشد و اعلام خروج از ناحیه بحرانی با UnLock(signal) باشد.

توجه: سازمان سنجش آموزش کشور، گزینه اول را به عنوان پاسخ نهایی اعلام کرده بود.

۷- گزینه (۳) صحیح است.

شرایط رقابتی (مسابقه)

هرگاه دو یا چند فرآیند همزمان با هم وارد ناحیه‌ی بحرانی (منبع مشترک) شوند، شرایط رقابتی پیش می‌آید. در شرایط رقابتی، نتیجه‌ی نهایی بستگی به ترتیب دسترسی‌ها دارد. در واقع فرآیندهای همکار بر هم اثر دارند و اینکه پردازنده، به چه ترتیبی و در چه زمان‌هایی بین آنها تعویض متن انجام دهد در ایجاد پاسخ نهایی اثرگذار خواهد بود. بنابراین علت شرایط رقابت **تعویض متن پردازنده** بین فرآیندهای همکار است.

برای کنترل شرایط رقابتی، باید راه حلی ارائه شود که سه شرط زیر را به عنوان **معیارهای اخلاقی** در رقابت، رعایت کند:

۱- شرط انحصار متقابل

برای برقرای شرط انحصارمتقابل، عامل مشترک را اسکورت کنید، مانند زمانی که وارد باجه‌ی تلفن همگانی (عامل مشترک) می‌شوید، در را می‌بندید تا مانع ورود شخص دیگری گردید! در عالم انسان‌ها، هیچ دو فردی نباید به طور همزمان وارد عامل مشترک شوند. در عالم فرآیندها نیز هیچ دو فرآیندی نباید به طور همزمان وارد عامل مشترک (ناحیه بحرانی) شوند. استفاده‌ی همزمان از عامل مشترک معنا ندارد! (اخلاقی نیست) بنابراین باید راهی پیدا کنیم که از ورود همزمان دو یا چند فرآیند به ناحیه‌ی بحرانی جلوگیری کند. به عبارت دیگر، آنچه که ما به آن نیاز داریم، انحصار متقابل است که در متون فارسی به آن دو به دو ناسازگاری یا مانعه الجمعی نیز گفته می‌شود، یعنی اگر یکی از فرآیندها در حال استفاده از حافظه‌ی اشتراکی، فایل اشتراکی و یا هر عامل اشتراکی رقابت‌زاست باید مطمئن باشیم که دیگر فرآیندها، در آن زمان از انجام همان کار محروم می‌باشند. در واقع از بین تمام فرآیندها، در هر لحظه تنها یک فرآیند مجاز است، در عامل مشترک باشد. بدین معنی که اگر فرآیندی در ناحیه‌ی بحرانی است، از ورود فرآیندهای دیگر به همان ناحیه‌ی بحرانی جلوگیری شود و تا خارج شدن فرآیند اول منتظر بمانند، زیرا هیچ دو فرآیندی نباید به طور همزمان وارد ناحیه‌ی بحرانی شوند. به یاد داشته باشید که استفاده‌ی همزمان از عامل مشترک معنا ندارد!

بنابراین برای برقراری شرط انحصار متقابل باید ساختاری را طراحی کنیم که در هر لحظه فقط یک فرآیند مجوز ورود به ناحیه‌ی بحرانی را داشته باشد. لذا هر فرآیند برای ورود به بخش بحرانی‌اش باید اجازه بگیرد. بخشی از کد فرآیند که این اجازه گرفتن را پیاده‌سازی می‌کند، بخش ورودی نام دارد. بخش بحرانی می‌تواند با بخش خروجی دنبال شود. این بخش خروجی کاری می‌کند که فرآیندهای دیگر بتوانند وارد ناحیه‌ی بحرانی‌شان، شوند. بقیه‌ی کد فرآیند را بخش باقی‌مانده می‌نامند. بنابراین ساختار کلی فرآیندها برای برقراری شرط انحصار متقابل به صورت زیر می‌باشد:

```
P (int i)  {
while ( TRUE)  {
entry_section () ;  // تلاش برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی
critical_section (); // ناحیه‌ی بحرانی
exit_section () ; // اعلام خروج از ناحیه‌ی بحرانی
remainder_section () ; // ناحیه‌ی باقی مانده
   }
}
```

توجه: بدترین شرایط وقتی است که یک فرآیند بخواهد بارها و بارها وارد ناحیه بحرانی خود شود، برای اینکه سخت‌ترین شرایط بررسی شود، ناحیه بحرانی را داخل حلقه بی‌نهایت قرار می‌دهیم.

۲- شرط پیشرفت

فرآیندی که داوطلب ورود به ناحیه‌ی بحرانی نیست و نیز در ناحیه‌ی بحرانی قرار ندارد، نباید در رقابت برای ورود سایر فرآیندها به ناحیه‌ی بحرانی شرکت کند، به عبارت دیگر، نباید مانع ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شود. در یک بیان ساده‌تر می‌توان گفت، فرآیندی که در ناحیه-ی باقی‌مانده قرار دارد، حق جلوگیری از ورود فرآیندهای دیگر به ناحیه‌ی بحرانی را ندارد، یعنی نباید در تصمیم‌گیری برای ورود فرآیندها به ناحیه‌ی بحرانی شرکت کند.

۳- شرط انتظار محدود

فرآیندهایی که نیاز به ورود به ناحیه‌ی بحرانی دارند، باید مدت انتظارشان محدود باشد، یعنی نباید به طور نامحدود در حالت انتظار باقی بمانند.

انتظار نامحدود به دو دسته می‌باشد: (۱) قحطی، (۲) بن‌بست، بنابراین نباید در شرایط رقابتی بین فرآیندها، قحطی یا بن‌بست رخ دهد. برای اینکه شرط انتظار محدود برقرار باشد، باید هم قحطی و هم بن‌بست رخ ندهد.

قحطی (گرسنگی)

در عالم زندگی قحطی زمانی رخ می‌دهد که عده‌ای مدام از منابع مشترک استفاده کنند، و عده‌ای دیگر قادر به استفاده از منابع مشترک نباشند. زیرا دسته‌ی اول از اختصاص منابع به دسته‌ی دوم به طور مداوم و بدون رعایت یک حد بالای مشخص جلوگیری می‌کنند. در عالم فرآیندها نیز هرگاه فرآیندی به مدت نامعلوم و بدون رعایت یک حد بالای مشخص در انتظار گرفتن یک منبع بحرانی یا دسترسی به یک عامل مشترک بماند و فرآیندی دیگر مدام در حال استفاده از منبع بحرانی باشد، در این حالت فرآیند اول دچار قحطی شده است. بنابراین در صورت اقدام یک فرآیند برای ورود به ناحیه‌ی بحرانی، باید محدودیتی برای تعداد دفعاتی که سایر فرآیندها می‌توانند وارد ناحیه‌ی بحرانی شوند، وجود داشته باشد تا قحطی رخ ندهد.

بن‌بست

به وضعیتی که در آن مجموعه‌ای متشکل از دو یا چند فرآیند برای همیشه منتظر یکدیگر بمانند (مسدود) و به عبارت دیگر دچار سیکل انتظار ابدی شوند، بن‌بست گفته می‌شود.

توجه: به تفاوت قحطی و بن‌بست توجه کنید، در قحطی فرآیندی مدام در حال کار و فرآیندی دیگر به مدت نامعلوم در انتظار است. اما در بن‌بست، مجموعه‌ای از فرآیندها در سیکل انتظار ابدی، گرفتار شده‌اند. نه راه پس دارند و نه راه پیش.

توجه: در کنترل شرایط رقابتی، رعایت شرط انحصار متقابل، شرط لازم و رعایت شروط پیشروی و انتظار محدود، شروط کافی برای ارائه‌ی یک راه‌حل **جامع** و **اخلاقی** به شمار می‌آیند.

ابتدا کد مطرح شده در صورت سوال را برای دو فرآیند P_1 و P_2 به صورت زیر بازنویسی می‌کنیم:

```
P1:
While (TRUE){
① key[1]= TRUE;
② while (key[1]) swap (key[1],lock);
/*critical section*/
③ lock=FALSE;
/*remainder_section*/
}
```

```
P2:
While (TRUE) {
① key[2]= TRUE;
② while (key[2]) swap (key[2],lock);
/*critical section*/
③ lock=FALSE;
/*remainder_section*/
}
```

توجه: مقادیر اولیه بر اساس صورت سوال به صورت زیر است:

lock = FALSE, Key[1] = FALSE, Key[2] = FALSE

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

برای کنترل برقراری شرط انحصار متقابل، شرط پیشرفت و شرط انتظار محدود (گرسنگی و بن‌بست) از آزمون‌های زیر استفاده می‌کنیم:

توجه: ما نام این آزمون‌ها را به عنوان مبدع آن «**قوانین ارسطو**» نام‌گذاری کردیم، این قوانین به «**قوانین چهارگانه ارسطو**» نیز موسوم است.

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_1 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① key[1]=TRUE;
② while (key[1])  swap (key[1],lock);
```

توجه: هم اکنون key[1] =TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جابه جا می‌شوند، به صورت زیر:

```
swap (key[1],lock);
```

توجه: پس از جا به جایی key[1] = FALSE و lock = TRUE است.

در ادامه حرکت شرط حلقه FALSE است، پس دستور swap اجرا نمی‌گردد، و کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

P1:

/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون اول وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P1 مشغول حرکت است.

(گام ۲): فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، یعنی:

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P2 بدهید.

فرض کنید فرآیند P2 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P2:

① key[2]= TRUE;

② while (key[2]) swap (key[2],lock);

توجه: هم اکنون key[2] =TRUE و lock = TRUE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[2] و lock جابه جا می‌شوند، به صورت زیر:

swap (key[2],lock);

توجه: پس از جا به جایی key[2] = TRUE و lock = TRUE است.

در ادامه حرکت، شرط حلقه TRUE است، پس دستور swap مجددا اجرا می‌گردد، و کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P2 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P2 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، خب موفق نشد. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط اول انحصار متقابل برقرار است.

فرم ساده قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، **(گام ۲)** سپس یه آدم دیگه رو هم جور کن که بخواد وارد باجه تلفن همگانی بشه، اگه اونم تونست وارد باجه تلفن همگانی بشه اونوقت شرط اول انحصار متقابل نقض شده. اخلاق می‌گه اگه یه نفر داخل باجه تلفن همگانی

هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگه بشه شرط اول انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند آنگاه در این حالت شرط انحصار متقابل نقض شده است.

```
P1:
While (TRUE){
① key[1]= TRUE;
② while (key[1]) swap (key[1],lock);
/*critical section*/
③ lock=FALSE;
/*remainder_section*/
}
```

```
P2:
While (TRUE) {
① key[2]= TRUE;
② while (key[2]) swap (key[2],lock);
/*critical section*/
③ lock=FALSE;
/*remainder_section*/
}
```

توجه: مقادیر اولیه براساس صورت سؤال به صورت زیر است:

lock = FALSE, Key[1] = FALSE, Key[2] = FALSE

فرض کنید فرآیند P_1 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① key[1]=TRUE;
```

همچنین فرض کنید فرآیند P_2 نیز به شکل همروند یا موازی با فرآیند P_1 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P2:
① key[2]=TRUE;
```

اگر فرض شود که دستور Swap به صورت اتمیک و تجزیه‌ناپذیر اجرا شود، آنگاه شرط انحصار متقابل برقرار است.

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین آرایه فرآیندها هر دو key[1] = TRUE و key[2] = TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند.

دستور swap جای مقادیر را جا به جا می‌کند. مقدار اولیه متغیر lock برابر FALSE است، هر فرآیندی که زودتر دستور swap را اجرا کند، داخل ناحیه بحرانی می‌شود و فرآیندی که دیرتر دستور swap را اجرا کند، باید صبر پیشه کند و در حلقه انتظار بچرخد.

فرض کنید، فرآیند P_1 **زودتر** و فرآیند P_2 **دیرتر** اقدام به اجرای دستور swap کنند، بنابراین فرآیند P_1 وارد ناحیه بحرانی می‌شود، اما فرآیند P_2 باید در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P1:
② while (key[1]) swap (key[1],lock);

توجه: هم اکنون key[1] = TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جا به جا می‌شوند، به صورت زیر:

swap (key[1],lock);

توجه: پس از جا به جایی key[1] = FALSE و lock = TRUE است.

در ادامه حرکت شرط حلقه FALSE است، پس دستور swap اجرا نمی‌گردد، و کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

P1:
/*critical_section*/

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P2 بدهید.

P2:
② while (key[2]) swap (key[2],lock);

توجه: هم اکنون key[2] = TRUE و lock = TRUE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جا به جا می‌شوند، به صورت زیر:

swap (key[1],lock);

توجه: پس از جا به جایی key[2] = TRUE و lock = TRUE است.

در ادامه حرکت شرط حلقه TRUE است، پس دستور swap مجددا اجرا می‌گردد، و کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P2 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P2 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.
همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفق نشدند. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط دوم انحصار متقابل نیز برقرار است.

فرم ساده قانون دوم ارسطو (آزمون دوم شرط انحصار متقابل)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هر دو تونستن به طور همزمان وارد باجه تلفن همگانی بشنن اونوقت شرط دوم انحصار متقابل نقض شده. اخلاق می‌گه دو نفر نباید همزمان باهم داخل باجه تلفن همگانی باشن، یعنی اگه یه نفر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگه بشه شرط دوم انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

توجه: برای برقرار بودن شرط انحصار متقابل باید **قانون اول ارسطو** (آزمون اول شرط انحصار متقابل) و **قانون دوم ارسطو** (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشند. بنابراین شرط انحصار متقابل در سوال مطرح شده برقرار است.

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۳)** در ادامه فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، **(گام ۴)** سپس همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۵)** در نهایت همان فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_1 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① key[1]=TRUE;
② while (key[1])  swap (key[1],lock);
```

توجه: هم اکنون key[1] =TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جابه جا می‌شوند، به صورت زیر:

```
swap (key[1],lock);
```

توجه: پس از جا به جایی key[1] = FALSE و lock = TRUE است.

در ادامه حرکت شرط حلقه FALSE است، پس دستور swap اجرا نمی‌گردد، و کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

```
P1:
/*critical_section*/
```

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P1 مشغول حرکت است.

(گام ۲): همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P1 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

```
P1:
/*critical_section*/
③ lock=FALSE;
```

حال در ادامه فرآیند P1 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

```
P1:
/*remainder_section*/
```

همانطور که در (گام ۲) گفتیم قرار شد که همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P1 مشغول حرکت است.

(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P2 بدهید.

فرض کنید فرآیند P2 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P2:
① key[2]= TRUE;
② while (key[2])  swap (key[2],lock);
```

توجه: هم اکنون key[2] =TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[2] و lock جابه جا می‌شوند، به صورت زیر:

```
swap (key[2],lock);
```

توجه: پس از جا به جایی key[2] = FALSE و lock = TRUE است.

در ادامه حرکت، شرط حلقه FALSE است، پس دستور swap اجرا نمی‌گردد، و کنترل برنامه از حلقه خارج می‌شود و کنترل برنامه داخل ناحیه بحرانی فرآیند P2 قرار می‌گیرد، به صورت زیر:

```
P2:
/*critical_section*/
```

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند دوم را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۴) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P2 مشغول حرکت است.

(گام ۴): همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:
فرض کنید فرآیند P2 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P2:
/*critical_section*/
③ lock=FALSE;

حال در ادامه فرآیند P2 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P2:
/*remainder_section*/

همانطور که در (گام ۴) گفتیم قرار شد که همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۵) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P2 مشغول حرکت است.

(گام ۵): فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست. یعنی:
فرض کنید فرآیند P2 نیز قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P2:
① key[2]= TRUE;
② while (key[2]) swap (key[2],lock);

توجه: هم اکنون key[2] =TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[2] و lock جابه جا می‌شوند، به صورت زیر:

swap (key[2],lock);

توجه: پس از جا به جایی key[2] = FALSE و lock = TRUE است.

در ادامه حرکت، شرط حلقه FALSE است، پس دستور swap اجرا نمی‌گردد، و کنترل برنامه از حلقه خارج می‌شود و کنترل برنامه داخل ناحیه بحرانی فرآیند P2 قرار می‌گیرد، به صورت زیر:

P2:
/*critical_section*/

همانطور که در (گام ۵) گفتیم قرار شد که فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، ، خب شد، فرآیند دوم مجددا وارد ناحیه بحرانی خودش شد. بنابراین شرط پیشرفت برقرار است.

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۲) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۳) در ادامه یه آدم دیگه رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۴) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۵) در نهایت اگه همون آدم دوباره تونست بره داخل باجه تلفن، اونوقت شرط پیشرفت برقرار است. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن نبود، وقتی از باجه تلفن اومدی بیرون می‌تونی دوباره بری داخل باجه تلفن. به عبارت دیگر اخلاق می‌گه اگه کسی قصد ورود به باجه تلفن رو نداشته باشه یعنی کسی منتظر زدن تلفن نباشه اونوقت یه شخص دیگه‌ای می‌تونه بارها و بارها داخل باجه تلفن همگانی بره. اخلاق اینو می‌گه، اخلاق.

قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، **(گام ۳)** در ادامه فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۴)** در نهایت همان فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_1 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① key[1]=TRUE;
② while (key[1])  swap (key[1],lock);
```

توجه: هم اکنون key[1] =TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جابه جا می‌شوند، به صورت زیر:

```
swap (key[1],lock);
```

توجه: پس از جا به جایی key[1] = FALSE و lock = TRUE است.

در ادامه حرکت شرط حلقه FALSE است، پس دستور swap اجرا نمی‌گردد، و کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

```
P1:
/*critical_section*/
```

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P1 مشغول حرکت است.

(گام ۲): فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P2 بدهید.

فرض کنید فرآیند P2 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P2:
① key[2]= TRUE;
② while (key[2])  swap (key[2],lock);
```

توجه: هم اکنون key[2] =TRUE و lock = TRUE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[2] و lock جابه جا می‌شوند، به صورت زیر:

```
swap (key[2],lock);
```

توجه: پس از جا به جایی key[2] = TRUE و lock = TRUE است.

در ادامه حرکت، شرط حلقه TRUE است، پس دستور swap مجددا اجرا می‌گردد، و کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P2 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P2 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۳) می‌شویم. هم اکنون پردازنده پشت ناحیه بحرانی فرآیند P2 در یک حلقه انتظار، دچار انتظار مشغول است.

(گام ۳): فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P2 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

```
P1:
/*critical_section*/
③ lock=FALSE;
```

حال در ادامه فرآیند P1 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

```
P1:
/*remainder_section*/
```

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۴) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P1 مشغول حرکت است.

(گام ۴): فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. یعنی:

فرض کنید فرآیند P_1 قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① key[1]=TRUE;
② while (key[1])  swap (key[1],lock);
```

توجه: هم اکنون key[1] =TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جابه جا می‌شوند، به صورت زیر:

```
swap (key[1],lock);
```

توجه: پس از جا به جایی key[1] = FALSE و lock = TRUE است.

در ادامه حرکت شرط حلقه FALSE است، پس دستور swap اجرا نمی‌گردد، و کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

```
P1:
/*critical_section*/
```

همانطور که در (گام ۴) گفتیم قرار شد که فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب شد، فرآیند اول مجددا وارد ناحیه بحرانی خودش شد و فرآیند دوم دچار گرسنگی شده است. بنابراین شرط انتظار محدود به دلیل گرسنگی برقرار نیست.

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، **(گام ۲)** سپس یه آدم دیگه رو جور کن پشت در باجه تلفن همگانی قرار بده، **(گام ۳)** در ادامه آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، **(گام ۴)** در نهایت اگه همون آدم دوباره تونست بره داخل باجه تلفن، اونوقت اون یکی آدمه دچار گرسنگی شده. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن بود، وقتی از باجه تلفن اومدی بیرون

نباید دوباره بری داخل باجه تلفن چون اون موقع دوستت رو دچار گرسنگی کـردی. اخـلاق اینـو می‌گه، اخلاق.

قانون دوم ارسطو(آزمون بن بست)

جهت بررسی بن بست از همان قانون دوم استفاده می‌شود. در واقع روال بررسی همان قـانون دوم است، اما نتیجه قانون متفاوت است.

فرآیند اول و دوم را به طور همرونـد در سیسـتم تـک پردازنـده‌ای و یـا مـوازی در سیسـتم چنـد پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شـوند و هـردو بـاهم پشـت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده اسـت و شـرط انتظـار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی بـه طـور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خـواب رفته‌اند که در این حالت بن‌بست رخ داده است.

P1:
While (TRUE){
① key[1]= TRUE;
② while (key[1]) swap (key[1],lock);
/*critical section*/
③ lock=FALSE;
/*remainder_section*/
}

P2:
While (TRUE) {
① key[2]= TRUE;
② while (key[2]) swap (key[2],lock);
/*critical section*/
③ lock=FALSE;
/*remainder_section*/
}

توجه: مقادیر اولیه براساس صورت سؤال به صورت زیر است:

lock = FALSE, Key[1] = FALSE, Key[2] = FALSE

فرض کنید فرآیند P_1 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:
① key[1]=TRUE;

همچنین فرض کنید فرآیند P_2 نیز به شکل همروند یا موازی با فرآینـد P_1 قصـد دارد وارد ناحیـه بحرانی خودش شود، به صورت زیر:

P2:
① key[2]=TRUE;

اگر فرض شود که دستور Swap به صورت اتمیک و تجزیه‌ناپذیر اجرا شود، آنگاه شـرط انحصـار متقابل برقرار است.

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود بـه ناحیـه‌ی بحرانـی را دارنـد. بنـابراین آرایه فرآیندها هر دو key[1] = TRUE و key[2] = TRUE می‌شود، زیرا هر دو فرآینـد بـه شـکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند.

دستور swap جای مقادیر را جا به جا می‌کند. مقدار اولیـه متغیـر lock برابـر FALSE اسـت، هـر فرآیندی که زودتر دستور swap را اجرا کند، داخل ناحیه بحرانی مـی‌شـود و فرآینـدی کـه دیرتـر دستور swap را اجرا کند، باید صبر پیشه کند و در حلقه انتظار بچرخد.

فرض کنید، فرآیند P_1 **زودتر** و فرآیند P_2 **دیرتر** اقدام به اجرای دستور swap کنند، بنابراین فرآینـد P_1 وارد ناحیه بحرانی می‌شود، اما فرآیند P_2 باید در یک حلقه انتظار مشغول، مشغول باشـد. پـس انحصار متقابل رعایت می‌شود. به صورت زیر:

P1:

② while (key[1]) swap (key[1],lock);

توجه: هم اکنون key[1] = TRUE و lock = FALSE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جا به جا مـی‌شـوند، بـه صورت زیر:

swap (key[1],lock);

توجه: پس از جا به جایی key[1] = FALSE و lock = TRUE است.

در ادامه حرکت شرط حلقه FALSE است، پس دستور swap اجرا نمـی‌گـردد، و کنتـرل برنامـه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

P1:

/*critical_section*/

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P2 بدهید.

P2:

② while (key[2]) swap (key[2],lock);

توجه: هم اکنون key[2] = TRUE و lock = TRUE است.

شرط حلقه TRUE است، پس توسط دستور swap مقادیر key[1] و lock جا به جا مـی‌شـوند، بـه صورت زیر:

swap (key[1],lock);

توجه: پس از جا به جایی key[2] = TRUE و lock = TRUE است.

در ادامه حرکت شرط حلقه TRUE است، پس دستور swap مجددا اجرا می‌گردد، و کنترل برنامـه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P2 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شـود

و فرآیند P2 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.
همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. خب هر دو باهم پشت ناحیه بحرانی خودشان مسدود نشدند. فرآیند اول توانست وارد ناحیه بحرانی خودش شود، اما فرآیند دوم نتوانست وارد ناحیه بحرانی خودش شود. بنابراین بن بست رخ نداده است.

فرم ساده قانون دوم ارسطو (آزمون بن بست)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هر دو باهم نتونستن به طور همزمان وارد باجه تلفن همگانی بشن و هردو باهم پشت در باجه تلفن همگانی مسدود شدن اونوقت بن بست رخ داده. اخلاق دیگه اینجا چیزی نمی‌گه و سکوت می‌کنه، چون دیگه بن‌بست شده!

توجه: برای برقرار بودن شرط انتظار محدود باید **قانون دوم ارسطو** (آزمون بن بست) و **قانون چهارم ارسطو**(آزمون گرسنگی) هر دو باهم برقرار باشند. بنابراین شرط انتظار محدود در سوال مطرح شده به دلیل وجود گرسنگی برقرار نیست.

صورت سوال به این شکل است:

آیا کد مطرح شده می‌تواند راه حلی برای دو پردازش همروند باشد؟

۱) راه حل صحیح نیست زیرا انحصار متقابل رعایت نمی‌شود.

گزینه اول نادرست است، زیرا شرط انحصار متقابل برقرار است و رعایت می‌شود.

۲) راه حل صحیح نیست زیرا شرط پیشرفت برقرار نیست.

گزینه دوم نادرست است، زیرا شرط پیشرفت برقرار است و رعایت می‌شود.

۳) راه حل صحیح نیست زیرا تضمینی برای محدودیت زمان انتظار ندارد.

گزینه سوم درست است، زیرا شرط انتظار محدود برقرار نیست و رعایت نمی‌شود.

۴) راه حل صحیح است.

گزینه چهارم نادرست است، زیرا شرط انتظار محدود برقرار نیست و رعایت نمی‌شود.

توجه: سازمان سنجش آموزش کشور، گزینه سوم را به عنوان پاسخ نهایی اعلام کرده بود.

۸- گزینه ( ) صحیح است.

شرایط رقابتی (مسابقه)

هرگاه دو یا چند فرآیند همزمان با هم وارد ناحیه‌ی بحرانی (منبع مشترک) شوند، شرایط رقابتی پیش می‌آید. در شرایط رقابتی، نتیجه‌ی نهایی بستگی به ترتیب دسترسی‌ها دارد. در واقع

فرآیندهای همکار بر هم اثر دارند و اینکه پردازنده، به چه ترتیبی و در چه زمان‌هایی بین آنها تعویض متن انجام دهد در ایجاد پاسخ نهایی اثرگذار خواهد بود. بنابراین علت شرایط رقابت **تعویض متن پردازنده** بین فرآیندهای همکار است.

برای کنترل شرایط رقابتی، باید راه حلی ارائه شود که سه شرط زیر را به عنوان **معیارهای اخلاقی** در رقابت، رعایت کند:

۱- شرط انحصار متقابل

برای برقرای شرط انحصارمتقابل، عامل مشترک را اسکورت کنید، مانند زمانی که وارد باجه‌ی تلفن همگانی (عامل مشترک) می‌شوید، در را می‌بندید تا مانع ورود شخص دیگری گردید! در عالم انسان‌ها، هیچ دو فردی نباید به طور همزمان وارد عامل مشترک شوند. در عالم فرآیندها نیز هیچ دو فرآیندی نباید به طور همزمان وارد عامل مشترک (ناحیه بحرانی) شوند. استفاده‌ی همزمان از عامل مشترک معنا ندارد! (اخلاقی نیست) بنابراین باید راهی پیدا کنیم که از ورود همزمان دو یا چند فرآیند به ناحیه‌ی بحرانی جلوگیری کند. به عبارت دیگر، آنچه که ما به آن نیاز داریم، انحصار متقابل است که در متون فارسی به آن دو به دو ناسازگاری یا مانعه الجمعی نیز گفته می‌شود، یعنی اگر یکی از فرآیندها در حال استفاده از حافظه‌ی اشتراکی، فایل اشتراکی و یا هر عامل اشتراکی رقابت‌زاست باید مطمئن باشیم که دیگر فرآیندها، در آن زمان از انجام همان کار محروم می‌باشند. در واقع از بین تمام فرآیندها، در هر لحظه تنها یک فرآیند مجاز است، در عامل مشترک باشد. بدین معنی که اگر فرآیندی در ناحیه‌ی بحرانی است، از ورود فرآیندهای دیگر به همان ناحیه‌ی بحرانی جلوگیری شود و تا خارج شدن فرآیند اول منتظر بمانند، زیرا هیچ دو فرآیندی نباید به طور همزمان وارد ناحیه‌ی بحرانی شوند. به یاد داشته باشید که استفاده‌ی همزمان از عامل مشترک معنا ندارد!

بنابراین برای برقراری شرط انحصار متقابل باید ساختاری را طراحی کنیم که در هر لحظه فقط یک فرآیند مجوز ورود به ناحیه‌ی بحرانی را داشته باشد. لذا هر فرآیند برای ورود به بخش بحرانی‌اش باید اجازه بگیرد. بخشی از کد فرآیند که این اجازه گرفتن را پیاده‌سازی می‌کند، بخش ورودی نام دارد. بخش بحرانی می‌تواند با بخش خروجی دنبال شود. این بخش خروجی کاری می‌کند که فرآیندهای دیگر بتوانند وارد ناحیه‌ی بحرانی‌شان، شوند. بقیه‌ی کد فرآیند را بخش باقی‌مانده می‌نامند. بنابراین ساختار کلی فرآیندها برای برقراری شرط انحصار متقابل به صورت زیر می‌باشد:

```
P (int i)  {
while ( TRUE)  {
entry_section () ;  // تلاش برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی
critical_section (); // ناحیه‌ی بحرانی
exit_section () ; // اعلام خروج از ناحیه‌ی بحرانی
remainder_section () ; // ناحیه‌ی باقی مانده
   }
}
```

توجه: بدترین شرایط وقتی است که یک فرآیند بخواهد بارها و بارها وارد ناحیه بحرانی خود شود، برای اینکه سخت‌ترین شرایط بررسی شود، ناحیه بحرانی را داخل حلقه بی‌نهایت قرار می‌دهیم.

۲- شرط پیشرفت

فرآیندی که داوطلب ورود به ناحیه‌ی بحرانی نیست و نیز در ناحیه‌ی بحرانی قرار ندارد، نباید در رقابت برای ورود سایر فرآیندها به ناحیه‌ی بحرانی شرکت کند، به عبارت دیگر، نباید مانع ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شود. در یک بیان ساده‌تر می‌توان گفت، فرآیندی که در ناحیه‌ی باقی‌مانده قرار دارد، حق جلوگیری از ورود فرآیندهای دیگر به ناحیه‌ی بحرانی را ندارد، یعنی نباید در تصمیم‌گیری برای ورود فرآیندها به ناحیه‌ی بحرانی شرکت کند.

۳- شرط انتظار محدود

فرآیندهایی که نیاز به ورود به ناحیه‌ی بحرانی دارند، باید مدت انتظارشان محدود باشد، یعنی نباید به طور نامحدود در حالت انتظار باقی بمانند.

انتظار نامحدود به دو دسته می‌باشد: (۱) قحطی، (۲) بن‌بست، بنابراین نباید در شرایط رقابتی بین فرآیندها، قحطی یا بن‌بست رخ دهد. برای اینکه شرط انتظار محدود برقرار باشد، باید هم قحطی و هم بن‌بست رخ ندهد.

قحطی (گرسنگی)

در عالم زندگی قحطی زمانی رخ می‌دهد که عده‌ای مدام از منابع مشترک استفاده کنند، و عده‌ای دیگر قادر به استفاده از منابع مشترک نباشند. زیرا دسته‌ی اول از اختصاص منابع به دسته‌ی دوم به طور مداوم و بدون رعایت یک حد بالای مشخص جلوگیری می‌کنند. در عالم فرآیندها نیز هرگاه فرآیندی به مدت نامعلوم و بدون رعایت یک حد بالای مشخص در انتظار گرفتن یک منبع بحرانی یا دسترسی به یک عامل مشترک بماند و فرآیندی دیگر مدام در حال استفاده از منبع بحرانی باشد، در این حالت فرآیند اول دچار قحطی شده است. بنابراین در صورت اقدام یک فرآیند برای ورود به ناحیه‌ی بحرانی، باید محدودیتی برای تعداد دفعاتی که سایر فرآیندها می‌توانند وارد ناحیه‌ی بحرانی شوند، وجود داشته باشد تا قحطی رخ ندهد.

بن‌بست

به وضعیتی که در آن مجموعه‌ای متشکل از دو یا چند فرآیند برای همیشه منتظر یکدیگر بمانند (مسدود) و به عبارت دیگر دچار سیکل انتظار ابدی شوند، بن‌بست گفته می‌شود.

توجه: به تفاوت قحطی و بن‌بست توجه کنید، در قحطی فرآیندی مدام در حال کار و فرآیندی دیگر به مدت نامعلوم در انتظار است. اما در بن‌بست، مجموعه‌ای از فرآیندها در سیکل انتظار ابدی، گرفتار شده‌اند. نه راه پس دارند و نه راه پیش.

توجه: در کنترل شرایط رقابتی، رعایت شرط انحصار متقابل، شرط لازم و رعایت شروط پیشروی و انتظار محدود، شروط کافی برای ارائه‌ی یک راه‌حل **جامع** و **اخلاقی** به شمار می‌آیند.
ابتدا کد مطرح شده در صورت سوال را برای دو فرآیند P_0 و P_1 به صورت زیر بازنویسی می‌کنیم:

```
P0:
wait (0){
① C[0]= FALSE;
② while(C[1]) do;
}
/*critical_section*/
Signal(0){
③ C[0]=TRUE;
}
/*remainder_section*/
}
```

```
P1:
wait (1){
① C[1]= FALSE;
② while(C[0]) do;
}
/*critical_section*/
Signal(1){
③ C[1]=TRUE;
}
/*remainder_section*/
```

توجه: مقادیر اولیه بر اساس صورت سوال به صورت زیر است:

C[0] = TRUE, C[1] = TRUE

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

برای کنترل برقراری شرط انحصار متقابل، شرط پیشرفت و شرط انتظار محدود (گرسنگی و بن‌بست) از آزمون‌های زیر استفاده می‌کنیم:

توجه: ما نام این آزمون‌ها را به عنوان مبدع آن «**قوانین ارسطو**» نام‌گذاری کردیم، این قوانین به «**قوانین چهارگانه ارسطو**» نیز موسوم است.

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
① C[0]= FALSE;
② while(C[1]) do;
```

توجه: هم اکنون C[1] = TRUE است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خـودش قـرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

خط اول فرآیند P1 را اجرا کنید.

P1:

① C[1] =FALSE;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط دوم فرآیند P0 را مجددا اجرا کنید.

P0:

② while(C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآینـد P_0 قرار می‌گیرد، به صورت زیر:

P0:

/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قـرار بـدهیم، خب قرار دادیم. حال در ادامه آزمون اول وارد (گام ۲) می‌شـویم. هـم اکنـون پردازنـده در ناحیـه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفـق شـود کـه وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیـه بحرانـی خـودش شـود، بـه صورت زیر:

P1:

② while(C[0]) do;

توجه: هم اکنون C[0] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

P1:

/*critical_section*/

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض

شده است، خب موفق شد. فرآیند دوم هم توانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط اول انحصار متقابل برقرار نیست.

فرم ساده قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، **(گام ۲)** سپس یه آدم دیگه رو هم جور کن که بخواد وارد باجه تلفن همگانی بشه، اگه اونم تونست وارد باجه تلفن همگانی بشه اونوقت شرط اول انحصار متقابل نقض شده. اخلاق می‌گه اگه یه نفر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگه بشه شرط اول انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

قانون دوم ارسطو (آزمون دوم شرط انحصار متقابل)

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند آنگاه در این حالت شرط انحصار متقابل نقض شده است.

```
P0:
wait (0){
① C[0]= FALSE;
② while(C[1]) do;
}
/*critical_section*/
Signal(0){
③ C[0]=TRUE;
}
/*remainder_section*/
}
```

```
P1:
wait (1){
① C[1]= FALSE;
② while(C[0]) do;
}
/*critical_section*/
Signal(1){
③ C[1]=TRUE;
}
/*remainder_section*/
```

توجه: مقادیر اولیه بر اساس صورت سوال به صورت زیر است:

C[0] = TRUE, C[1] = TRUE

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
① C[0]=FALSE;
```

همچنین فرض کنید فرآیند P1 نیز به شکل همروند یا موازی با فرآیند P0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① C[1]=FALSE;
```

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود بـه ناحیـه‌ی بحرانـی را دارنـد، بنـابراین آرایه فرآیندها هر دو C[0] = FALSE و C[1] = FALSE می‌شود، زیـرا هـر دو فرآینـد بـه شـکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. هنگامیکه دو فرآینـد بـه دسـتور while می‌رسند، خط ② برای فرآیند P_0 و P1 برقـرار نیسـت، بنـابراین هـردو فرآینـد بـاهم وارد ناحیـه بحرانی خودشان می‌شوند، پس انحصار متقابل رعایت نمی‌شود. به صورت زیر:

P_0:
② while (C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآینـد P_0 قرار می‌گیرد، به صورت زیر:

P0:
/*critical_section*/

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_1:
② while (C[0]) do;

توجه: هم اکنون C[0] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآینـد P1 قرار می‌گیرد، به صورت زیر:

P1:
/*critical_section*/

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تـک پردازنـده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانسـتند وارد ناحیـه بحرانـی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفـق شدند. فرآیند اول و دوم هردو توانستند باهم وارد ناحیه بحرانی خودشان شوند. بنابراین شرط دوم انحصار متقابل نیز برقرار نیست.

فرم ساده قانون دوم ارسطو (آزمون دوم شرط انحصار متقابل)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگـه هر دو تونستن به طور همزمان وارد باجه تلفن همگانی بشنن اونوقت شـرط دوم انحصـار متقابـل نقض شده. اخلاق می‌گه دو نفر نباید همزمان باهم داخل باجه تلفن همگانی باشن، یعنـی اگـه یـه نفر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفـن همگـانی بشـه و اگـه بشـه شرط دوم انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

توجه: برای برقرار بودن شرط انحصار متقابل باید **قانون اول ارسطو** (آزمون اول شرط انحصار متقابل) و **قانون دوم ارسطو** (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشند. بنابراین شرط انحصار متقابل در سوال مطرح شده برقرار نیست.

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۳)** در ادامه فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، **(گام ۴)** سپس همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۵)** در نهایت همان فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= FALSE;
② while(C[1]) do;

توجه: هم اکنون C[1] = TRUE است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =FALSE;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط دوم فرآیند P0 را مجددا اجرا کنید.

P0:
② while(C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:
/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P1 مشغول حرکت است.

(گام ۲): همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P0:

/*critical_section*/

③ C[0]=TRUE;

حال در ادامه فرآیند P0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P0:

/*remainder_section*/

همانطور که در (گام ۲) گفتیم قرار شد که همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

② while (C[0]) do;

توجه: هم اکنون C[0] = TRUE است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند دوم را داخل ناحیه بحرانی خودش قرار بدهیم، خب نتوانستیم قرار بدهیم. پس در ادامه آزمون سوم نمی‌توانیم وارد (گام ۴) و به تبع (گام ۵) شویم. بنابراین شرط پیشرفت برقرار نیست. شرط پیشرفت در صورتی رعایت می‌شود که هر پنج گام قانون سوم به طور کامل طی شود، در غیر اینصورت شرط پیشرفت برقرار نیست. به عبارت دیگر شرط پیشرفت در صورتی برقرار نیست که فرآیندی که داخل ناحیه باقی‌مانده قرار دارد، جلوی پیشرفت (یعنی ورود به ناحیه بحرانی) فرآیند رقیب را بگیرد.

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۲) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۳) در ادامه یه آدم دیگه رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۴) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۵) در نهایت اگه همون آدم دوباره تونست بره داخل باجه تلفن، اونوقت شرط پیشرفت برقرار است. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن نبود، وقتی از باجه تلفن اومدی بیرون می‌تونی دوباره بری داخل باجه تلفن. به عبارت دیگر اخلاق می‌گه اگه کسی قصد ورود به باجه تلفن رو نداشته باشه یعنی کسی منتظر زدن تلفن نباشه اونوقت یه شخص دیگه‌ای می‌تونه بارها و بارها داخل باجه تلفن همگانی بره. اخلاق اینو می‌گه، اخلاق.

قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، **(گام ۳)** در ادامه فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۴)** در نهایت همان فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

① C[0]= FALSE;

② while(C[1]) do;

توجه: هم اکنون C[1] = TRUE است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

خط اول فرآیند P1 را اجرا کنید.

P1:

① C[1] =FALSE;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط دوم فرآیند P0 را مجددا اجرا کنید.

P0:

② while(C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:

/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:

② while(C[0]) do;

توجه: هم اکنون C[0] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

P1:

/*critical_section*/

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدهیم، خب نتوانستیم قرار بدهیم. پس در ادامه آزمون چهارم نمی‌توانیم وارد (گام ۳) و به تبع (گام ۴) شویم. بنابراین گرسنگی رخ نمی‌دهد. گرسنگی هنگامی رخ می‌دهد که هر چهار گام قانون چهارم به طور کامل طی شود، در غیر اینصورت گرسنگی رخ نمی‌دهد.

توجه مهم: گرسنگی را هنگامی چک می‌کنیم که شرط انحصار متقابل برقرار باشد. اگر شرط انحصار متقابل برقرار نباشد، هیچگاه گرسنگی رخ نمی‌دهد. به عبارت دیگر شرط لازم برای ایجاد گرسنگی، برقراری شرط انحصار متقابل است و شرط کافی، بررسی گرسنگی یعنی بررسی قانون چهارم است. در جایی که انحصار متقابل رعایت نمی‌شود، کسی گرسنه نمی‌ماند. در جاییکه که اگر کسی داخل ناحیه بحرانی است، افراد دیگری هم همزمان وارد ناحیه بحرانی می‌شوند، یعنی انحصار متقابل و ادب و اخلاق برقرار نیست، در چنین فضایی کسی گرسنه نمی‌ماند.

توجه مهم: در کنترل گرسنگی باید یکی از فرآیندها **حتما** داخل ناحیه بحرانی باشد و دیگری **حتما** پشت ناحیه بحرانی بماند و نه اینکه یکی از فرآیندها داخل ناحیه باقی‌مانده باشد و دیگری پشت ناحیه بحرانی بماند این مورد آخر نقض پیشرفت هست و نه وقوع گرسنگی.

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بـده، **(گام ۲)** سـپس یـه آدم دیگـه رو جور کن پشت در باجه تلفن همگانی قرار بده، **(گام ۳)** در ادامه آدم داخل باجه تلفن همگـانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، **(گام ۴)** در نهایت اگه همون آدم دوباره تونست بره داخل باجه تلفن، اونوقت اون یکی آدمه دچار گرسنگی شده. اخلاق می‌گه اگه یه دفعـه داخـل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن بود، وقتی از باجه تلفن اومدی بیـرون نباید دوباره بری داخل باجه تلفن چون اون موقع دوستت رو دچار گرسنگی کـردی. اخـلاق اینـو می‌گه، اخلاق.

قانون دوم ارسطو(آزمون بن بست)

جهت بررسی بن بست از همان قانون دوم استفاده می‌شود. در واقع روال بررسی همان قـانون دوم است، اما نتیجه قانون متفاوت است.

فرآیند اول و دوم را به طور همرونـد در سیسـتم تـک پردازنـده‌ای و یـا مـوازی در سیسـتم چنـد پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شـوند و هـردو بـاهم پشـت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده اسـت و شـرط انتظـار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی بـه طـور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خـواب رفته‌اند که در این حالت بن‌بست رخ داده است.

```
P0:
wait (0){
① C[0]= FALSE;
② while(C[1]) do;
}
/*critical_section*/
Signal(0){
③ C[0]=TRUE;
}
/*remainder_section*/
}
```

```
P1:
wait (1){
① C[1]= FALSE;
② while(C[0]) do;
}
/*critical_section*/
Signal(1){
③ C[1]=TRUE;
}
/*remainder_section*/
```

توجه: مقادیر اولیه بر اساس صورت سوال به صورت زیر است:

C[0] = TRUE, C[1] = TRUE

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
① C[0]=FALSE;
```

همچنین فرض کنید فرآیند P1 نیز به شکل همروند یا موازی با فرآیند P0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:
① C[1]=FALSE;

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند، بنابراین آرایه فرآیندها هر دو C[0] = FALSE و C[1] = FALSE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. هنگامیکه دو فرآیند به دستور while می‌رسند، خط ② برای فرآیند P_0 و P1 برقرار نیست، بنابراین هردو فرآیند باهم وارد ناحیه بحرانی خودشان می‌شوند، پس انحصار متقابل رعایت نمی‌شود. به صورت زیر:

P_0:
② while (C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:
/*critical_section*/

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_1:
② while (C[0]) do;

توجه: هم اکنون C[0] = FALSE است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

P1:
/*critical_section*/

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. خب هر دو باهم پشت ناحیه بحرانی خودشان مسدود نشدند. فرآیند اول و دوم هردو توانستند باهم وارد ناحیه بحرانی خودشان شوند. بنابراین بن بست رخ نداده است.

فرم ساده قانون دوم ارسطو (آزمون بن بست)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هر دو باهم نتونستن به طور همزمان وارد باجه تلفن همگانی بشن و هردو باهم پشت در باجه تلفن

همگانی مسدود شدن اونوقت بن بست رخ داده. اخلاق دیگه اینجا چیزی نمی‌گه و سکوت می‌کنه، چون دیگه بن‌بست شده!

توجه: برای برقرار بودن شرط انتظار محدود باید **قانون دوم ارسطو** (آزمون بن بست) و **قانون چهارم ارسطو** (آزمون گرسنگی) هر دو باهم برقرار باشند. بنابراین شرط انتظار محدود در سوال مطرح شده برقرار است.

صورت سوال به این شکل است:

کدام یک از گزینه‌های زیر درست **نیست**؟

۱) این راه حل استفاده انحصاری از ناحیه بحرانی را برآورده می‌کند.

گزینه اول **نادرست** است، زیرا شرط انحصار متقابل برقرار نیست و برآورده نمی‌کند.

۲) این راه حل شرط انتظار محدود را برآورده می‌کند.

گزینه دوم درست است، زیرا شرط انتظار محدود برقرار است و برآورده می‌کند.

۳) این راه حل همه شرایط ناحیه بحرانی را برآورده می‌کند.

گزینه سوم **نادرست** است، زیرا شرط انحصار متقابل و شرط پیشرفت برقرار نیست و برآورده نمی‌کند و فقط شرط انتظار محدود برقرار است و برآورده می‌کند.

۴) این راه حل شرط پیشرفت را برآورده می‌کند.

گزینه چهارم **نادرست** است، زیرا شرط پیشرفت اصلا برقرار نیست و برآورده نمی‌کند.

توجه: همانطور که واضح و مشخص هست، گزینه‌های اول، سوم و چهارم نادرست هستند و هر سه گزینه می‌تواند پاسخ سوال باشد.

توجه: سازمان سنجش آموزش کشور در کلید اولیه خود ابتدا گزینه چهارم را به عنوان پاسخ اعلام نمود، سپس در کلید نهایی نظر خود را عوض کرد و گزینه سوم را به عنوان پاسخ اعلام کرد، که باز هم پاسخ درست نبود.

۹- گزینه ( ) صحیح است.

شرایط رقابتی (مسابقه)

هرگاه دو یا چند فرآیند همزمان با هم وارد ناحیه‌ی بحرانی (منبع مشترک) شوند، شرایط رقابتی پیش می‌آید. در شرایط رقابتی، نتیجه‌ی نهایی بستگی به ترتیب دسترسی‌ها دارد. در واقع فرآیندهای همکار بر هم اثر دارند و اینکه پردازنده، به چه ترتیبی و در چه زمان‌هایی بین آنها تعویض متن انجام دهد در ایجاد پاسخ نهایی اثرگذار خواهد بود. بنابراین علت شرایط رقابت **تعویض متن پردازنده** بین فرآیندهای همکار است.

برای کنترل شرایط رقابتی، باید راه حلی ارائه شود که سه شرط زیر را به عنوان **معیارهای اخلاقی** در رقابت، رعایت کند:

۱- شرط انحصار متقابل

برای برقرای شرط انحصارمتقابل، عامل مشترک را اسکورت کنید، مانند زمانی که وارد باجه‌ی تلفن همگانی (عامل مشترک) می‌شوید، در را می‌بندید تا مانع ورود شخص دیگری گردید! در عالم انسان‌ها، هیچ دو فردی نباید به طور همزمان وارد عامل مشترک شوند. در عالم فرآیندها نیز هیچ دو فرآیندی نباید به طور همزمان وارد عامل مشترک (ناحیه بحرانی) شوند. استفاده‌ی همزمان از عامل مشترک معنا ندارد! (اخلاقی نیست) بنابراین باید راهی پیدا کنیم که از ورود همزمان دو یا چند فرآیند به ناحیه‌ی بحرانی جلوگیری کند. به عبارت دیگر، آنچه که ما به آن نیاز داریم، انحصار متقابل است که در متون فارسی به آن دو به دو ناسازگاری یا مانعه الجمعی نیز گفته می‌شود، یعنی اگر یکی از فرآیندها در حال استفاده از حافظه‌ی اشتراکی، فایل اشتراکی و یا هر عامل اشتراکی رقابت‌زاست باید مطمئن باشیم که دیگر فرآیندها، در آن زمان از انجام همان کار محروم می‌باشند. در واقع از بین تمام فرآیندها، در هر لحظه تنها یک فرآیند مجاز است، در عامل مشترک باشد. بدین معنی که اگر فرآیندی در ناحیه‌ی بحرانی است، از ورود فرآیندهای دیگر به همان ناحیه‌ی بحرانی جلوگیری شود و تا خارج شدن فرآیند اول منتظر بمانند، زیرا هیچ دو فرآیندی نباید به طور همزمان وارد ناحیه‌ی بحرانی شوند. به یاد داشته باشید که استفاده‌ی همزمان از عامل مشترک معنا ندارد!

بنابراین برای برقراری شرط انحصار متقابل باید ساختاری را طراحی کنیم که در هر لحظه فقط یک فرآیند مجوز ورود به ناحیه‌ی بحرانی را داشته باشد. لذا هر فرآیند برای ورود به بخش بحرانی‌اش باید اجازه بگیرد. بخشی از کد فرآیند که این اجازه گرفتن را پیاده‌سازی می‌کند، بخش ورودی نام دارد. بخش بحرانی می‌تواند با بخش خروجی دنبال شود. این بخش خروجی کاری می‌کند که فرآیندهای دیگر بتوانند وارد ناحیه‌ی بحرانی‌شان، شوند. بقیه‌ی کد فرآیند را بخش باقی‌مانده می‌نامند. بنابراین ساختار کلی فرآیندها برای برقراری شرط انحصار متقابل به صورت زیر می‌باشد:

```
P (int i)  {
while ( TRUE)  {
entry_section () ;  // تلاش برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی
critical_section (); // ناحیه‌ی بحرانی
exit_section () ; // اعلام خروج از ناحیه‌ی بحرانی
remainder_section () ; // ناحیه‌ی باقی مانده
   }
}
```

توجه: بدترین شرایط وقتی است که یک فرآیند بخواهد بارها و بارها وارد ناحیه بحرانی خود شود، برای اینکه سخت‌ترین شرایط بررسی شود، ناحیه بحرانی را داخل حلقه بی‌نهایت قرار می‌دهیم.

۲- شرط پیشرفت

فرآیندی که داوطلب ورود به ناحیه‌ی بحرانی نیست و نیز در ناحیه‌ی بحرانی قرار ندارد، نباید در رقابت برای ورود سایر فرآیندها به ناحیه‌ی بحرانی شرکت کند، به عبارت دیگر، نباید مانع ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شود. در یک بیان ساده‌تر می‌توان گفت، فرآیندی که در ناحیه-ی باقی‌مانده قرار دارد، حق جلوگیری از ورود فرآیندهای دیگر به ناحیه‌ی بحرانی را ندارد، یعنی نباید در تصمیم‌گیری برای ورود فرآیندها به ناحیه‌ی بحرانی شرکت کند.

۳- شرط انتظار محدود

فرآیندهایی که نیاز به ورود به ناحیه‌ی بحرانی دارند، باید مدت انتظارشان محدود باشد، یعنی نباید به طور نامحدود در حالت انتظار باقی بمانند.

انتظار نامحدود به دو دسته می‌باشد: (۱) قحطی، (۲) بن‌بست، بنابراین نباید در شرایط رقابتی بین فرآیندها، قحطی یا بن‌بست رخ دهد. برای اینکه شرط انتظار محدود برقرار باشد، باید هم قحطی و هم بن‌بست رخ ندهد.

قحطی (گرسنگی)

در عالم زندگی قحطی زمانی رخ می‌دهد که عده‌ای مدام از منابع مشترک استفاده کنند، و عده‌ای دیگر قادر به استفاده از منابع مشترک نباشند. زیرا دسته‌ی اول از اختصاص منابع به دسته‌ی دوم به طور مداوم و بدون رعایت یک حد بالای مشخص جلوگیری می‌کنند. در عالم فرآیندها نیز هرگاه فرآیندی به مدت نامعلوم و بدون رعایت یک حد بالای مشخص در انتظار گرفتن یک منبع بحرانی یا دسترسی به یک عامل مشترک بماند و فرآیندی دیگر مدام در حال استفاده از منبع بحرانی باشد، در این حالت فرآیند اول دچار قحطی شده است. بنابراین در صورت اقدام یک فرآیند برای ورود به ناحیه‌ی بحرانی، باید محدودیتی برای تعداد دفعاتی که سایر فرآیندها می‌توانند وارد ناحیه‌ی بحرانی شوند، وجود داشته باشد تا قحطی رخ ندهد.

بن‌بست

به وضعیتی که در آن مجموعه‌ای متشکل از دو یا چند فرآیند برای همیشه منتظر یکدیگر بمانند (مسدود) و به عبارت دیگر دچار سیکل انتظار ابدی شوند، بن‌بست گفته می‌شود.

توجه: به تفاوت قحطی و بن‌بست توجه کنید، در قحطی فرآیندی مدام در حال کار و فرآیندی دیگر به مدت نامعلوم در انتظار است. اما در بن‌بست، مجموعه‌ای از فرآیندها در سیکل انتظار ابدی، گرفتار شده‌اند. نه راه پس دارند و نه راه پیش.

توجه: در کنترل شرایط رقابتی، رعایت شرط انحصار متقابل، شرط لازم و رعایت شروط پیشروی و انتظار محدود، شروط کافی برای ارائه‌ی یک راه‌حل **جامع** و **اخلاقی** به شمار می‌آیند.

ابتدا کد مطرح شده در صورت سوال را برای دو فرآیند P_0 و P_1 به صورت زیر بازنویسی می‌کنیم:

```
P0:
wait (0){
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;
}
/*critical_section*/
Signal(0){
④ C[0]=FALSE;
}
/*remainder_section*/
```

```
P1:
wait (1){
① C[1]= TRUE;
② turn= 0;
③ while(C[1] && turn = 0) do;
}
/*critical_section*/
Signal(1){
④ C[1]=FALSE;
}
/*remainder_section*/
```

توجه: مقادیر اولیه بر اساس صورت سوال به صورت زیر است:

turn =1, C[0] = TRUE, C[1] = TRUE

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

برای کنترل برقراری شرط انحصار متقابل، شرط پیشرفت و شرط انتظار محدود (گرسنگی و بن‌بست) از آزمون‌های زیر استفاده می‌کنیم:

توجه: ما نام این آزمون‌ها را به عنوان مبدع آن «**قوانین ارسطو**» نام‌گذاری کردیم، این قوانین به «**قوانین چهارگانه ارسطو**» نیز موسوم است.

قانون اول ارسطو(آزمون اول شرط انحصار متقابل)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;
```

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

دو خط اول فرآیند P1 را اجرا کنید.

```
P1:
① C[1] =TRUE;
② turn= 0;
```

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط سوم فرآیند P0 را مجددا اجرا کنید.

```
P0:
③ while(C[0] && turn = 1) do;
```

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

```
P0:
/*critical_section*/
```

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون اول وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
③ while(C[1] && turn = 0) do;
```

توجه:هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، خب موفق نشد. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط اول انحصار متقابل برقرار است.

فرم ساده قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، **(گام ۲)** سپس یه آدم دیگه رو هم جور کن که بخواد وارد باجه تلفن همگانی بشه، اگه اونم تونست وارد باجه تلفن همگانی بشه اونوقت شرط اول انحصار متقابل نقض شده. اخلاق می‌گه اگه یه نفر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگه بشه شرط اول انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است.

```
P0:
wait (0){
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;
}
/*critical_section*/
Signal(0){
④ C[0]=FALSE;
}
/*remainder_section*/
```

```
P1:
wait (1){
① C[1]= TRUE;
② turn= 0;
③ while(C[1] && turn = 0) do;
}
/*critical_section*/
Signal(1){
④ C[1]=FALSE;
}
/*remainder_section*/
```

توجه: مقادیر اولیه براساس صورت سؤال به صورت زیر است:

turn =1, C[0] = TRUE, C[1] = TRUE

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
① C[0]= TRUE;
```

همچنین فرض کنید فرآیند P1 نیز به شکل همروند یا موازی با فرآیند P0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① C[1]= TRUE;
```

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود بـه ناحیـه‌ی بحرانـی را دارنـد. بنـابراین تابلوی وضعیت فرآیندها هر دو C[0] = TRUE و C[1] = TRUE می‌شود، زیرا هر دو فرآینـد بـه شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. اما متغیر نوبت turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآینـد، شـماره‌ی فرآینـد خـود را در متغیر نوبت turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیـره کنـد، فرآینـدی اسـت کـه شماره‌اش در متغیر نوبت turn باقی می‌ماند و دیگری اثرش در متغیر نوبت turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغییر turn گره خورده است، بنـابراین فرآینـدی که دیرتر متغیر نوبت turn را مقداردهی کرده است، باید صـبر پیشـه کنـد و متغیـر نوبـت turn را نگه‌داری کند و در حلقه‌ی انتظار بچرخد. و فرآیندی که زودتر متغیر نوبت turn را مقداردهی کرده است، وارد ناحیه بحرانی می‌شود.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنـابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتـی کـه دو فرآینـد بـه دسـتور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود، اما فرآینـد P1 بایـد در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P_0:
② turn= 1;

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_1:
② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

P_0:
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآینـد P_0 قرار می‌گیرد، به صورت زیر:

P0:
/*critical_section*/

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_0:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآینـد P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشـت ناحیـه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده بـه انتظـار مشـغول (Busy Waiting) موسوم است.

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تـک پردازنـده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانسـتند وارد ناحیـه بحرانـی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفـق نشدند. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شـرط دوم انحصـار متقابـل نیز برقرار است.

فرم ساده قانون دوم ارسطو (آزمون دوم شرط انحصار متقابل)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هـر دو تونستن به طور همزمان وارد باجه تلفن همگانی بشنن اونوقت شرط دوم انحصار متقابل نقـض شده. اخلاق می‌گه دو نفر نباید همزمان باهم داخل باجه تلفن همگانی باشـن، یعنـی اگـه یـه نفـر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگـه بشـه شـرط دوم انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

توجه: برای برقرار بودن شرط انحصار متقابل باید **قانون اول ارسطو** (آزمـون اول شـرط انحصـار متقابل) و **قانون دوم ارسطو** (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشـند. بنـابراین شرط انحصار متقابل در سوال مطرح شده برقرار است.

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سـپس همـان فرآینـد اول را داخل ناحیه باقی‌مانده خودش قرار بـده، **(گـام ۳)** در ادامـه فرآینـد دوم را داخـل ناحیـه بحرانـی خودش قرار بده، **(گام ۴)** سپس همان فرآیند دوم را داخل ناحیـه بـاقی‌مانـده خـودش قـرار بـده، **(گام ۵)** در نهایت همان فرآیند دوم به ابتدای برنامه برگردد و مجـددا تصـمیم بگیـرد وارد ناحیـه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانـی خـودش شـود، آنگـاه در ایـن حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

دو خط اول فرآیند P1 را اجرا کنید.

```
P1:
① C[1] =TRUE;
② turn= 0;
```

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط سوم فرآیند P0 را مجددا اجرا کنید.

```
P0:
③ while(C[0] && turn = 1) do;
```

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

```
P0:
/*critical_section*/
```

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

```
P0:
/*critical_section*/
④ C[0]= FALSE;
```

حال در ادامه فرآیند P0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

```
P0:
/*remainder_section*/
```

همانطور که در (گام ۲) گفتیم قرار شد که همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند دوم را داخل ناحیه بحرانی خودش قرار بدهیم، خب نتوانستیم قرار بدهیم. پس در ادامه آزمون سوم نمی‌توانیم وارد (گام ۴) و به تبع (گام ۵) شویم. بنابراین شرط پیشرفت برقرار نیست. شرط پیشرفت در صورتی رعایت می‌شود که هر پنج گام قانون سوم به طور کامل طی شود، در غیر اینصورت شرط پیشرفت برقرار نیست. به عبارت دیگر شرط پیشرفت در صورتی برقرار نیست که فرآیندی که داخل ناحیه باقی‌مانده قرار دارد، جلوی پیشرفت (یعنی ورود به ناحیه بحرانی) فرآیند رقیب را بگیرد.

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۲) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۳) در ادامه یه آدم دیگه رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۴) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۵) در نهایت اگه همون آدم دوباره تونست بره داخل باجه تلفن، اونوقت شرط پیشرفت برقرار است. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن نبود، وقتی از باجه تلفن اومدی بیرون می‌تونی دوباره بری داخل باجه تلفن. به عبارت دیگر اخلاق می‌گه اگه کسی قصد ورود به باجه تلفن رو نداشته باشه یعنی کسی منتظر زدن تلفن نباشه اونوقت یه شخص دیگه‌ای می‌تونه بارها و بارها داخل باجه تلفن همگانی بره. اخلاق اینو می‌گه، اخلاق.

قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، **(گام ۳)** در ادامه فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۴)** در نهایت همان فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه

بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

دو خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =TRUE;
② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:
/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۳) می‌شویم. هم اکنون پردازنده پشت ناحیه بحرانی فرآیند P1 در یک حلقه انتظار، دچار انتظار مشغول است.

(گام ۳): فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

فرض کنید فرآیند P0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P0:

/*critical_section*/

④ C[0]=FALSE;

حال در ادامه فرآیند P0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P0:

/*remainder_section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۴) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

(گام ۴): فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. یعنی:

فرض کنید فرآیند P0 قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

① C[0]= TRUE;

② turn= 1;

③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

توجه: شاید در این مرحله پیش خودتان فکر کنید که خب مثل سابق دوباره سراغ فرآیند P1 می‌رویم و با turn= 0 در خط ② راه را برای ورود فرآیند P0 باز می‌کنیم، اول اینکه خب نادرست فکر کردید، دقت کنید که شما اصلا نمی‌توانید مجددا سراغ خط ② از فرآیند P1 بروید چون قبلا از آن خط عبور کرده بودید و اگر هم سراغ فرآیند P1 بروید در ادامه همان خط ③ و حلقه while را مشاهده خواهید کرد. مانند انداختن یک سکه که بعد انداختن به سمت پایین حرکت می‌کند، مگر اینکه دوباره آنرا به سمت بالا پرتاپ کنیم. دوم اینکه از قوانین تبعیت کنید و به هیچ عنوان به مراحل آن دست نزنید و فقط و فقط مطابق قوانین حرکت کنید. اینطوری موفق می‌شوید.

همانطور که در (گام ۴) گفتیم قرار شد که فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب نشد، فرآیند اول نتوانست مجددا وارد ناحیه بحرانی خودش بشود. بنابراین گرسنگی رخ نداده است.

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، **(گام ۲)** سپس یه آدم دیگه رو جور کن پشت در باجه تلفن همگانی قرار بده، **(گام ۳)** در ادامه آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، **(گام ۴)** در نهایت اگه همون آدم دوباره تونست بره داخل باجه تلفن، اونوقت اون یکی آدمه دچار گرسنگی شده. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن بود، وقتی از باجه تلفن اومدی بیرون نباید دوباره بری داخل باجه تلفن چون اون موقع دوستت رو دچار گرسنگی کردی. اخلاق اینو می‌گه، اخلاق.

قانون دوم ارسطو(آزمون بن بست)

جهت بررسی بن بست از همان قانون دوم استفاده می‌شود. در واقع روال بررسی همان قانون دوم است، اما نتیجه قانون متفاوت است.

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند که در این حالت بن‌بست رخ داده است.

```
P0:
wait (0){
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;
}
/*critical_section*/
Signal(0){
④ C[0]=FALSE;
}
/*remainder_section*/
```

```
P1:
wait (1){
① C[1]= TRUE;
② turn= 0;
③ while(C[1] && turn = 0) do;
}
/*critical_section*/
Signal(1){
④ C[1]=FALSE;
}
/*remainder_section*/
```

توجه: مقادیر اولیه براساس صورت سؤال به صورت زیر است:

turn =1, C[0] = TRUE, C[1] = TRUE

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
① C[0]= TRUE;
```

همچنین فرض کنید فرآیند P1 نیز به شکل همروند یا موازی با فرآیند P0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① C[1]= TRUE;
```

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین تابلوی وضعیت فرآیندها هر دو C[0] = TRUE و C[1] = TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. اما متغیر نوبت turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآیند، شماره‌ی فرآیند خود را در متغیر نوبت turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیره کند، فرآیندی است که شماره‌اش در متغیر نوبت turn باقی می‌ماند و دیگری اثرش در متغیر نوبت turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغییر turn گره خورده است، بنابراین فرآیندی که دیرتر متغیر نوبت turn را مقداردهی کرده است، باید صبر پیشه کند و متغیر نوبت turn را نگه‌داری کند و در حلقه‌ی انتظار بچرخد. و فرآیندی که زودتر متغیر نوبت turn را مقداردهی کرده است، وارد ناحیه بحرانی می‌شود.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتی که دو فرآیند به دستور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود اما فرآیند P1 باید در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P_0:
② turn= 1;

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_1:
② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

P_0:
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:
/*critical_section*/

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_0:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. خب هر دو باهم پشت ناحیه بحرانی خودشان مسدود نشدند. فرآیند اول توانست وارد ناحیه بحرانی خودش شود، اما فرآیند دوم نتوانست وارد ناحیه بحرانی خودش شود. بنابراین بن بست رخ نداده است.

فرم ساده قانون دوم ارسطو (آزمون بن بست)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هر دو باهم نتونستن به طور همزمان وارد باجه تلفن همگانی بشن و هردو باهم پشت در باجه تلفن

همگانی مسدود شدن اونوقت بن بست رخ داده. اخلاق دیگه اینجا چیزی نمی‌گه و سکوت می‌کنه، چون دیگه بن‌بست شده!

توجه: برای برقرار بودن شرط انتظار محدود باید **قانون دوم ارسطو** (آزمون بن بست) و **قانون چهارم ارسطو** (آزمون گرسنگی) هر دو باهم برقرار باشند. بنابراین شرط انتظار محدود در سوال مطرح شده برقرار است.

صورت سوال به این شکل است:

کدام یک از گزینه‌های زیر درست **نیست**؟

۱) این راه حل همه شرایط ناحیه بحرانی را برآورده می‌کند.

گزینه اول **نادرست** است، زیرا شرط پیشرفت برقرار نیست و برآورده نمی‌کند.

۲) این راه حل تنها شرط انتظار محدود را برآورده می‌کند.

گزینه دوم **نادرست** است، زیرا شرط انحصار متقابل هم برقرار است و برآورده می‌کند.

۳) این راه حل تنها استفاده انحصاری از ناحیه بحرانی را برآورده می‌کند.

گزینه سوم **نادرست** است، زیرا شرط انتظار محدود هم برقرار است و برآورده می‌کند.

۴) این راه حل تنها شرط پیشرفت را برآورده می‌کند.

گزینه چهارم **نادرست** است، زیرا شرط پیشرفت اصلا برقرار نیست و برآورده نمی‌کند.

توجه: همانطور که واضح و مشخص هست، همه گزینه‌ها نادرست هستند و هر چهار گزینه می‌تواند پاسخ سوال باشد.

توجه: سازمان سنجش آموزش کشور، گزینه اول را به عنوان پاسخ نهایی اعلام کرده بود.

۱۰- گزینه (۴) صحیح است.

در راه حل سمافور اجرای دو عمل wait و signal باید به صورت اتمیک و تجزیه‌ناپذیر باشند. چرا که در این اعمال نیز باید **انحصار متقابل** رعایت شود و به عبارت دیگر، در هر زمان فقط یک فرآیند حق دستکاری شمارنده سمافور را با یکی از دو عمل مزبور داشته باشد. راه حل معمولی این است که wait و signal را به صورت فراخوان سیستمی پیاده‌سازی کنیم. آن وقت هسته یک سیستم عامل تک پردازنده می‌تواند به راحتی تمام وقفه‌ها را در ابتدای رویه‌های wait و signal از کار بیندازد و پس از انجام رویه دوباره فعال سازد. از آن‌جا که تعداد دستورالعمل‌های هر یک از دو فراخوان اندک‌اند، از کار انداختن وقفه‌ها هیچ اشکالی را به وجود نخواهد آورد. اما این راه حل در سیستم‌های تک پردازنده‌ای کارآمد خواهد بود، زیرا هر فرآیند قادر به از کار انداختن وقفه‌های پردازنده جاری خود خواهد بود، و ممکن است فرآیند دیگری، توسط پردازنده دیگری، همزمان با فرآیند اول پس از عبور تقریباً همزمان از خطوط تابع wait وارد ناحیه بحرانی گردد. بنابراین در سیستم‌های چند پردازنده‌ای، به دنبال راه حل دیگری باید بود. یک راه حل این است که شمارنده

سمافور، یا در وسعت بزرگ‌تر کل الگوریتم تابع wait به عنوان یک ناحیه بحرانی، در بخش ناحیه بحرانی راه حل TSL قرار گیرد. از آن‌جا که راه حل TSL گذرگاه حافظه را مسدود می‌کند، بنابراین مطابق شرح احوال الگوریتم TSL، این راه حل گارانتی می‌کند که در هر لحظه فقط یک پردازنده به شمارنده سمافور دسترسی داشته باشد. در راه حل TSL مشکل انتظار مشغول وجود دارد.

همانطور که گفتیم در راه حل سمافور دو عمل wait و signal باید به صورت اتمیک و تجزیه ناپذیر باشند. چراکه آنچه اهمیت دارد، برقراری شرط انحصار متقابل در پیاده‌سازی دو عمل wait و signal است.

ساختار کلی این راه‌حل به صورت زیر می‌باشد:

```
wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();
```

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

توجه: دو تابع wait (mutex) و signal (mutex)، باید به صورت اتمیک (تجزیه‌ناپذیر) انجام گیرند. اتمیک بودن، تضمین می‌کند که از لحظه‌ای که یک عملیات بر روی شمارنده سمافور شروع می‌شود، هیچ فرآیند دیگری نتواند به سمافور دسترسی پیدا کند تا زمانی که آن عملیات به پایان برسد. اتمیک بودن این عملیات برای حل مسایل همگام‌سازی و کنترل شرایط رقابتی و به تبع برقراری شرط انحصار متقابل کاملاً ضروری و لازم است.

راه‌حل سمافور بر دو دسته‌ی کلی (۱) سمافور عمومی و (۲) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور عمومی می‌پردازیم:

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
Int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

برای برقراری شرط انحصار متقابل از این شمارنده با مقدار اولیه یک استفاده می‌گردد.

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله‌ی انتظار مشغول و حل مسأله‌ی اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این‌گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

بر روی سمافور عمومی s دو تابع wait(s) و signal(s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
```

توجه: راه‌حل سمافور و تابع اتمیک wait باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if (s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر اینصورت، فرآیند، وارد ناحیه‌ی بحرانی می‌گردد.

تابع signal(s): عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
signal (semaphore s)
{
s.count = s.count + 1
   if (s.count< = 0)
   {
    remove a process from queue;
    wake up ();
     }
}
```

توجه: راه‌حل سمافور و تابع اتمیک signal باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند علاقه‌مند به خروج از ناحیه‌ی بحرانی، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به ناحیه‌ی بحرانی که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up() بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند که از ناحیه‌ی بحرانی خارج می‌شود، با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

ساختار کلی از کار انداختن وقفه در سیستم **تک پردازنده‌ای**، به صورت زیر می‌باشد:

```
Disable_Interupts ();
Critical _section ();
Enable_ Interupts ();
remainder_section ();
```

ساده‌ترین راه‌حل، آن است که هر فرآیند بلافاصله پس از ورود به ناحیه‌ی بحرانی‌اش، تمام وقفه‌ها را از کار بیندازد و دقیقاً قبل از خروج از ناحیه‌ی بحرانی همه‌ی آن‌ها را مجدداً فعال سازد. با متوقف کردن وقفه‌ها، دیگر وقفه‌های ساعت نیز رخ نخواهد داد. پردازنده فقط در نتیجه‌ی وقوع وقفه‌های ساعت و یا وقفه‌های ورودی و خروجی است که می‌تواند از یک فرآیند به فرآیندی دیگر تعویض متن کند. پس با از کار انداختن وقفه‌ها، پردازنده دیگر تحت هیچ شرایطی قادر

نخواهد بود از فرآیندی به فرآیند دیگر تعویض متن کند. بنابراین هنگامی که یک فرآیند وقفه‌ها را غیر فعال می‌کند، می‌تواند بدون هیچ مشکلی و بدون ترس از مداخله‌ی دیگر فرآیندها به خواندن و نوشتن در حافظه‌ی مشترک بپردازد. پس برای رعایت شرط انحصار متقابل **در سیستم‌های تک‌پردازنده‌ای** کافی است، وقفه‌ها متوقف شوند.

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

```
wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();
```

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک پردازنده‌ای** به کمک از کار انداختن وقفه به صورت زیر می‌باشد:

```
Disable_Interupts ();
wait (semaphore)
    {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
Enable_ Interupts ();
critical_section ();
Disable_Interupts ();
signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
    if (s.count < = 0)
    {
     remove a process from queue;
     wake up ();
      }
}
Enable_ Interupts ();
remainder_section ();
```

توجه: از کار انداختن وقفه‌ها، در سیستم‌های تک‌پردازنده‌ای، اغلب در داخل خود سیستم عامل برای مدیریت نواحی بحرانی همچون اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal، تکنیک مفیدی است، اما برای مدیریت نواحی بحرانی فرآیندهای کاربر، به دلیل فراموش‌کار بودن کاربر، تکنیک مناسبی نیست.

توجه: در سیستم‌های چندپردازنده‌ای، غیرفعال کردن وقفه‌ها، فقط در پردازنده‌ای اثر دارد که دستورالعمل از کار انداختن وقفه را اجرا می‌کند. پس پردازنده‌های دیگر به کارشان ادامه می‌دهند. از آنجایی که ممکن است بیش از یک فرآیند در هر لحظه در حال اجرا باشد، ممکن است فرآیندهای دیگر که روی پردازنده‌های دیگر در حال اجرا هستند نیز وارد ناحیه‌ی بحرانی شوند. به عبارت دیگر، در سیستم‌های چندپردازنده‌ای، این ذات توازی است که باعث ورود همزمان فرآیندها به ناحیه‌ی بحرانی می‌شود و نه وقوع وقفه، که با جلوگیری از آن، بخواهیم مشکل را حل کنیم. پس در سیستم‌های چند پردازنده‌ای ممکن است شرط انحصار متقابل برقرار نباشد.

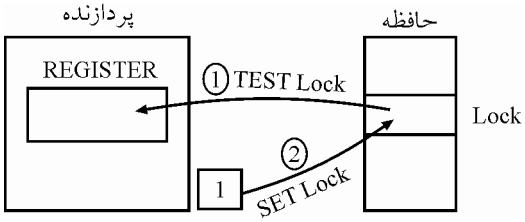
ساختار کلی دستورالعمل TSL در سیستم **تک و چند پردازنده‌ای**، به صورت زیر می‌باشد:

```
enter_section ();          enter_section              exit_section
critical_section ();        TSL REGISTER, LOCK         MOVE LOCK, 0
exit_section ();           CMP REGISTER, 0            RET
remainder_section ();       JNE enter_section
                           RET
```

بسیاری از پردازنده‌ها، دستورالعمل دو بخشی اما اتمیک خاصی دارند به نام TSL (TEST and SET Lock)، بدین نحو که این دستورالعمل به شکل تجزیه ناپذیر، عملیات زیر را انجام می‌دهد:

- ابتدا محتویات یک کلمه از حافظه به نام Lock را می‌خواند و مقدار آن را در رجیستر قرار می‌دهد. (TEST Lock)
- سپس مقدار یک را در متغیر Lock قرار می‌دهد. (SET Lock)

به شکل زیر توجه کنید:



سخت‌افزار تضمین می‌کند که دو عمل خواندن و مقداردهی متغیر قفل به صورت اتمیک (تجزیه ناپذیر) انجام شود. بدین شکل که هیچ فرآیند و حتی پردازنده‌ی دیگری نتواند به این متغیر قفل دسترسی پیدا کند تا وقتی که اجرای دستورالعمل به پایان برسد. پردازنده‌ای که دستورالعمل TSL

را اجرا می‌کند، گذرگاه حافظه را قفل می‌کند تا از دسترسی دیگر پردازنده‌ها به حافظه جلوگیری کند تا اینکه این دستورالعمل به پایان برسد.

توجه مهم: در این راه‌حل شرط ورود به ناحیه‌ی بحرانی، اجرای زودتر دستور TSL است.

در این الگوریتم، فرآیندها برای کسب اجازه و ورود به ناحیه‌ی بحرانی از تابع enter_section و برای خروج از ناحیه‌ی بحرانی از تابع exit_section استفاده می‌کنند. اگر فرآیندی علاقه‌مند به ورود به ناحیه‌ی بحرانی باشد. ابتدا تابع enter_section را صدا می‌زند تا بررسی‌های لازم جهت فراهم بودن یا نبودن ورود به ناحیه‌ی بحرانی انجام گردد. بدین نحو که ابتدا، توسط دستور TSL و به شکل اتمیک مقدار متغیر Lock در رجیستر قرار داده می‌شود، سپس مقدار متغیر Lock برابر یک مقداردهی می‌شود. سپس مقدار رجیستر که حاوی مقدار متغیر Lock می‌باشد با صفر مقایسه می‌شود. اگر مقدار رجیستر برابر صفر باشد به معنی خالی بودن ناحیه‌ی بحرانی است و در ادامه دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن مقدار رجیستر با صفر، انجام نمی‌شود، زیرا مقدار رجیستر برابر صفر است و در ادامه دستور RET (RETURN) باعث می‌شود تا تابع تمام شود. بنابراین فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، وارد ناحیه‌ی بحرانی می‌گردد. و پس از آنکه فرآیند کارش با ناحیه‌ی بحرانی تمام شد، تابع exit_section را به نشانه‌ی خروج از ناحیه‌ی بحرانی صدا می‌زند، اجرای این تابع سبب می‌شود تا مقدار متغیر Lock توسط دستور MOVE برابر صفر گردد، صفر بودن مقدار متغیر Lock به معنی خالی بودن ناحیه‌ی بحرانی است، بنابراین این امکان فراهم می‌شود تا فرآیندهای دیگر بتوانند پس از کسب اجازه توسط تابع enter_section وارد ناحیه‌ی بحرانی شوند. در طرف مقابل اگر در حین اجرای تابع enter_section، مقدار متغیر Lock برابر یک باشد، به معنی پر بودن ناحیه‌ی بحرانی است و در ادامه شرط دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن رجیستر با مقدار صفر برقرار است، زیرا مقدار رجیستر برابر یک است. بنابراین یک حلقه‌ی انتظار مشغول تا خالی شدن ناحیه‌ی بحرانی ایجاد می‌گردد. این حلقه‌ی انتظار مانع ورود فرآیند رقیب به ناحیه‌ی بحرانی می‌گردد.

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

```
wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();
```

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک و چند پردازنده‌ای** به کمک دستورالعمل TSL به صورت زیر می‌باشد:

```
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Wait (semaphore)
    {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
Add this process to s.queue;
block ();
  }
}
exit_section
MOVE LOCK, 0
RET
critical_section ();
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
    if (s.count< = 0)
    {
     Remove a process from queue;
     Wake up ();
      }
}
exit_section
MOVE LOCK, 0
```

RET
remainder_section ();

در این راه‌حل، مشکل **انتظار مشغول** وجود دارد، زیرا زمانی که مثلاً فرآیند P_0 در ناحیه‌ی بحرانی قرار دارد، فرآیند P_1 در یک حلقه‌ی انتظار زمان پردازنده را در بررسی برقراری شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی به هدر می‌رود.

توجه: این راه‌حل، فقط در پردازنده‌هایی قابل اجراست که دستورالعمل TSL را پشتیبانی می‌کنند.

توجه: پردازنده Intel، دستور TSL را پشتیبانی نمی‌کند و از دستور SWAP پشتیبانی می‌کند.

۱۱- گزینه (۱) صحیح است.

در راه حل سمافور اجرای دو عمل wait و signal باید به صورت اتمیک و تجزیه‌ناپذیر باشند. چرا که در این اعمال نیز باید **انحصار متقابل** رعایت شود و به عبارت دیگر، در هر زمان فقط یک فرآیند حق دستکاری شمارنده سمافور را با یکی از دو عمل مزبور داشته باشد. راه حل معمولی این است که wait و signal را به صورت فراخوان سیستمی پیاده‌سازی کنیم. آن وقت هسته یک سیستم عامل تک پردازنده می‌تواند به راحتی تمام وقفه‌ها را در ابتدای رویه‌های wait و signal از کار بیندازد و پس از انجام رویه دوباره فعال سازد. از آن‌جا که تعداد دستورالعمل‌های هر یک از دو فراخوان اندک‌اند، از کار انداختن وقفه‌ها هیچ اشکالی را به وجود نخواهد آورد. اما این راه حل در سیستم‌های تک پردازنده‌ای کارآمد خواهد بود، زیرا هر فرآیند قادر به از کار انداختن وقفه‌های پردازنده جاری خود خواهد بود، و ممکن است فرآیند دیگری، توسط پردازنده دیگری، همزمان با فرآیند اول پس از عبور تقریباً همزمان از خطوط تابع wait وارد ناحیه بحرانی گردد. بنابراین در سیستم‌های چند پردازنده‌ای، به دنبال راه حل دیگری باید بود. یک راه حل این است که شمارنده سمافور، یا در وسعت بزرگ‌تر کل الگوریتم تابع wait به عنوان یک ناحیه بحرانی، در بخش ناحیه بحرانی راه حل TSL قرار گیرد. از آن‌جا که راه حل TSL گذرگاه حافظه را مسدود می‌کند، بنابراین مطابق شرح احوال الگوریتم TSL، این راه حل گارانتی می‌کند که در هر لحظه فقط یک پردازنده به شمارنده سمافور دسترسی داشته باشد. در راه حل TSL مشکل انتظار مشغول وجود دارد.

همانطور که گفتیم در راه حل سمافور دو عمل wait و signal باید به صورت اتمیک و تجزیه ناپذیر باشند. چراکه آنچه اهمیت دارد، برقراری شرط انحصار متقابل در پیاده‌سازی دو عمل wait و signal است.

ساختار کلی این راه‌حل به صورت زیر می‌باشد:

wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

توجه: دو تابع wait (mutex) و signal (mutex)، باید به صورت اتمیک (تجزیه‌ناپذیر) انجام گیرند. اتمیک بودن، تضمین می‌کند که از لحظه‌ای که یک عملیات بر روی شمارنده سمافور شروع می‌شود، هیچ فرآیند دیگری نتواند به سمافور دسترسی پیدا کند تا زمانی که آن عملیات به پایان برسد. اتمیک بودن این عملیات برای حل مسایل همگام‌سازی و کنترل شرایط رقابتی و به تبع برقراری شرط انحصار متقابل کاملاً ضروری و لازم است.

راه‌حل سمافور بر دو دسته‌ی کلی(۱) سمافور عمومی و (۲) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور عمومی می‌پردازیم:

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
Int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

برای برقراری شرط انحصار متقابل از این شمارنده با مقدار اولیه یک استفاده می‌گردد.

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله‌ی انتظار مشغول و حل مسأله‌ی اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این‌گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

بر روی سمافور عمومی s دو تابع wait(s) و signal(s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
```

توجه: راه‌حل سمافور و تابع اتمیک wait باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if (s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر اینصورت، فرآیند، وارد ناحیه‌ی بحرانی می‌گردد.

تابع signal(s) : عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
signal (semaphore s)
{
s.count = s.count + 1
   if (s.count< = 0)
   {
    remove a process from queue;
    wake up ();
     }
}
```

توجه: راه‌حل سمافور و تابع اتمیک signal باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند علاقه‌مند به خروج از ناحیه‌ی بحرانی، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به ناحیه‌ی بحرانی که در حال حاضر در صف سمافور قرار

دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up() بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند که از ناحیه‌ی بحرانی خارج می‌شود، با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

ساختار کلی از کار انداختن وقفه در سیستم **تک پردازنده‌ای**، به صورت زیر می‌باشد:

```
Disable_Interupts ();
Critical _section ();
Enable_ Interupts ();
remainder_section ();
```

ساده‌ترین راه‌حل، آن است که هر فرآیند بلافاصله پس از ورود به ناحیه‌ی بحرانی‌اش، تمام وقفه‌ها را از کار بیندازد و دقیقاً قبل از خروج از ناحیه‌ی بحرانی همه‌ی آن‌ها را مجدداً فعال سازد. با متوقف کردن وقفه‌ها، دیگر وقفه‌های ساعت نیز رخ نخواهد داد. پردازنده فقط در نتیجه‌ی وقوع وقفه‌های ساعت و یا وقفه‌های ورودی و خروجی است که می‌تواند از یک فرآیند به فرآیندی دیگر تعویض متن کند. پس با از کار انداختن وقفه‌ها، پردازنده دیگر تحت هیچ شرایطی قادر نخواهد بود از فرآیندی به فرآیند دیگر تعویض متن کند. بنابراین هنگامی که یک فرآیند وقفه‌ها را غیر فعال می‌کند، می‌تواند بدون هیچ مشکلی و بدون ترس از مداخله‌ی دیگر فرآیندها به خواندن و نوشتن در حافظه‌ی مشترک بپردازد. پس برای رعایت شرط انحصار متقابل **در سیستم‌های تک‌پردازنده‌ای** کافی است، وقفه‌ها متوقف شوند.

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

```
wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();
```

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک پردازنده‌ای** به کمک از کار انداختن وقفه به صورت زیر می‌باشد:

```
Disable_Interupts ();
wait (semaphore)
   {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
Enable_ Interupts ();
critical_section ();
Disable_Interupts ();
signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
   if (s.count < = 0)
   {
    remove a process from queue;
    wake up ();
     }
}
Enable_ Interupts ();
remainder_section ();
```

توجه: از کار انداختن وقفه‌ها، در سیستم‌های تک‌پردازنده‌ای، اغلب در داخل خود سیستم عامل برای مدیریت نواحی بحرانی همچون اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal، تکنیک مفیدی است، اما برای مدیریت نواحی بحرانی فرآیندهای کاربر، به دلیل فراموش‌کار بودن کاربر، تکنیک مناسبی نیست.

توجه: در سیستم‌های چندپردازنده‌ای، غیرفعال کردن وقفه‌ها، فقط در پردازنده‌ای اثر دارد که دستورالعمل از کار انداختن وقفه را اجرا می‌کند. پس پردازنده‌های دیگر به کارشان ادامه می‌دهند. از آنجایی که ممکن است بیش از یک فرآیند در هر لحظه در حال اجرا باشد، ممکن است فرآیندهای دیگر که روی پردازنده‌های دیگر در حال اجرا هستند نیز وارد ناحیه‌ی بحرانی شوند. به

عبارت دیگر، در سیستم‌های چندپردازنده‌ای، این ذات توازی است که باعث ورود همزمان فرآیندها به ناحیه‌ی بحرانی می‌شود و نه وقوع وقفه، که با جلوگیری از آن، بخواهیم مشکل را حل کنیم. پس در سیستم‌های چند پردازنده‌ای ممکن است شرط انحصار متقابل برقرار نباشد.

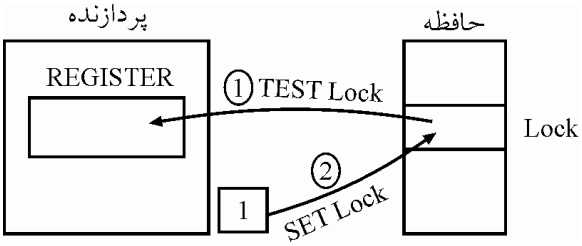
ساختار کلی دستورالعمل TSL در سیستم **تک و چند پردازنده‌ای**، به صورت زیر می‌باشد:

```
enter_section ();          enter_section              exit_section
critical_section ();       TSL REGISTER, LOCK         MOVE LOCK, 0
exit_section ();           CMP REGISTER, 0            RET
remainder_section ();      JNE enter_section
                           RET
```

بسیاری از پردازنده‌ها، دستورالعمل دو بخشی اما اتمیک خاصی دارند به نام TSL (TEST and SET Lock)، بدین نحو که این دستورالعمل به شکل تجزیه ناپذیر، عملیات زیر را انجام می‌دهد:

- ابتدا محتویات یک کلمه از حافظه به نام Lock را می‌خواند و مقدار آن را در رجیستر قرار می‌دهد. (TEST Lock)
- سپس مقدار یک را در متغیر Lock قرار می‌دهد. (SET Lock)

به شکل زیر توجه کنید:



سخت‌افزار تضمین می‌کند که دو عمل خواندن و مقداردهی متغیر قفل به صورت اتمیک (تجزیه ناپذیر) انجام شود. بدین شکل که هیچ فرآیند و حتی پردازنده‌ی دیگری نتواند به این متغیر قفل دسترسی پیدا کند تا وقتی که اجرای دستورالعمل به پایان برسد. پردازنده‌ای که دستورالعمل TSL را اجرا می‌کند، گذرگاه حافظه را قفل می‌کند تا از دسترسی دیگر پردازنده‌ها به حافظه جلوگیری کند تا اینکه این دستورالعمل به پایان برسد.

توجه مهم: در این راه‌حل شرط ورود به ناحیه‌ی بحرانی، اجرای زودتر دستور TSL است.

در این الگوریتم، فرآیندها برای کسب اجازه و ورود به ناحیه‌ی بحرانی از تابع enter_section و برای خروج از ناحیه‌ی بحرانی از تابع exit_section استفاده می‌کنند. اگر فرآیندی علاقه‌مند به

ورود به ناحیه‌ی بحرانی باشد. ابتدا تابع enter_section را صدا می‌زند تا بررسی‌های لازم جهت فراهم بودن یا نبودن ورود به ناحیه‌ی بحرانی انجام گردد. بدین نحو که ابتدا، توسط دستور TSL و به شکل اتمیک مقدار متغیر Lock در رجیستر قرار داده می‌شود، سپس مقدار متغیر Lock برابر یک مقداردهی می‌شود. سپس مقدار رجیستر که حاوی مقدار متغیر Lock می‌باشد با صفر مقایسه می‌شود. اگر مقدار رجیستر برابر صفر باشد به معنی خالی بودن ناحیه‌ی بحرانی است و در ادامه دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن مقدار رجیستر با صفر، انجام نمی‌شود، زیرا مقدار رجیستر برابر صفر است و در ادامه دستور RET (RETURN) باعث می‌شود تا تابع تمام شود. بنابراین فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، وارد ناحیه‌ی بحرانی می‌گردد. و پس از آنکه فرآیند کارش با ناحیه‌ی بحرانی تمام شد، تابع exit_section را به نشانه‌ی خروج از ناحیه‌ی بحرانی صدا می‌زند، اجرای این تابع سبب می‌شود تا مقدار متغیر Lock توسط دستور MOVE برابر صفر گردد، صفر بودن مقدار متغیر Lock به معنی خالی بودن ناحیه‌ی بحرانی است، بنابراین این امکان فراهم می‌شود تا فرآیندهای دیگر بتوانند پس از کسب اجازه توسط تابع enter_section وارد ناحیه‌ی بحرانی شوند. در طرف مقابل اگر در حین اجرای تابع enter_section، مقدار متغیر Lock برابر یک باشد، به معنی پر بودن ناحیه‌ی بحرانی است و در ادامه شرط دستور JNE به معنی پرش به ابتدای تابع به شرط برابر نبودن رجیستر با مقدار صفر برقرار است، زیرا مقدار رجیستر برابر یک است. بنابراین یک حلقه‌ی انتظار مشغول تا خالی شدن ناحیه‌ی بحرانی ایجاد می‌گردد. این حلقه‌ی انتظار مانع ورود فرآیند رقیب به ناحیه‌ی بحرانی می‌گردد.

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

```
wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();
```

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک و چند پردازنده‌ای** به کمک دستورالعمل TSL به صورت زیر می‌باشد:

```
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Wait (semaphore)
    {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
Add this process to s.queue;
block ();
  }
}
exit_section
MOVE LOCK, 0
RET
critical_section ();
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
    if (s.count< = 0)
    {
     Remove a process from queue;
     Wake up ();
      }
}
exit_section
MOVE LOCK, 0
RET
remainder_section ();
```

در این راه‌حل، مشکل **انتظار مشغول** وجود دارد، زیرا زمانی که مثلاً فرآیند P_0 در ناحیه‌ی بحرانی قرار دارد، فرآیند P_1 در یک حلقه‌ی انتظار زمان پردازنده را در بررسی برقراری شروط لازم و کافی برای ورود به ناحیه‌ی بحرانی به هدر می‌رود.

توجه: این راه‌حل، فقط در پردازنده‌هایی قابل اجراست که دستورالعمل TSL را پشتیبانی می‌کنند.

توجه: پردازنده Intel، دستور TSL را پشتیبانی نمی‌کند و از دستور SWAP پشتیبانی می‌کند.

۱۲- گزینه ( ) صحیح است.

	Thread 1		Thread 2
(1)	if $(a<b)$ then	(4)	$b=10$
(2)	$c=b-a$	(5)	$c=-3$
(3)	$c=b+a$		

مطابق فرض سؤال مقادیر اولیه a برابر 4، b برابر صفر و c برابر صفر است. بنابراین ترتیب‌های زیر را داریم:

ترتیب اجرا	مقدار c در انتها	گزینه‌ی نادرست
$(4)\rightarrow(5)\rightarrow(1)\rightarrow(2)$	6	-
$(1)\rightarrow(3)\rightarrow(4)\rightarrow(5)$	-3	-
$(1)\rightarrow(4)\rightarrow(5)\rightarrow(3)$	14	1 و 2
$(4)\rightarrow(1)\rightarrow(5)\rightarrow(2)$	6	-

بنابراین، گزینه‌ی سوم یا چهارم درست است، سایر مقادیر قابل تولید نیست!

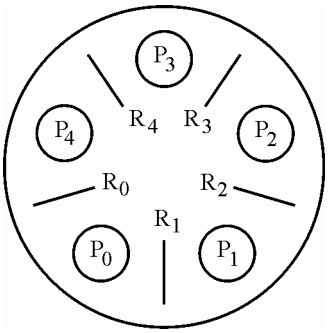
توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه چهارم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

۱۳- گزینه (۲) صحیح است.

حل مسأله فیلسوفان خورنده توسط سمافور

5 فیلسوف زندگی خود را صرف فکر کردن و خوردن کرده‌اند. آن‌ها دور یک میز دایره‌ای با 5 بشقاب و 5 عدد چنگال نشسته‌اند. هر فیلسوف برای غذا خوردن حتماً باید دو چنگال در دست داشته باشد. بین هر جفت بشقاب روی میز، یک چنگال وجود دارد. هنگامی که فیلسوفی در حال فکر کردن است با بقیه هیچ ارتباطی ندارد. هر از گاهی فیلسوف احساس گرسنگی کرده و سعی می‌کند، چنگال‌های سمت راست و چپش را یکی یکی و با هر ترتیب ممکن بردارد، اگر موفق به برداشتن هر دو چنگال شود برای مدتی غذا خورده و دوباره چنگال‌ها را پایین گذاشته و به فکر

کردن ادامه می‌دهد. فیلسوف مجاز است که در هر بار فقط یک چنگال را بردارد و همچنین نمی‌تواند چنگالی که دست فیلسوف دیگری است را به زور بگیرد.



راه حل اول

از آنجایی که هر فیلسوفی که در حال خوردن است، نمی‌تواند چنگال‌های در اختیارش را به فیلسوف مجاور بدهد، لازم است برای هر چنگال شرط انحصار متقابل برقرار باشد. لذا برای هر چنگال یک شمارنده سمافور تعریف می‌شود. مسأله فیلسوفان خورنده در راه حل اول بر دو نوع (1) چپگرد و (2) راستگرد می‌باشد.

راه حل چپگرد: فیلسوفان ابتدا چنگال سمت چپ را بر می‌دارند.

ساختار کلی این راه حل به صورت زیر می‌باشد:

توجه: قطعه کد زیر را شبه کد فرض کنید.

```
#define N 5
semaphor fork[5]={1};
void philosopher (int i)
{
  while (TRUE)
  {
    wait (fork [i]);
    wait(fork [(i+1)%N]);
    eat( );
    signal(fork[i]);
    signal (fork [(i+1)%N]);
    think( );
      }
   }
```

برای هر چنگال (R_i)، یک شمارنده‌ی سمافور با مقدار اولیه یک تعریف شده است، مطابق الگوریتم فوق هر فیلسوفی که می‌خواهد تغذیه کند، باید بتواند ابتدا چنگال چپ خود را بردارد، این کار با دستور wait(fork[i]) انجام می‌گیرد. سپس با اجرای دستور wait(fork[(i+1)%N]) باید بتواند چنگال راست خود را بردارد. اگر فیلسوفی موفق به انجام این دو عمل شد، می‌تواند خوردن را شروع کند.

اگرچه این راه حل تضمین می‌کند که هیچ دو همسایه‌ای همزمان غذا نخورند، ولی این روش ممکن است دچار بن‌بست شود.

فرض کنید که هر پنج فیلسوف همزمان گرسنه شده و هر کدام چنگال سمت چپ خود را بردارند (همه‌ی 5 فیلسوف خط اول یعنی ; (wait (fork [i]) را اجرا کنند). بنابراین تمام عناصر آرایه fork[5] برابر با صفر می‌شوند. هنگامی که فیلسوفان سعی می‌کنند تا چنگال سمت راست خود را بردارند ;(wait(fork [(i+1)%5])، یک بن‌بست به وجود می‌آید.

شرایط بن‌بست را به یاد آورید:

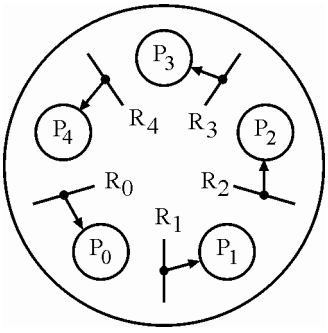
۱- انحصار متقابل (برقرار است، توسط تعریف شمارنده سمافور برای هر چنگال (منبع بحرانی))

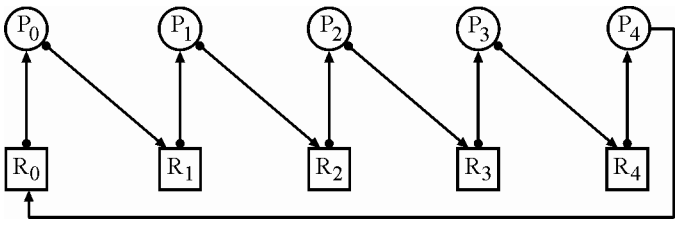
۲- انحصاری بودن (برقرار است، نمی‌توان چنگال (منبع بحرانی) را به زور پس گرفت).

۳- نگهداری و انتظار (برقرار است.)

۴- سیکل انتظار چرخشی (برقرار است.)

به شکل بازسازی شده‌ی فیلسوفان خورنده در شرایط بن‌بست توجه کنید:





راه حل راستگرد: فیلسوفان ابتدا چنگال سمت راست را برمی‌دارند.

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
#define N 5
semaphore fork[5]={1};
void philsopher (int i)
  {
    while (TRUE)
        {
        wait(fork[(i+1)%N]);
        wait(fork[i]);
        eat ( );
        signal(fork[(i+1)%N]);
        signal (fork[i]) ;
        think ( );
        }
}
```

توجه: کلیه تعاریف، توضیحات و خصوصیاتی که برای راه حل چپگرد گفته شد، به شکل بالعکس برای راه حل راستگرد نیز برقرار است.

توجه: حال اگر در راه حل چپگرد حداقل یک فیلسوف راست دست و یا در راه حل راستگرد حداقل یک فیلسوف چپ دست وجود داشته باشد، شرط سیکل نقض می‌گردد، بنابراین هیچ‌گاه بن‌بست رخ نمی‌دهد.

توجه: مطابق فرض سوال، با وجود حداقل یک فیلسوف چپ دست و یک فیلسوف راست دست، بنابراین هر نوع راه حلی، اعم از چپگرد یا راستگرد پیش رود، از آن‌جا که حداقل یک فیلسوف در جهت مقابل قرار دارد، بنابراین شرایط نقض شرط سیکل، ایجاد می‌گردد، که همین عامل باعث می‌شود، هیچ‌گاه بن‌بست رخ ندهد.

۱۴- گزینه (۲) صحیح است.

راه حل سمافور در سال 1965 توسط Dijkstra پیشنهاد شده است.

ساختار کلی این راه حل به صورت زیر می‌باشد:

```
wait(mutex);
critical_section();
signal(mutex);
remainder_section();
```

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

توجه: دو تابع wait (mutex) و signal (mutex)، باید به صورت اتمیک (تجریه ناپذیر) انجام گیرند. اتمیک بودن، تضمین می‌کند که از لحظه‌ای که یک عملیات بر روی شمارنده سمافور شروع

می‌شود، هیچ فرآیند دیگری نتواند به سمافور دسترسی پیدا کند تا زمانی که آن عملیات به پایان برسد. اتمیک بودن این عملیات برای حل مسایل همگام‌سازی و کنترل شرایط رقابتی و به تبع برقراری شرط انحصار متقابل کاملاً لازم و ضروری است.

توجه: در مقاله Dijkstra، از نام‌های P و V (حرف اول کلمات آلمانی تست "probern" و افزایش "Verhogen") به ترتیب به جای wait و signal استفاده شده بود و همچنین در سایر متون، از نام‌های down و up به ترتیب برای این دو استفاده می‌کنند. در همان متون گاهاً به جای نام‌های down و up، به ترتیب از عبارت‌های mutex_lock و mutex_unlock نیز استفاده شده است.

راه حل سمافور بر دو دسته کلی (1) سمافور عمومی و (2) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور دودویی می‌پردازیم:

سمافور دودویی

تنها تفاوت سمافور دودویی و سمافور عمومی در نحوه‌ی تعریف توابع wait و singal است.

تابع wait(s): عملیات آن به ترتیب شامل، تست کردن مقدار شمارنده، کاهش مقدار شمارنده (اما در نهایت فقط تا مقدار صفر) و احیاناً خواباندن یک فرآیند.

ساختار این تابع به صورت زیر است:

```
wait(semaphore s)
{
        if(s.count == 1)
        s.count = 0
        else
        {
        add this process to s.queue;
        block ();
        }
}
```

تابع signal (s): عملیات آن به ترتیب شامل تست خالی بودن صف، افزایش مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

```
signal (semaphore s)
{
        if(s.queue is empty)
        s.count = 1
        else
        {
        remove a process from queue;
        wake up ();
        }
}
```

توجه: در سمافور دودویی، شمارنده سمافور، هیچگاه منفی نمی‌شود و در هر شرایطی فقط می‌تواند دو مقدار باینری یا دودویی 0 یا 1 را داشته باشد، برای اثبات، مجدداً به تعاریف توابع wait و signal در سمافور دودویی دقت کنید.

توجه: هر فرآیندی که از ناحیه‌ی بحرانی خارج شود با اجرای تابع signal(s) فرآیند ابتدای صف را بیدار می‌کند (تغییری در شمارنده سمافور ایجاد نشده و برابر مقدار صفر باقی می‌ماند) و اگر هیچ فرآیند منتظری درون صف سمافور وجود نداشته باشد، مقدار شمارنده سمافور به یک مقداردهی می‌شود.

توجه: در این سوال تابع release معادل تابع signal در نظر گرفته شده است.

با توجه به مقادیر شمارنده‌های سمافور مطرح شده در سوال، $S_0 = 1$، $S_1 = 0$ و $S_2 = 0$، فرآیند P_0 ابتدا اجرا می‌شود.

توجه: فرآیندهای P_1 و P_2 فاقد حلقه هستند، و توسط شرایطی که فرآیند P_0 برای فرآیندهای P_1 و P_2 فراهم می‌کند، فرآیندهای P_1 و P_2 فقط و فقط می‌توانند **یکبار** اجرا شوند و تمام شوند.

سناریوی اول: (دو بار چاپ '0')

ابتدا فرآیند P_0 اجرا می‌شود و روی شمارنده سمافور S_0، عمل $wait(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $release(S_1)$ و $release(S_2)$ مقدار شمارنده سمافورهای S_1 و S_2 را برابر یک می‌کند.

حال فرآیندهای P_1 و P_2 می‌توانند به هر ترتیبی (اول P_1 بعد P_2 و یا اول P_2 بعد P_1)، با توجه به مقدار شمارنده سمافورهای S_1 و S_2، اجرا شوند. ولی از آنجا که شمارنده سمافور دودویی است، در انتهای اجرای دو فرآیند P_1 و P_2، حداکثر مقدار شمارنده سمافور دودویی S_0 به واسطه دو بار اجرای دستور $release(s_0)$ برابر یک خواهد بود. اگر سمافور S عمومی بود، مقدار شمارنده سمافور S_0 برابر 2 می‌بود، اما فرض سوال سمافور دودویی است!

در این لحظه فرآیندهای P_1 و P_2 تمام شده‌اند و دیگر اجرا نمی‌شوند، از آنجا که در انتهای کار فرآیندهای P_1 و P_2 مقدار شمارنده سمافور S_0، برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یک بار دیگر اجرا شود و دومین '0' را نیز چاپ کند. در ادامه فرآیند P_0، با اجرای دستور $wait(S_0)$، با سرنوشتی که دارد، برای همیشه می‌خوابد.

دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ دومین '0'، تغییری در روند اجرای کار ندارند، چون فرآیندهای P_1 و P_2 دیگر نیستند و تمام شده‌اند.

سناریوی دوم: (سه بار چاپ '0')

ابتدا فرآیند P_0 اجرا می‌شود و روی شمارنده سمافور S_0، عمل $wait(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $release(S_1)$ و $release(S_2)$ مقدار شمارنده سمافورهای S_1 و S_2 را برابر یک می‌کند.

حال اگر فرآیند P_1، با توجه به مقدار شمارنده سمافور S_1 اجرا شود، در انتهای اجرای فرآیند P_1، مقدار شمارنده سمافور دودویی S_0 به واسطه $release(S_0)$ برابر یک خواهد شد. در این لحظه فرآیند P_1 تمام شده است و دیگر اجرا نمی‌شود. از آنجا که در انتهای کار فرآیند P_1، مقدار شمارنده سمافور S_0 برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یکبار دیگر اجرا شود و دومین '0' را چاپ کند. دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ دومین '0'، تغییری در روند اجرای کار ندارد، چون فرآیند P_1، دیگر نیست و تمام شده است و برای فرآیند P_2 نیز مقدار شمارنده سمافور S_2 قبلاً و بعد از چاپ اولین '0' برابر یک شده است و چون شمارنده سمافور دودویی است، انجام دستور $release(S_2)$ بعد از چاپ دومین '0' تأثیری در مقدار شمارنده سمافور S_2 نخواهد داشت و در همان مقدار یک باقی خواهد ماند.

توجه: دقت کنید که حداکثر مقدار شمارنده سمافور دودویی برابر یک می‌باشد.

حال اگر فرآیند P_2، با توجه به مقدار شمارنده سمافور S_2، اجرا شود، در انتهای اجرای فرآیند P_2، مقدار شمارنده سمافور دودویی S_0 به واسطه $release(S_0)$ برابر یک خواهد شد. در این لحظه فرآیند P_2 تمام شده است و دیگر اجرا نمی‌شود. از آنجا که در انتهای کار فرآیند P_2 مقدار شمارنده سمافور S_0 برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یکبار دیگر اجرا شود و سومین '0' را چاپ کند. دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ سومین '0'، تغییری در روند اجرای کار ندارد، چون فرآیندهای P_1 و P_2 تمام شده‌اند و دیگر اجرا نمی‌شوند.

در ادامه، فرآیند P_0، با اجرای دستور $wait(S_0)$، با سرنوشتی که دارد، برای همیشه می‌خوابد.

بنابراین گزینه دوم درست خواهد بود.

۱۵- گزینه (۴) صحیح است.

راه‌حل سمافور در سال ۱۹۶۵ توسط Dijkstra پیشنهاد شده است.

ساختار کلی این راه‌حل به صورت زیر می‌باشد:

```
wait (mutex);
critical_section ();
signal (mutex);
remainder_section ();
```

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

توجه: دو تابع Wait (mutex) و signal (mutex)، باید به صورت اتمیک (تجزیه‌ناپذیر) انجام گیرند. اتمیک بودن، تضمین می‌کند که از لحظه‌ای که یک عملیات بر روی شمارنده سمافور شروع می‌شود، هیچ فرآیند دیگری نتواند به سمافور دسترسی پیدا کند تا زمانی که آن عملیات به پایان برسد. اتمیک بودن این عملیات برای حل مسایل همگام‌سازی و کنترل شرایط رقابتی و به تبع برقراری شرط انحصار متقابل کاملاً لازم و ضروری است.

توجه: در مقاله Dijkstra، از نام‌های P و V (حرف اول کلمات آلمانی تست "Probern" و افزایش "Verhogen") به ترتیب به جای wait و signal استفاده شده بود و همچنین در سایر متون، از نام‌های down و up به ترتیب برای این دو استفاده می‌کنند. در همان متون گاهاً به جای نام‌های down و up، به ترتیب از عبارت‌های mutex_lock و mutex_unlock نیز استفاده شده است.

راه‌حل سمافور بر دو دسته‌ی کلی (۱) سمافور عمومی و (۲) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور عمومی می‌پردازیم:

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
Int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

برای برقراری شرط انحصار متقابل از این شمارنده با مقدار اولیه یک استفاده می‌گردد.

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله‌ی انتظار مشغول و حل مسأله‌ی اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این‌گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

بر روی سمافور عمومی s دو تابع wait(s) و signal(s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
add  this process to s.queue;
block ();
  }
}
```

توجه: راه‌حل سمافور و تابع اتمیک wait باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if (s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر اینصورت، فرآیند، وارد ناحیه‌ی بحرانی می‌گردد.

توجه: در سمافور عمومی، وقتی شمارنده سمافور، مقدار منفی دارد، قدر مطلق این مقدار، معرف تعداد فرآیندهای بلوکه شده در صف سمافور است.

تابع signal(s) : عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
signal (semaphore s)
{
s.count = s.count + 1
   if (s.count  < = 0)
   {
    remove a process from queue;
    wake up ();
     }
}
```

توجه: راه‌حل سمافور و تابع اتمیک signal باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه سازی کرد.

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند علاقه‌مند به خروج از ناحیه‌ی بحرانی، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به ناحیه‌ی بحرانی که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up() بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

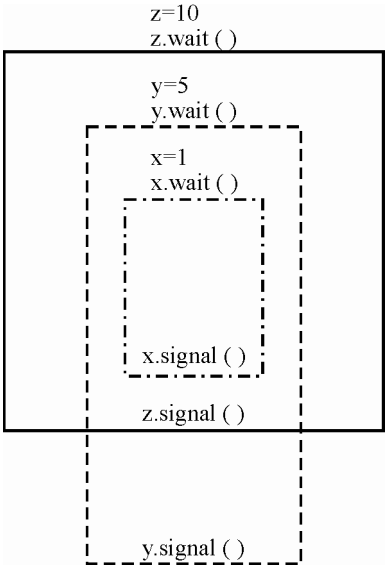
به بیان دیگر هر فرآیند که از ناحیه‌ی بحرانی خارج شود، با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

در صورت سوال مطرح شده است که سه سمافور با مقدار اولیه x=1، y=5 و z=10 در نظر گرفته شود. همچنین گفته شده است که قطعه کد زیر توسط 20 پردازه (process) اجرا می‌شود.

```
. . .
z.wait();
…
y.wait();
. . .
x.wait();
…
x.signal();
. . .
z.signal ();
…
y.signal ();
```

در ادامه خواسته سوال این است که، حداکثر طول صفی که برای سمافور y تشکیل می‌شود، چقدر است؟

شکل زیر را در نظر بگیرید:



دقت کنید که در صورت سوال مطرح شده است که قطعه کـد فـوق توسـط 20 پـردازه (process) اجرا می‌شود. پس می‌بایست قطعه کد فوق توسط 20 فرآیند اجرا گردد و از آن عبور کنند.

20 فرآیند P_1، P_2، P_3، P_4، P_5، P_6، P_7، P_8، P_9، P_{10}، P_{11}، P_{12}، P_{13} ، P_{14}، P_{15}، P_{16}، P_{17}، P_{18}، P_{19} و P_{20} را در نظر بگیرید. از این تعداد با توجه به شمارنده سمافور z که برابر مقـدار 10 اسـت، ابتـدا فرآیندهای P_1، P_2، P_3، P_4، P_5، P_6، P_7، P_8، P_9 و P_{10} به ترتیب اجرا می‌شوند و از سمافور z عبـور مـی‌کننـد. و فرآینـدهای P_{11}، P_{12}، P_{13} ، P_{14}، P_{15}، P_{16}، P_{17}، P_{18}، P_{19} و P_{20} پشـت سـمافور z در صف سمافور z آرام می‌گیرند و می‌خوابند. در ادامه از بـین فرآینـدهای عبـور کـرده از سـمافور z یعنی فرآیندهای P_1، P_2، P_3، P_4، P_5، P_6، P_7، P_8، P_9 و P_{10} با توجه به شمارنده سمافور y که برابـر مقدار 5 است، فقط فرآیندهای P_1، P_2، P_3، P_4 و P_5 این شانس را پیـدا مـی‌کننـد کـه از سـمافورy عبور کنند و عبور می‌کنند، و فرآیندهای P_6، P_7، P_8، P_9 و P_{10} پشت سمافور y در صف سمافور y آرام می‌گیرند و می‌خوابند. پس درحال حاضر 5 فرآیند در صف سـمافور y بـه خـواب رفتـه‌انـد، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار5 است. در ادامـه از بین فرآیندهای عبور کرده از سمافور y یعنی فرآیندهای P_1، P_2، P_3، P_4 وP_5 با توجـه بـه شـمارنده سمافور x که برابر مقدار 1 است، فقط فرآیند P_1 این شانس را پیدا می‌کند کـه از سـمافورx عبـور کند و عبـور مـی‌کنـد، و فرآینـدهای P_2، P_3، P_4 وP_5 پشـت سـمافور x در صـف سـمافور x آرام می‌گیرند و می‌خوابند. پس درحال حاضر 4 فرآیند در صف سمافور x به خواب رفته‌انـد، بنـابراین طول صفی که برای سمافور x تشکیل شده است، هم اکنون برابر مقدار4 است. در ادامه فرآینـد P_1 پس از عبور از ناحیه سمافور x تابع x.signal() را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_2 می‌گردد، در این لحظه فرآیند P_2 در خط بعد از تابع x.wait() و قبل از تـابع x.signal() قـرار می‌گیرد، همچنین در ادامه فرآیند P_1 پس از عبـور از ناحیـه سـمافور z تـابع z.signal() را اجـرا

می‌کند، این امر منجر به بیدار شدن فرآیند P_{11} می‌گردد، در این لحظه فرآیند P_{11} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{11} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{11} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد. پس درحال حاضر 6 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 6 است. در ادامه فرآیند P_2 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_3 می‌گردد، در این لحظه فرآیند P_3 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_2 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{12} می‌گردد، در این لحظه فرآیند P_{12} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{12} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{12} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد.

پس درحال حاضر 7 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 7 است. در ادامه فرآیند P_3 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_4 می‌گردد، در این لحظه فرآیند P_4 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_3 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{13} می‌گردد، در این لحظه فرآیند P_{13} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{13} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{13} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد.

پس درحال حاضر 8 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 8 است. در ادامه فرآیند P_4 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_5 می‌گردد، در این لحظه فرآیند P_5 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_4 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{14} می‌گردد، در این لحظه فرآیند P_{14} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{14} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{14} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد.

پس درحال حاضر 9 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 9 است. در ادامه فرآیند P_5 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_6 می‌گردد، در این لحظه فرآیند P_6 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_5 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{15} می‌گردد، در این لحظه فرآیند P_{15} در خط بعد از تابع ()z.wait

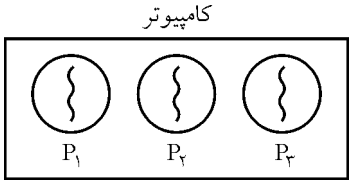
و قبل از تابع y.wait() قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{15} اجرا گردد و تابع y.wait() را اجرا کند، فرآیند P_{15} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد. پس درحال حاضر 10 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 10 است. این حداکثر طول صفی است که می‌تواند برای سمافورy ایجاد شود. در ادامه این سناریو وجود دارد که فرآیند P_1 پس از عبور از ناحیه سمافور y تابع y.signal() را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_6 می‌گردد، در این لحظه فرآیند P_6 در خط بعد از تابع y.wait() و قبل از تابع x.wait() قرار می‌گیرد، درحال حاضر 9 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 9 است. این روند تا اتمام ملاقات هر سه ناحیه x، y و z برای همه فرآیندها ادامه پیدا می‌کند.

۱۶- گزینه (۲) صحیح است.

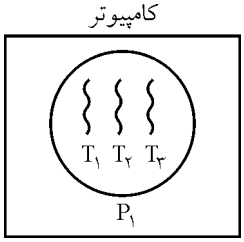
نخ یا ریسمان (Thread)

در سیستم‌های قدیمی‌تر، به ازای هر فرآیند یک رشته نخ یا رشته اجرایی و به تبع یک شمارنده برنامه (PC) وجود داشت اما در سیستم عامل‌های امروزی به ازای هر فرآیند می‌توان چند نخ یا رشته اجرایی داشت.

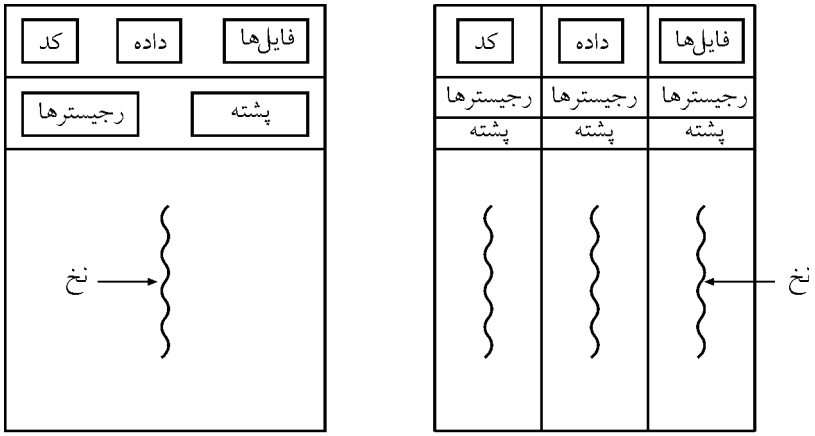
شکل زیر سه فرآیند معمولی را نشان می‌دهد که هریک برای خودشان یک رشته اجرایی و یک حافظه مختص به خود را دارند.



ولی در شکل زیر یک فرآیند، سه رشته اجرایی دارد که هر یک رجیستر، پشته و شمارنده برنامه (PC) مجزای خود را دارند و مانند فرآیندها می‌توانند همروند (در سیستم‌های تک‌پردازنده‌ای) و موازی (در سیستم‌های چندپردازنده‌ای یا چندهسته‌ای) اجرا شوند.



توجه: نخ‌های همتا که در یک فرآیند قرار دارند، از کد، داده و منابع مشترک استفاده می‌کنند اما هر نخ، شمارنده برنامه، مجموعه رجیستر و فضای پشته جداگانه‌ای در اختیار دارد. در واقع هر نخ، TCB مجزایی دارد.



توجه: از آن‌جا که نخ‌های همتا در یک فرآیند قرار داشته و اشتراکات زیادی با هم دارند، عمل تعویض متن بین آنها به راحتی و با هزینه کمتری صورت می‌گیرد، در واقع TCB مربوط به نخ‌ها، محتوی کمتری نسبت به PCB فرآیندها دارد، مثلاً لیست فایل‌های باز مربوط به فرآیندها است، بنابراین این لیست به هنگام تعویض متن فرآیندها باید داخل PCB مربوط به فرآیند ذخیره گردد، در حالی که به هنگام تعویض متن بین نخ‌ها نیازی به ذخیره‌سازی لیست فایل‌های باز مربوط به یک فرآیند در TCB یک نخ نیست. بنابراین تعویض متن بین نخ‌ها نسبت به فرآیندها ارزان‌تر است.

توجه: گاهی از نخ به عنوان Light Weight Process (فرآیند سبک وزن) نیز یاد می‌کنند و به کل یک فرآیند، Heavy Weight Process (فرآیند سنگین وزن) نیز می‌گویند.

توجه: نخ‌ها هم مانند فرآیندها می‌توانند حالت‌های مختلفی را تجربه کنند مانند آماده، در حال اجرا یا منتظر. در واقع پردازنده می‌تواند بین نخ‌ها به اشتراک گذاشته شود.

چند نخی در زبان #C

#C پیاده‌سازی چند نخی را پشتیبانی می‌کند. در زبان #C، هر برنامه به طور پیش فرض از یک نخ تشکیل شده است و درصورت ایجاد نخ‌های دیگر، مفهوم چندنخی پیاده‌سازی می‌گردد.

توجه: نخ اول به صورت پیش‌فرض وجود دارد و برنامه با نخ اول شروع به اجرا می‌کند.

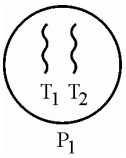
مثال: در قطعه کد زیر نخ T1 به طور پیش‌فرض وجود دارد و نخ T2 ایجاد می‌گردد.

```
static void main ()
    {
Thread     T2=new     Thread (Go);
   T2.Start( )
   Go ( );
        }
static void Go( )
{
for (int i = 0 ; i<5 ; i++)
        console.write("*");
}
```

در این بخش نخ T2 ایجاد می‌گردد و فعالیت Go داخل این نخ قرار می‌گیرد.

نخ T2 اجرا می‌گردد.

فعالیت Go داخل نخ T1 قرار داده شده است. می‌شود فعالیت دیگری تعریف شود و در این بخش فراخوانی شود.



توجه: فعالیت Go داخل نخ T1 قرار دارد، اما فعالیت Go در نخ T2 هم قرار داده شده است. در دو نخ T1 و T2 که فعالیت Go داخل آن قرار دارد، متغیر محلی i در داخل پشته مربوط به هر نخ ایجاد می‌گردد.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

****

چاپ 10 عدد ستاره به دلیل اجرای همروند (سیستم تک‌پردازنده‌ای) یا موازی (سیستم چندپردازنده‌ای یا چند هسته‌ای) دو نخ T1 و T2 است!

مثال: در قطعه کد زیر نخ T1 به طور پیش‌فرض وجود دارد و نخ T2 ایجاد می‌گردد.

```
static void main ()
    {
  Thread     T2=new     Thread (func);
   T2.Start( ) ;
   console.write("X");
        }
static void func( )
{
console.write("Y");
}
```

در این بخش نخ T2 ایجاد می‌گردد و فعالیت func داخل این نخ قرار می‌گیرد.

نخ T2 اجرا می‌گردد.

فعالیت چاپ "X" داخل نخ T1 قرار داده شده است.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

XY

مزایای فرآیندهای چندنخی

۱- ساختار بسیاری از برنامه‌های کاربری ذاتاً از بخش‌های کاملاً مستقل تشکیل می‌شوند که جدا نکردن آن‌ها باعث پیچیدگی بالا و کاهش خوانایی در برنامه می‌گردد. مهندسی نرم‌افزار نیز بر ساخت برنامه‌های کاربردی توسط پیمانه‌های مستقل نیز تأکید دارد. برای مثال، خطاست اگر بیندیشید که برنامه شبیه‌سازی 11 بازیکن یک تیم فوتبال در یک نخ قرار گیرد. به عنوان مثالی دیگر یک برنامه واژه‌پرداز می‌تواند از نخ‌های مستقلی مانند کنترل املا و گرامر، صفحه‌آرایی، مدیریت ورودی‌های کاربر و غیره تشکیل شده باشد.

۲- در فرآیند تک‌نخی، هرگاه فراخوان سیستمی مسدود کننده‌ای اجرا شود، کل فرآیند مسدود می‌گردد. در حالی‌که در فرآیندهای چندنخی در صورتی که سیستم عامل زمان‌بندی چند نخی را پشتیبانی کند، فقط نخی که فراخوان سیستمی مسدودکننده دارد، مسدود می‌گردد و مابقی نخ‌های یک فرآیند می‌توانند پس از در اختیار گرفتن پردازنده، اجرا گردند. فرآیند تک‌نخی مانند قانونی می‌باشد که اگر یک نفر در خانواده خطا کند، همه خانواده محکوم می‌گردند و فرآیند چندنخی مانند قانونی می‌باشد که اگر یک نفر در خانواده خطا کند، فقط همان یک نفر محکوم می‌گردد و بقیه خانواده می‌توانند به زندگی طبیعی خود ادامه دهند.

۳- ایجاد هم‌روندی (در سیستم تک‌پردازنده‌ای) و توازی (در سیستم چندپردازنده‌ای یا چند هسته‌ای) در نخ‌های یک فرآیند و فرآیندهای دیگر.

مثال: کاربرد چندنخی در فرآیند سمت سرویس‌دهنده.

در این مدل، فرآیند سمت سرویس‌دهنده از چندین نخ جهت پاسخ به درخواست‌های کاربر یعنی سرویس‌گیرنده تشکیل شده است. پاسخ هر کاربر می‌تواند توسط یک نخ از سمت سرویس‌دهنده داده شود. چنانچه نخی در فرآیند سرویس‌دهنده جهت تبادل داده از روی دیسک به سمت سرویس‌گیرنده مسدود گردد، نخ‌های دیگر فرآیند سرویس‌دهنده می‌توانند به درخواست‌های دیگر، پاسخ دهند. زیرا کارکرد آن‌ها وابسته به نخ مسدود شده نیست.

توجه: شاید بگویید به جای قرار دادن کارهای مختلف یک سرویس‌دهنده در داخل نخ‌های یک فرآیند، می‌شد هریک از کارها را در داخل یک فرآیند قرار داد و چند فرآیندی را در مقابل چندنخی ابداع کرد. اما به دلایل زیر استفاده از چندنخی معقولانه‌تر به‌نظر می‌رسد:

- هزینه زمانی ایجاد (بارگذاری TCB) و پایان دادن (ذخیره‌سازی TCB) یک نخ در یک فرآیند به مراتب کمتر از ایجاد (بارگذاری PCB) و پایان دادن (ذخیره‌سازی PCB) یک فرآیند است. نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند، در حالی که فرآیندها، منابع مختص به خود را در اختیار می‌گیرند.
- نخ‌های همتا در یک فرآیند، اشتراکات زیادی باهم دارند، بنابراین عمل تعویض متن بین آنها با هزینه کمتری انجام می‌گردد. در حالی که تعویض متن بین فرآیندها به دلیل عدم اشتراکات با هزینه بیشتری انجام می‌گردد.

توجه: وجه اشتراک نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، فایل‌های باز و وجه اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد.

مدیریت نخ‌های هم‌روند(Thread Safety)

در اغلب سیستم‌های امروزی، تعدادی از فرآیندها یا نخ‌ها به صورت همروند بر روی یک پردازنده و یا به صورت موازی بر روی چندین پردازنده یا چندین هسته اجرا می‌شوند. در سیستم‌های چندبرنامگی، چندپردازنده‌ای و چندهسته‌ای، هم‌روندی و توازی فرآیندها و نخ‌ها، یک پدیده‌ی عادی به شمار می‌آید.

فرآیندها و نخ‌های هم‌روند و همکار، به ارتباط با یکدیگر نیاز دارند. آن‌ها برای دستیابی به یک هدف مشترک، نیازمند همکاری، هماهنگی، تبادل داده و استفاده از داده‌ها و سایر منابع مشترک هستند. بنابراین مدیریت اجرای همروند چند فرآیند یا چند نخ بر روی یک پردازنده و اجرای موازی چند فرآیند یا چند نخ بر روی چندین پردازنده یا چندین هسته، حائز اهمیت فراوان است. این مدیریت باید به گونه‌ای باشد که اجرای یک فرآیند یا نخ آسیبی به اجرای فرآیندها یا نخ‌های همکار و هم‌روند دیگر نرساند.

در هنگام طراحی سیستم عامل، در زمینه‌ی ارتباط بین فرآیندها یا نخ‌ها با سه مسأله‌ی اساسی زیر مواجه هستیم:

الف) تبادل داده

گاهی یک فرآیند یا یک نخ، به نتیجه‌ی محاسبات یک فرآیند یا نخ دیگر نیاز دارد، بنابراین به یک مکانیسم برای ارتباط بین فرآیندها یا نخ‌ها نیاز است. انواع مکانیزم‌های تبادل داده بین فرآیندها یا نخ‌ها به روش‌های زیر است:

- **حافظه مشترک**
- **فایل مشترک**

توجه: تبادل داده برای نخ‌های همتا و هم‌خانواده درون یک فرآیند ساده می‌باشد، زیرا نخ‌ها یک فضای آدرس مشترک دارند. اما نخ‌های متعلق به فرآیندهای جداگانه یعنی غیرهمتا که در فضای آدرس متفاوت قرار دارند، در صورت نیاز به ارتباط باید از مکانیسم‌های ارتباط فرآیندها، استفاده کنند.

ناحیه بحرانی

اگر چند فرآیند یا چند نخ قصد دسترسی به یک منبع مشترک را داشته باشند، قطعه کدی از هر فرآیند یا نخ را که در آن به دستکاری این منبع مشترک می‌پردازد، ناحیه‌ی بحرانی می‌گویند.

توجه: در همه‌ی نواحی بحرانی، دسترسی به منبع مشترک، وجود دارد، اما عکس آن همیشه صادق نیست و هرگونه دسترسی به منبع مشترک باعث رقابت و ایجاد ناحیه‌ی بحرانی نمی‌شود.

منبع بحرانی

منبعی که توسط ناحیه‌ی بحرانی مورد دستیابی قرار می‌گیرد، منبع بحرانی نام دارد، مانند متغیرهای مشترک رقابت‌زا.

ب) شرایط رقابتی (مسابقه)

هرگاه دو یا چند فرآیند یا دو یا چند نخ همزمان با هم وارد ناحیه‌ی بحرانی (منبع مشترک) شوند، شرایط رقابتی پیش می‌آید. در شرایط رقابتی، نتیجه‌ی نهایی بستگی به ترتیب دسترسی‌ها دارد. در واقع فرآیندهای همکار یا نخ‌های همکار بر هم اثر دارند و اینکه پردازنده، به چه ترتیبی و در چه زمان‌هایی بین آنها تعویض متن انجام دهد در ایجاد پاسخ نهایی اثرگذار خواهد بود. بنابراین علت شرایط رقابت **تعویض متن پردازنده** بین فرآیندهای همکار یا نخ‌های همکار است.

مثال: شرایط رقابتی

دو نخ هم‌روند Thread1 و Thread2 در یک سیستم اشتراک زمانی که از متغیر مشترک سراسری S، در بخشی از کد خود استفاده می‌کنند در نظر بگیرید، بعد از اجرای کامل دو نخ، مقدار نهایی S چه خواهد شد؟ (مقدار اولیه متغیر سراسری S برابر صفر است).

Thread1 :	Thread2 :
S=S+1	S=S-1

از آنجا که این نخ‌ها به زبان اسمبلی یک ماشین فرضی در نظر گرفته می‌شوند، لذا در ادامه، دستورات فوق را به صورت سطح غیر انتزاعی‌تر (نمایش جزئیات) و در سطح اسمبلی بازنویسی می‌کنیم:

Thread1:	Thread2:

MOVE REGISTER, S
INC REGISTER
MOVE S, REGISTER
	MOVE REGISTER, S
	DEC REGISTER
	MOVE S, REGISTER

حالت اول:

فرض کنید Thread1 کامل اجرا شود و سپس Thread2 کامل اجرا شود (یا برعکس).

Thread1:	Thread2:
① REGISTER ← 0 ② REGISTER ← 1 ③ S ← 1	④ REGISTER ← 1 ⑤ REGISTER ← 0 ⑥ S ← 0

بنابراین مقدار نهایی متغیر S برابر صفر خواهد بود.(S=0)

حالت دوم:

فرض کنید ابتدا ترتیب زیر اجرا شود.

Thread1:

① REGISTER ← 0
INC REGISTER
MOVE S, REGISTER

Thread2:

② REGISTER ← 0
③ REGISTER ← -1
④ S ←-1

سپس پردازنده در اثر تعویض متن به نخ Thread1 باز گردد.

توجه: هر نخ محتویات رجیسترهای خودش را قبل از تعویض متن در TCB ذخیره می‌کند.

بنابراین در این لحظه مقدار رجیستر در TCB فرآیند Thread1، برابر صفر است.

حال در ادامه دستورات باقی مانده نخ Thread1 اجرا می‌شوند.

Thread1:

.
.
.

⑤ REGISTER ←1
⑥ S ←1

بنابراین مقدار نهایی متغیر S برابر مثبت یک خواهد بود.(S= +1)

حالت سوم:

فرض کنید ابتدا ترتیب زیر اجرا شود.

Thread1:

② REGISTER ← 0
③ REGISTER ← 1
④ S ←1

Thread2:

① REGISTER ← 0
DEC REGISTER
MOVE S, REGISTER

سپس پردازنده در اثر تعویض متن به نخ Thread2 باز گردد.

توجه: هر نخ محتویات رجیسترهای خودش را قبل از تعویض متن در TCB ذخیره می‌کند.

بنابراین در این لحظه مقدار رجیستر در TCB نخ Thread2، برابر صفر است.

حال در ادامه دستورات باقی مانده نخ Thread2 اجرا می‌شوند.

Thread2:

.
.
.

⑤ REGISTER ← -1
⑥ S ←-1

بنابراین مقدار نهایی متغیر S برابر منفی یک خواهد بود. (S= -1)

توجه: اما مشکل نهایی اینجاست که مقدار نهایی متغیر S، به نحوه‌ی **تعویض متن پردازنده** یا به عبارتی، به ترتیب اجرای دستورالعمل‌ها، بستگی دارد و می‌تواند مقادیر 0، 1، 1- را داشته باشد. این پدیده، حاصل رقابت بر سر تصاحب یک عامل مشترک (متغیر مشترک S) است.

توجه: وجود پدیده‌ی رقابت در مثال قبل در سیستم‌های تک‌پردازنده‌ای ناشی از وقفه‌ای است که می‌تواند اجرای دستورالعمل‌ها را در هر کجای نخ متوقف نماید. (پدیده‌ی تعویض متن). این وضعیت در سیستم‌های چندپردازنده‌ای و چندهسته‌ای نیز ممکن است پیش بیاید، به علاوه اینکه دو یا چند نخ می‌توانند به موازات هم اجرا شده و برای دسترسی به یک عامل مشترک در رقابت باشند.

برای کنترل شرایط رقابتی، باید راه حلی ارائه شود که سه شرط زیر را به عنوان **معیارهای اخلاقی** در رقابت، رعایت کند:

۱- شرط انحصار متقابل

برای برقرای شرط انحصارمتقابل، عامل مشترک را اسکورت کنید، مانند زمانی که وارد باجه‌ی تلفن همگانی (عامل مشترک) می‌شوید، در را می‌بندید تا مانع ورود شخص دیگری گردید! در عالم انسان‌ها، هیچ دو فردی نباید به طور همزمان وارد عامل مشترک شوند. در عالم نخ‌ها نیز، هیچ دو نخی نباید به طور همزمان وارد عامل مشترک (ناحیه بحرانی) شوند. استفاده‌ی همزمان از عامل مشترک معنا ندارد! (اخلاقی نیست) بنابراین باید راهی پیدا کنیم که از ورود همزمان دو یا چند نخ به ناحیه‌ی بحرانی جلوگیری کند. به عبارت دیگر، آنچه که ما به آن نیاز داریم، انحصار متقابل است که در متون فارسی به آن دو به دو ناسازگاری یا مانعه الجمعی نیز گفته می‌شود، یعنی اگر یکی از نخ‌ها در حال استفاده از حافظه‌ی اشتراکی و یا هر عامل اشتراکی رقابت‌زاست باید مطمئن باشیم که دیگر نخ‌ها، در آن زمان از انجام همان کار محروم می‌باشند. در واقع از بین تمام نخ‌ها، در هر لحظه تنها یک نخ مجاز است، در عامل مشترک باشد. بدین معنی که اگر نخی در ناحیه‌ی بحرانی است، از ورود نخ‌های دیگر به همان ناحیه‌ی بحرانی جلوگیری شود و تا خارج شدن نخ اول منتظر بمانند، زیرا هیچ دو نخی نباید به طور همزمان وارد ناحیه‌ی بحرانی شوند. به یاد داشته باشید که استفاده‌ی همزمان از عامل مشترک معنا ندارد!

بنابراین برای برقراری شرط انحصار متقابل باید ساختاری را طراحی کنیم که در هر لحظه فقط یک نخ مجوز ورود به ناحیه‌ی بحرانی را داشته باشد. لذا هر نخ برای ورود به بخش بحرانی‌اش باید اجازه بگیرد. بخشی ازکد نخ که این اجازه گرفتن را پیاده‌سازی می‌کند، بخش ورودی نام دارد. بخش بحرانی می‌تواند با بخش خروجی دنبال شود. این بخش خروجی کاری می‌کند که نخ‌های دیگر بتوانند وارد ناحیه‌ی بحرانی‌شان شوند. بقیه‌ی کد نخ را بخش باقی‌مانده می‌نامند. بنابراین ساختار کلی نخ‌ها برای برقراری شرط انحصار متقابل به صورت زیر می‌باشد:

```
Thread ()  {
while (TRUE)  {
entry_section () ;  // تلاش برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی
critical_ selection (); // ناحیه‌ی بحرانی
exit_ selection () ; // اعلام خروج از ناحیه‌ی بحرانی
remainder_ section () ; // ناحیه‌ی باقی مانده
  }
}
```

توجه: بدترین شرایط وقتی است که یک نخ بخواهد بارها و بارها وارد ناحیه بحرانی خود شود، برای اینکه سخت‌ترین شرایط بررسی شود، ناحیه بحرانی را داخل حلقه بی‌نهایت قرار می‌دهیم.

۲- شرط پیشرفت

نخی که داوطلب ورود به ناحیه‌ی بحرانی نیست و نیز در ناحیه‌ی بحرانی قرار ندارد، نباید در رقابت برای ورود سایر نخ‌ها به ناحیه‌ی بحرانی شرکت کند، به عبارت دیگر، نباید مانع ورود نخ‌های دیگر به ناحیه‌ی بحرانی شود. در یک بیان ساده‌تر می‌توان گفت، نخی که در ناحیه‌ی باقی مانده قرار دارد، حق جلوگیری از ورود نخ‌های دیگر به ناحیه‌ی بحرانی را ندارد. یعنی نباید در تصمیم‌گیری برای ورود نخ‌ها به ناحیه‌ی بحرانی شرکت کند.

۳- شرط انتظار محدود

نخ‌هایی که نیاز به ورود به ناحیه‌ی بحرانی دارند، باید مدت انتظارشان محدود باشد، یعنی نباید به طور نامحدود در حالت انتظار باقی بمانند.

انتظار نامحدود به دو دسته می‌باشد: (۱) قحطی، (۲) بن‌بست، بنابراین نباید در شرایط رقابتی بین نخ‌ها، قحطی یا بن‌بست رخ دهد. برای اینکه شرط انتظار محدود برقرار باشد، باید هم قحطی و هم بن‌بست رخ ندهد.

قحطی (گرسنگی)

در عالم زندگی قحطی زمانی رخ می‌دهد که عده‌ای مدام از منابع مشترک استفاده کنند و عده‌ای دیگر قادر به استفاده از منابع مشترک نباشند. زیرا دسته‌ی اول از اختصاص منابع به دسته‌ی دوم به طور مداوم و بدون رعایت یک حد بالای مشخص جلوگیری می‌کنند. در عالم نخ‌ها نیز هرگاه نخی به مدت نامعلوم و بدون رعایت یک حد بالای مشخص در انتظار گرفتن یک منبع بحرانی یا دسترسی به یک عامل مشترک بماند و نخی دیگر مدام در حال استفاده از منبع بحرانی باشد، در این حالت نخ اول دچار قحطی شده است. بنابراین در صورت اقدام یک نخ برای ورود به ناحیه‌ی بحرانی، باید محدودیتی برای تعداد دفعاتی که سایر نخ‌ها می‌توانند وارد ناحیه‌ی بحرانی شوند، وجود داشته باشد تا قحطی رخ ندهد.

بن‌بست

به وضعیتی که در آن مجموعه‌ای متشکل از دو یا چند نخ برای همیشه منتظر یکدیگر بمانند (مسدود) و به عبارت دیگر دچار سیکل انتظار ابدی شوند، بن‌بست گفته می‌شود.

توجه: به تفاوت قحطی و بن‌بست توجه کنید، در قحطی نخی مدام در حال کار و نخی دیگر به مدت نامعلوم در انتظار است. اما در بن‌بست، مجموعه‌ای از نخ‌ها در سیکل انتظار ابدی، گرفتار شده‌اند. نه راه پس دارند و نه راه پیش.

توجه: در کنترل شرایط رقابتی، رعایت شرط انحصارمتقابل، شرط لازم و رعایت شروط پیشروی و انتظار محدود، شروط کافی برای ارائه‌ی یک راه‌حل **جامع و اخلاقی** به شمار می‌آیند.

ج) همگام سازی

گاهی اوقات خواسته‌ی ما این است که نخ‌ها به یک ترتیب مشخص و از قبل تعیین شده بر روی یک عامل مشترک (داده مشترک) عملیاتی را انجام دهند. واضح است که در این حالت تبادل داده به روش استفاده از حافظه مشترک است. همچنین از آنجا که پای یک عامل مشترک در میان است، پس رقابت بر سر تصاحب این عامل مشترک هم در میان است. بنابراین راه‌حل همگام‌سازی باید به گونه‌ای باشد که نخ‌های همکار دچار شرایط رقابتی نشوند. به بیان دیگر باید مانع اثر مخرب **تعویض متن پردازنده** بین نخ‌های همکار که عامل ایجاد شرایط رقابتی است، شد.

برای مثال اگر نخ Thread1 داده‌ای را باید تولید کند و سپس نخ Thread2 آن را مصرف کند، نخ Thread2 باید منتظر باشد تا نخ Thread1، داده‌ی مورد نیازش را آماده کند و بعد شروع به مصرف نماید.

مثال: همگام سازی دو نخ

دو نخ Thread1 وThread2 را در نظر بگیرید که در یک سیستم تک‌پردازنده‌ای اشتراک زمانی به صورت هم‌روند اجرا می‌شوند. فرض کنید نخ Thread1 در بخشی از کد خود مقدار متغیر x را می‌خواند و نخ Thread2 نیز در بخشی از برنامه‌اش باید مقدار متغیر x خوانده شده توسط Thread1 را چاپ کند. هدف مسأله، نوشتن این دو نخ به صورتی است که Thread2 در چاپ متغیر x بر Thread1 پیشی نگیرد و اگر زودتر به بخش چاپ متغیر x رسید و هنوز مقدار متغیر x خوانده نشده بود، صبر کند تا Thread1 متغیر x را مقداردهی کند. بنابراین باید یک مسأله‌ی همگام‌سازی حل شود. واضح است که در این حالت تبادل داده به روش استفاده از حافظه مشترک است. یکی از راه‌حل‌های مناسب برای همگام‌سازی نخ‌های همکار، استفاده از کلاس سمافور و یا کلاس مانیتور است.

راه‌حل‌های کنترل شرایط رقابتی نخ‌های همکار

۱- راه‌حل‌های نرم‌افزاری (برعهده‌ی برنامه‌نویس)

۲- راه‌حل‌های سخت‌افزاری

۳- راه‌حل کلاس سمافور

۴- راه‌حل کلاس مانیتور

راه‌حل‌های همگام‌سازی نخ‌های همکار

۱- راه‌حل کلاس سمافور

۲- راه‌حل کلاس مانیتور

توجه: راه‌حل‌های فوق، در سیستم‌های تک‌پردازنده‌ای، چندپردازنده‌ای و چندهسته‌ای با حافظـه‌ی اشتراکی قابل استفاده‌اند.

خواسته سوال این است که چه وقت یک تابع در سطح نخ **ایمن** است؟

جهت پاسخ به خواسته سوال شبه کد زیر را در نظر بگیرید:

```
#include <pthread.h>
  static void main ( )
     {
      Thread T2=new Thread (increment_counter);
         T2.Start ( )
         increment_counter ( );
     }
int increment_counter ( )
{
static int counter = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//only allow one thread to increment at a time
pthread_mutex_lock (&mutex);
counter= counter+1;
//store value before any other threads increment it further
int result = counter;
pthread_mutex_unlock (&mutex);
return result;
}
```

توجـــه: فعالیـــت increment_counter داخـــل نـــخ Thread1 قـــرار دارد، همچنـــین فعالیـــت increment_counter در نخ Thread2 هم قرار داده شده است. هر نخ پشته و رجیستر مخـتص بـه خود را دارد و متغیرهای محلی یک نخ در داخل پشته مربوط به هر نخ ایجاد می‌گـردد. امـا در دو نخ Thread1 و Thread2 که فعالیت increment_counter داخـل آن قـرار دارد، متغیـر counter از نوع static است که داخل سگمنت داده نگهداری می‌شود که به تبع در دو نخ به اشـتراک گذاشـته می‌شود، بنابراین متغیر counter منبع بحرانی و دستور counter= counter+1 ناحیه بحرانـی مـابین دو نخ محسوب می‌گردد که باید محافظت گردد. **بنابراین یک تابع همچون increment_counter در سطح نخ ایمن است اگر و فقط اگر از نخ‌های همروند فراخوانی شود، همیشه نتیجه درست را برگرداند.** دقت کنید که این رابطه دو طرفه صـادق اسـت، بنـابراین شـرط لازم و کـافی در هـر طرف برقرار است و طرف دیگر را نتیجه می دهد. بنابراین گزینه دوم پاسخ سوال است.