فصل دوم:
فرایند (Process)

مریم تقی زاده

ترم اول ۱۴۰۴

Email: taghizadehmail@gmail.com

رئوس مطالب فصل ۲

- **فرآیند**
  - تعاریف و مفاهیم اولیه
  - حالات فرآیند
  - انواع فرآیندها
- **نخ**
- **ارتباط بین فرآیندها**
- **بررسی مشکلات کلاسیک**
- **زمانبندی**
- **بن بست**

فرآیند (Process)

- برنامه در حالت معمول نهاد غیرفعال است.
- **فرآیند:**
  - برنامه در حال اجرا را گویند.
  - چون فرآیند در حال اجرا است نیاز به منابعی دارد مانند
  - حافظه، CPU، دستگاه ورودی/خروجی
  - علاوه بر کد برنامه، شامل
  - مقدار شمارنده برنامه، ثبات های CPU، پشته و بخش داده ها (Data Section)
- کد برنامه را بخش متن می نامند. (Text Section)

تعاریف

- فرآیند CPU-Limited
  - بیشتر زمان کامپیوتر صرف محاسبات می شود و CPU بیشتر درگیر است

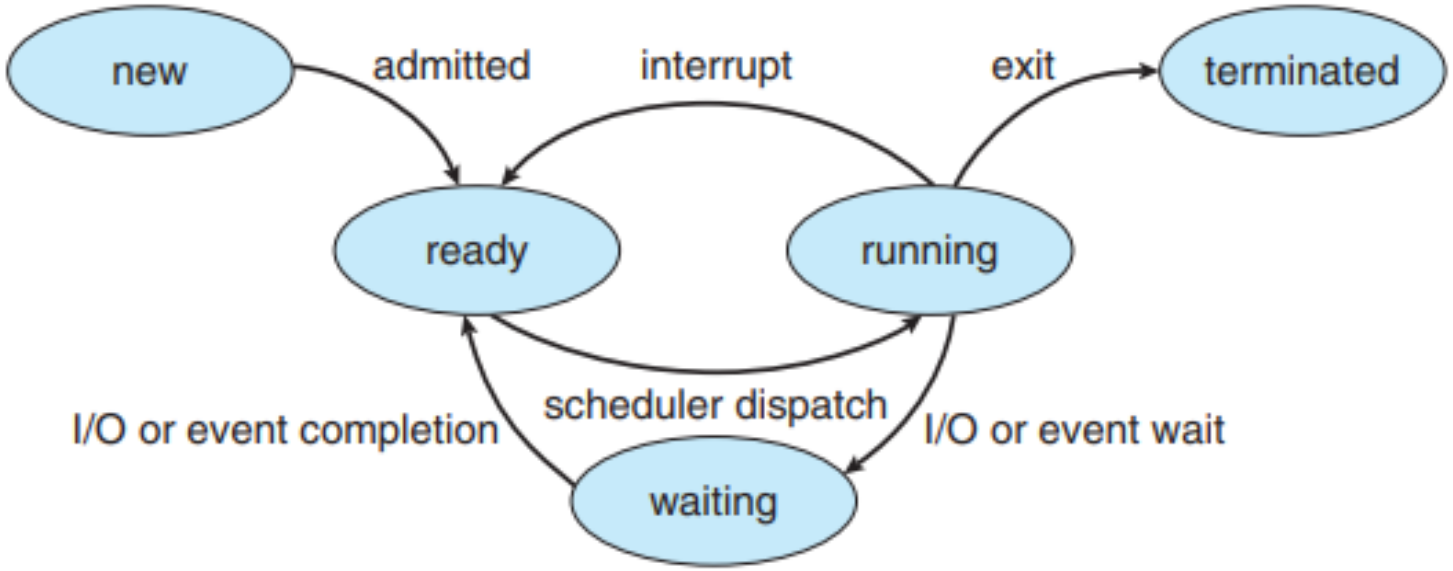
- فرآیند I/O-Limited
  - بیشتر زمان کامپیوتر صرف دستگاههای I/O (ورود و خروج داده ها) می شود.

تعاریف

- فرآیند وقتی که در حالت اجرا قرار دارد می تواند در حالت های مختلفی قرار گیرد.
- حالت فرآیند : توسط فعالیت فعلی آن تعریف می شود.

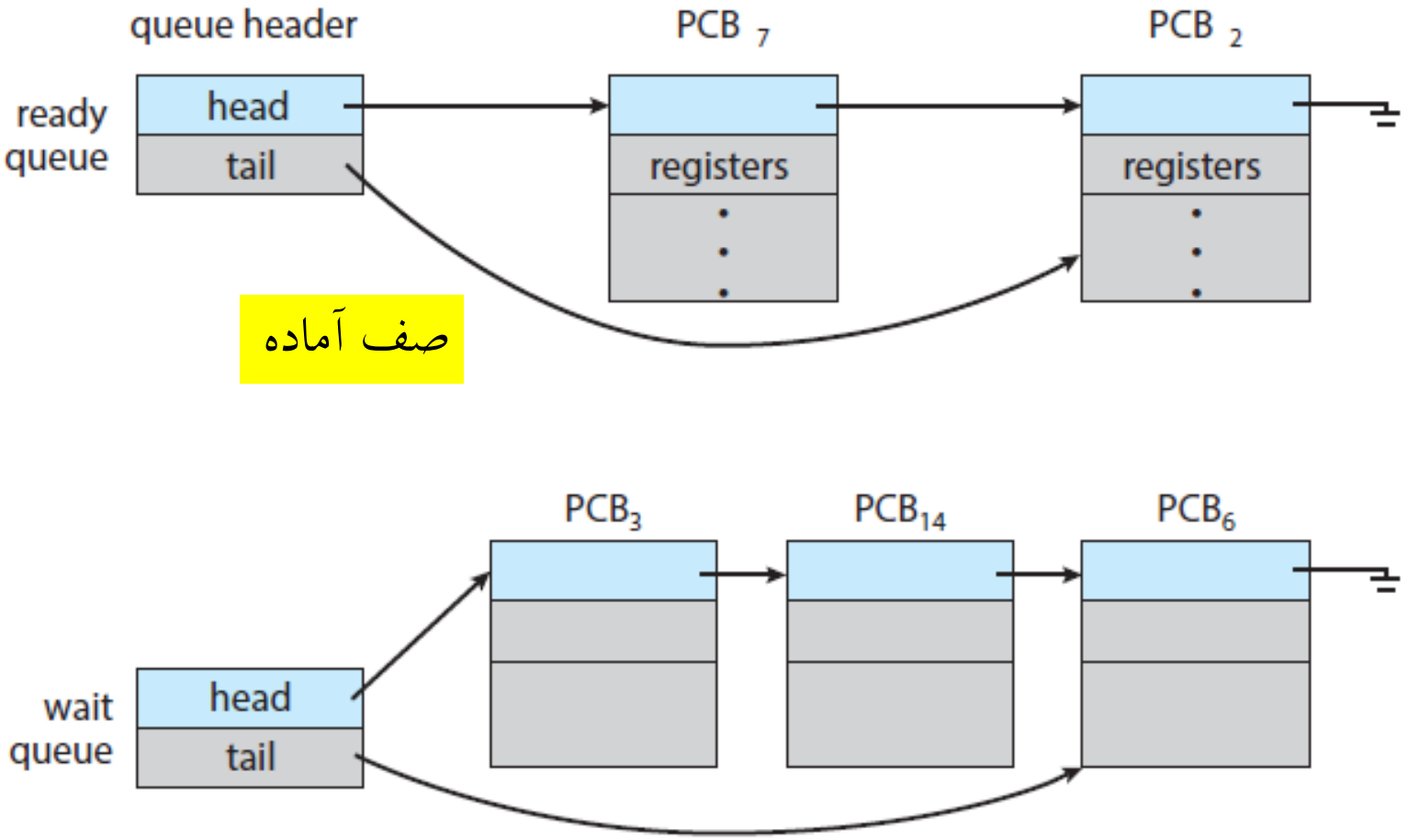
- **جدید (New):** فرآیند،جدیدا ایجاد شده است.
- **اجرا (Running):** دستورات فرآیند در حال اجرا است.
- **انتظار (Waiting) یا مسدود(Blocked):** فرآیند منتظر یک رویدادی است که رخ دهد.
- **آماده (Ready):** فرآیند آماده است تا CPU به آن اختصاص داده شود تا بتواند اجرا شود.
- **خاتمه (Terminated):** اجرای فرآیند به اتمام رسیده است.
- **در هر لحظه فقط یک فرآیند می تواند در پردازنده اجرا شود.**

مدل حالات فرآیند (۵ حالته)

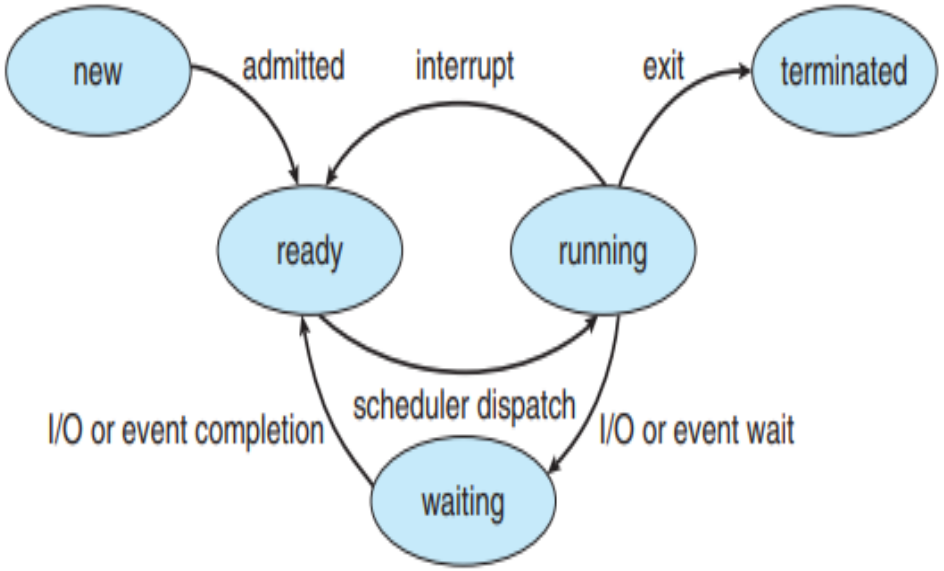


تمام حالات ارائه شده در این تصویر، در حافظه **RAM** قرار دارند.

صف آماده و انتظار



تغییر حالات فرآیند در مدل ۵ حالته



- جدید به آماده
- آماده به اجرا
- اجرا به خروج
- مسدود (انتظار) به آماده
- اجرا به مسدود (انتظار)
- اجرا به آماده
- آماده به خروج
- مسدود (انتظار) به خروج

تغییر حالات فرآیند در مدل ۵ حالته

زمانی که سیستم عامل یکی از فرآیندهای آماده را به حالت اجرا می برد، به این عمل توزیع (**dispatch**) می گویند.

زمانی که فرآیند از حالت انتظار (مسدود)، با رخ دادن رویداد به حالت آماده می رود، به آن **wake up** گویند.

به دلیل اتمام بازه زمانی اجرای فرآیند در سیستم چندبرنامه ای، یا روی دادن وقفه، فرآیند از اجرا به آماده می رود.

آماده به خروج (مسدود به خروج): با خاتمه فرآیند والد، ممکن است همه فرآیندهای فرزند نیز پایان یابد.

مدل حالات فرآیند (۷ حالته)

- **معلق (Suspend)**
- اگر تمام فرآیندهای در مرحله اجرا منتظر I/O باشند (که در حالت انتظار یا مسدود قرار خواهند گرفت)، این امکان وجود دارد که:
  - CPU بیکار بماند.
  - صف آماده تقریبا خالی باشد.
- **تکنیک مبادله (Swapping)**
- فرآیند از حافظه اصلی به حافظه دیسک (یا بالعکس) منتقل می شود. (Swap out/In)
- خارج یا داخل کردن فرآیندها از حافظه، باعث می شود که فرآیندهای وابسته به I/O که فعلا CPU نیاز ندارند از حافظه خارج و بالعکس، فرآیندهای وابسته به CPU که به CPU نیاز دارند در حافظه وارد شوند.

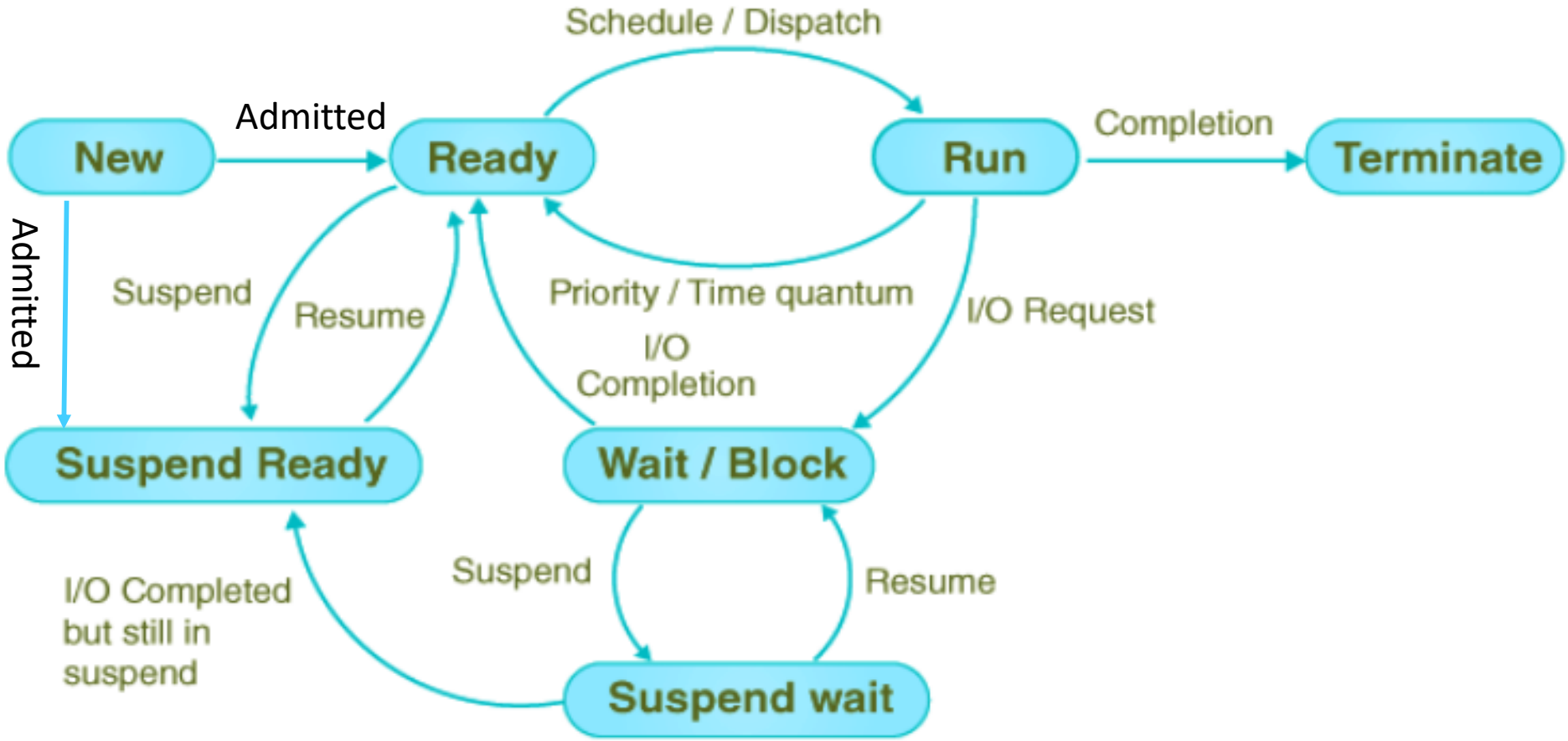
این حالت به منظور جلوگیری از هدر رفتن زمان پردازنده و استفاده حداکثری از حافظه فرض شده است.

مدل حالات فرآیند (۷ حالته)

- **حالت مسدود و معلق (Suspend-Block یا Suspend-Wait)**
- زمانی که تعداد فرآیندهای مسدود(منتظر) زیاد باشد ممکن است حافظه کم بیاد، سیستم عامل فرآیندها را از انتظار معلق می کند که به عنوان حال «مسدود و معلق» شناخته می شود.
- فرآیند در حافظه ثانویه است و منتظر بروز یک رویداد است.

- **حالت آماده و معلق (Suspend-Ready)**
- اگر فرآیند آماده باشد ولی حافظه نداشته باشیم به حالت «آماده و معلق» می رود.
- فرآیند در حافظه ثانویه است و به محض بارگذاری در حافظه اصلی، آماده اجرا است.

مدل حالات فرآیند (۷ حالته)



عملیات های بر روی فرآیند

- **ایجاد فرآیند**
- هنگامی که یک برنامه اجرا می شود یک فرآیند جدید ایجاد می شود.
- سیستم عامل منابع مورد نیاز را به آن فرآیند تخصیص می دهد.
- یک PCB برای ذخیره اطلاعات فرآیند جدید استفاده می شود.
- یک فرآیند (فرآیند والد) می تواند فرآیند جدیدی (فرآیند فرزند) ایجاد کند.
- **پایان فرآیند**
- با پایان اجرای برنامه، فرآیند خاتمه می یابد.
- سیستم عامل منابع تخصیص یافته به فرآیند را آزاد می کند.

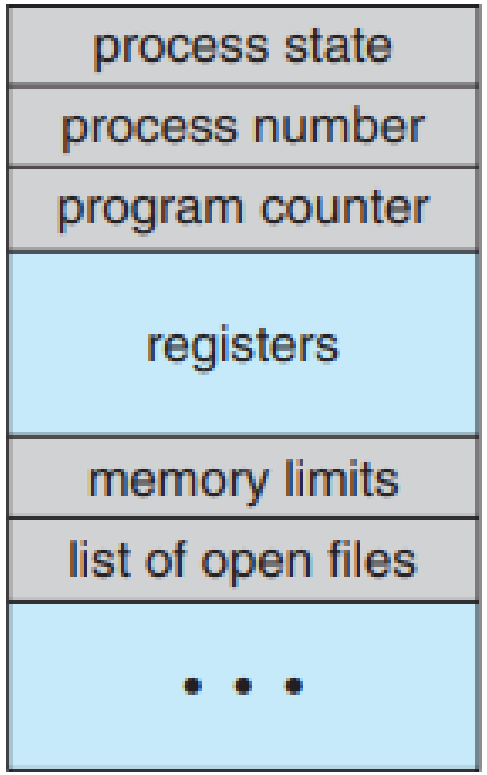
دلایل پایان فرآیند

- پایان نرمال برنامه
- سقف زمانی
- انتظار زیادتر از حد برای بروز یک رویداد
- خطای حافظه (نیاز به حافظه بیش از حد)
- خطای دسترسی به مکان های غیرمجاز حافظه
- خطای ورودی/خروجی
- استفاده نامناسب از داده
- خطای حفاظت (تلاش برای دسترسی به منبعی که مجاز نیست)
- خطای محاسباتی (مانند تقسیم به صفر)

- پایان یافتن فرآیند والد
- درخواست والد
- دستورالعمل نامعتبر
- اجرای دستورالعمل ممتاز (هسته)
- دخالت سیستم عامل

بلوک کنترل فرآیند

- بلوک کنترل فرآیند (Process Control Block)
  - یک ساختار داده در سیستم عامل است که اطلاعات مربوط به هر فرآیند در حال اجرا را نگهداری می کند.
  - این اطلاعات برای مدیریت و کنترل فرآیندها توسط سیستم عامل ضروری است.
- حاوی اطلاعات زیر است:
  - شناسه فرآیند (شماره منحصر بفرد فرآیند)، حالت فرآیند
  - شمارنده برنامه، ثبات های CPU
  - اطلاعات زمانبندی CPU (اولویت فرآیند، اشاره گر به صف زمانبندی)
  - اطلاعات مدیریت حافظه (اطلاعات حافظه تخصیص یافته)
  - اطلاعات حسابرسی (میزان استفاده از پردازنده)
  - اطلاعات وضعیت I/O



ضرورت بلوک کنترل فرآیند

مدیریت فرآیندها

- به سیستم عامل امکان می دهد تا فرآیندهای مختلف را ردیابی و مدیریت کند.

تعویض متن

- هنگام تعویض بین فرآیندها، سیستم عامل اطلاعات PCB را ذخیره و بازیابی می کند تا اجرای فرآیندها بدرستی ادامه یابد.

زمانبندی فرآیندها

- از اطلاعات PCB در تصمیم گیری در زمانبند سیستم عامل استفاده می شود.

مدیریت منابع PCB

- اطلاعات مربوط به منابع تخصیص یافته به هر فرآیند را نگهداری می کند.

تعویض متن (Context-switching)

- **تعویض متن:**
- زمانی که سیستم عامل CPU را از یک فرآیند (Pi) گرفته و در اختیار یک فرآیند دیگر (Pj) قرار می دهد، اطلاعات PCB که برای اجرای مجدد فرآیند Pi لازم دارد را نگه‌داری می‌کند و اطلاعات PCB فرآیند Pj را بازیابی می‌کند.

- این تعویض متن توسط هسته سیستم عامل انجام می شود.

- زمان تعویض متن، سربار اضافی برای کامپیوتر است.

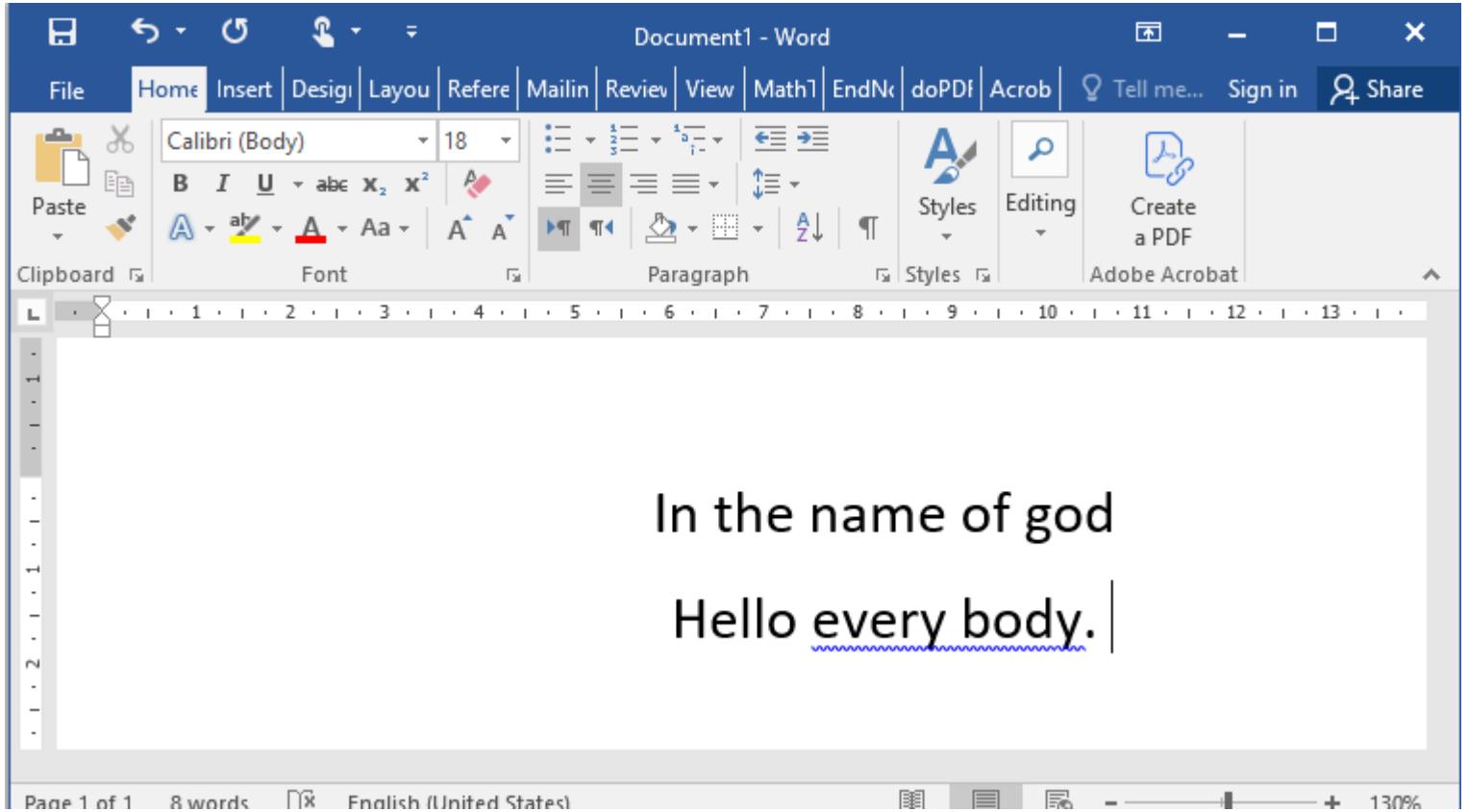
نخ (Thread)

- مدل اولیه فرآیند:
  - برنامه درحال اجرا با یک رشته اجرا
  - فرآیند، دستورات را به صورت متوالی اجرا می کند.
  - محدودیت: (۱):عدم توانایی در اجرای همزمان وظایف (۲): عدم بهره‌وری از پردازنده‌های چندهسته‌ای

- رشته، ترد (نخ)
  - واحدهای کوچکتری از اجرا در یک فرآیند هستند.
  - فرآیند می تواند چندین نخ داشته باشد که بطور همزمان اجرا شوند. (اجرای همروند)
  - امتیازات: (۱): اجرای همزمان وظایف (۲): استفاده کارآمدتر از منابع سخت افزاری

- هر فرآیند می‌تواند چند رشته اجرایی داشته باشد. به رشته‌های اجرایی که می‌توانند موازی با هم اجرا شوند، نخ یا ترد گویند.

مثال نخ (Thread)



مثال: پردازشگر داده

نخ ۱: نوشتن کارکترها

نخ ۲: بررسی املای کلمات

نخ ۳: بررسی گرامر

نخ ٤: ذخیره خودکار فایل

مثال نخ (Thread)

- مثال: صفحات مرورگر وب

- نخ۱: بارگذاری و نمایش متن وب
- نخ۲: برای بارگذاری هر تصویر در صفحه وب
  - Thread21, thread22, thread23, …
- نخ۳: بارگذاری ویدئوهای صفحه وب

مثال نخ (Thread)

- اجرای برنامه چند رسانه ای (بخواهیم یک فیلم را نمایش بدهیم)

- نخ ۱: مسؤل خواندن فایل
- نخ ۲: پخش صدا و تصویر
- نخ ۳: مدیریت رابط کاربری برنامه

با استفاده از چندنخی، می توان همزمان فایل را خواند و صدا و تصویر را پخش کرد و رابط کاربری را مدیریت کرد.

تفاوت نخ و فرآیند

- فرآیند یک واحد مستقل از اجرای برنامه است که دارای فضای آدرس، منابع سیستم و شناسه منحصر بفرد است.

- نخ یک واحد اجرایی درون فرآیند است که فضای آدرس، منابع سیستم و شناسه فرآیند را با سایر نخ های موجود در همان فرآیند به اشتراک می گذارد.

ویژگی نخ

- نخ‌ها دارای شناسه، شمارنده برنامه، ثبات ها و پشته مخصوص به خود هستند.
- نخ‌ها می‌توانند همزمان اجرا شوند و به این ترتیب، عملکرد برنامه را بهبود بخشند.
- نخ‌ها می‌توانند با یکدیگر ارتباط برقرار کنند و داده ها را به اشتراک بگذارند.

امتیازات نخ

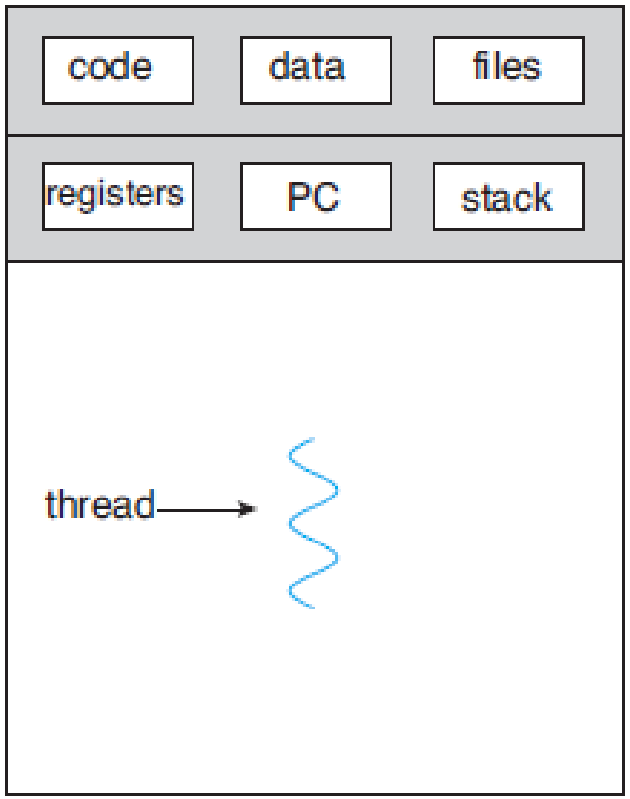
- پاسخگو بودن
  - اگر قسمتی از برنامه بلوکه شود و یا عملیات طولانی داشته باشد، بقیه نخ های برنامه اجرا می شوند.
- اشتراک منابع
  - نخ ها در حافظه و منابعی که مربوط به فرآیند آن ها است، شریک هستند.

- صرفه اقتصادی
  - تخصیص حافظه و منابع برای فرآیند هزینه بر است. اما به دلیل اشتراک منابع یک فرآیند برای نخ های درون آن، ایجاد نخ مقرون به صرفه است.
- توسعه پذیری
  - استفاده از نخ در سیستم های چندپردازنده ای می تواند قابلیت اجرای موازی را فراهم نماید.

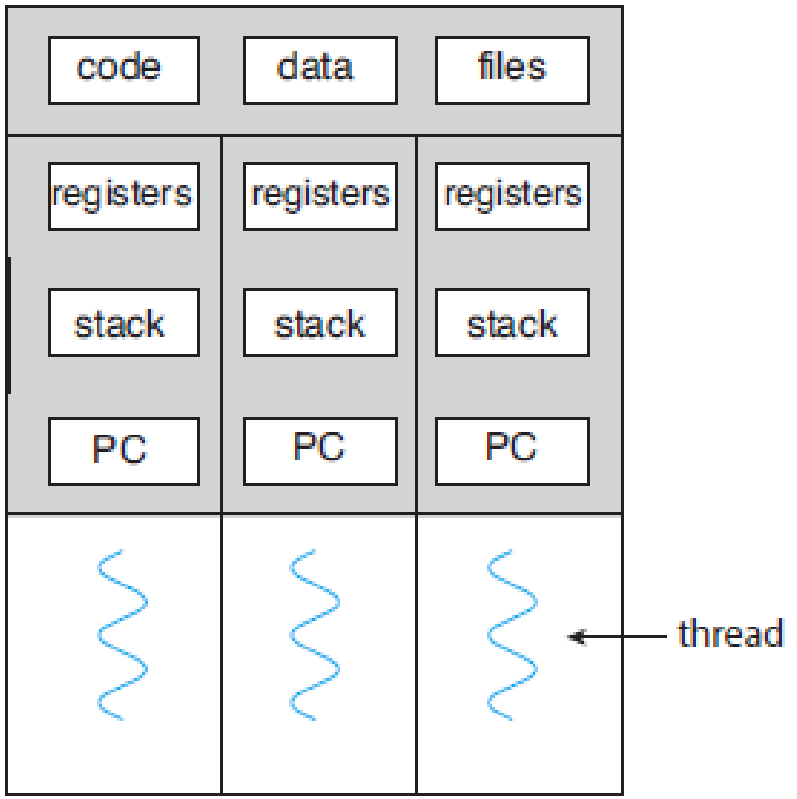
فرآیند تک نخی و چندنخی

- فرآیندهای تک نخی و چندنخی دو مدل اصلی در سیستم عامل برای اجرای برنامه ها هستند.
- تک نخی (Single thread)
- چند نخی (Multi thread)

فرآیند تک نخی و چندنخی



فرآیند تک نخی



فرآیند چند نخی

در یک فرآیند با چندین نخ، نخ‌ها در قسمت کد، داده، فایل‌ها و منابع سیستم‌عامل شریک هستند.

فرآیند تک نخی و چندنخی

- تک نخی (Single thread)
  - هر فرآیند فقط یک نخ اجرایی داد.
  - دستورات به صورت متوالی و پشت سر هم اجرا می‌شوند.
  - اگر دستور یا وظیفه ای مسدود شود، کل فرآیند مسدود می شود.
  - پیاده سازی ساده ای دارد.
  - برای برنامه ساده تک وظیفه ای مناسب است.

- **معایب**
  - عملکرد پایین تر به دلیل عدم امکان اجرای همزمان وظایف.
  - مسدود شدن یک دستور، کل نخ و بنابراین فرآیند را مسدود می کند.

فرآیند تک نخی و چندنخی

- چند نخی (Multi thread)
  - هر فرآیند می تواند چندین نخ اجرایی داشته باشد.
  - وظایف می توانند همزمان یا موازی اجرا شوند.
  - پیاده سازی و اشکال زدایی آن پیچیده است.
  - برای برنامه های چندوظیفه ای مناسب می باشد.
  - استفاده بهینه از منابع
  - اجرای همزمان عملکرد بالایی را به همراه دارد

- **معایب**
  - مدیریت نخ های متعدد سبب مصرف منابع بیشتر می شود.
  - به دلیل دسترسی همزمان به منابع مشترک مشکلات همزمانی را به دنبال دارد.

انواع نخ

- نخ در سطح کاربر (User thread)
  - نخ های کاربر در سطح کاربر مدیریت می‌شوند.
  - سیستم‌عامل عملی برای نخ های کاربر انجام نمی‌دهد و اطلاعی از وضعیت نخ ها ندارد.
  - نخ کاربر توسط کتابخانه‌های کاربر مدیریت می شود (شامل ایجاد و حذف و مدیریت و زمانبندی نخ) .

- نخ در سطح هسته (Kernel thread)
  - نخ‌های هسته سیستم‌عامل به وسیله سیستم‌عامل پشتیبانی و اجرا می‌شوند.
  - این نخ ها در سطح سیستم ایجاد می شوند.

مقایسه نخ کاربر و هسته

- نخ کاربر
- مزایا:
  - سرعت بالا: تعویض بین نخ های کاربر نیاز به نعامل با هسته ندارد.
  - هزینه پایین: ایجاد و مدیریت نخ های کاربر کم هزینه است.
  - قابل حمل بودن: نخ کاربر می تواند برروی هر سیستم عاملی که از کتابخانه نخ کاربر پشتیبانی می کند اجرا شود.
  - ارتباط و همگام سازی راحتتر است.
- معایب
  - مسدود شدن: اگر نخ فراخوانی سیستم نیاز داشته باشد، بدلیل اینکه سیستم عامل اطلاعی از نخ ها ندارد، پس کل فرآیند با تمام نخ های آن مسدود می شود، اگر نقص حافظه رخ دهد، کل فرآیند مسدود می شود.
  - مدیریت توسط کاربر سبب می شود که هسته بدلیل نا آگاهی از نخ‌ها، بخوبی تخصیص منابع بین نخ ها را انجام ندهد.
  - عدم پشتیبانی از چند پردازنده ای: توزیع نخ ها و زمانبندی آنها میان چند پردازنده به خوبی انجام نمی شود.

مقایسه نخ کاربر و هسته

- نخ هسته
- مزایا:
  - مدیریت توسط هسته: تخصیص بهتر منابع سیستم بین نخ ها توسط هسته انجام شده و زمانبندی اجرا را کنترل می کند.
  - مسدود شدن یک نخ سبب، مسدودی دیگر نخ های یک فرآیند نمی شود.
  - معایب نخ های سطح کاربر را ندارد.
  - پشتیبانی از چند پردازنده و همزمانی واقعی: بهبود عملکرد برنامه های چند نخی و افزایش سرعت را به همراه دارد.

- معایب
  - هر عملیات نخ توسط هسته انجام شده و به یک فراخوانی سیستمی نیاز دارد و هزینه بالا است.
  - سربار سیستم: تعویض بین نخ های هسته نیاز به تعامل با هسته دارد که سربار سیستم افزایش می یابد.

مقایسه نخ کاربر و هسته

اگر برنامه به عملکرد بالا و پشتیبانی از چند پردازنده نیاز دارد، پیاده سازی نخ هسته مناسب تر است.

اگر برنامه به سرعت بالاتر و انعطاف پذیری نیاز دارد، پیاده سازی نخ کاربر مناسب تر است.

ارتباط نخ کاربر و هسته

- سیستم‌عامل خبری از نخ‌های کاربرندارد. برای این که نخ های کاربر بتوانند از امکانات سیستم عامل نیز استفاده کنند، نخی در هسته سیستم عامل درست می شود که با نخ کاربر در ارتباط باشد.

- استفاده از مزایای نخ کاربر و نخ هسته

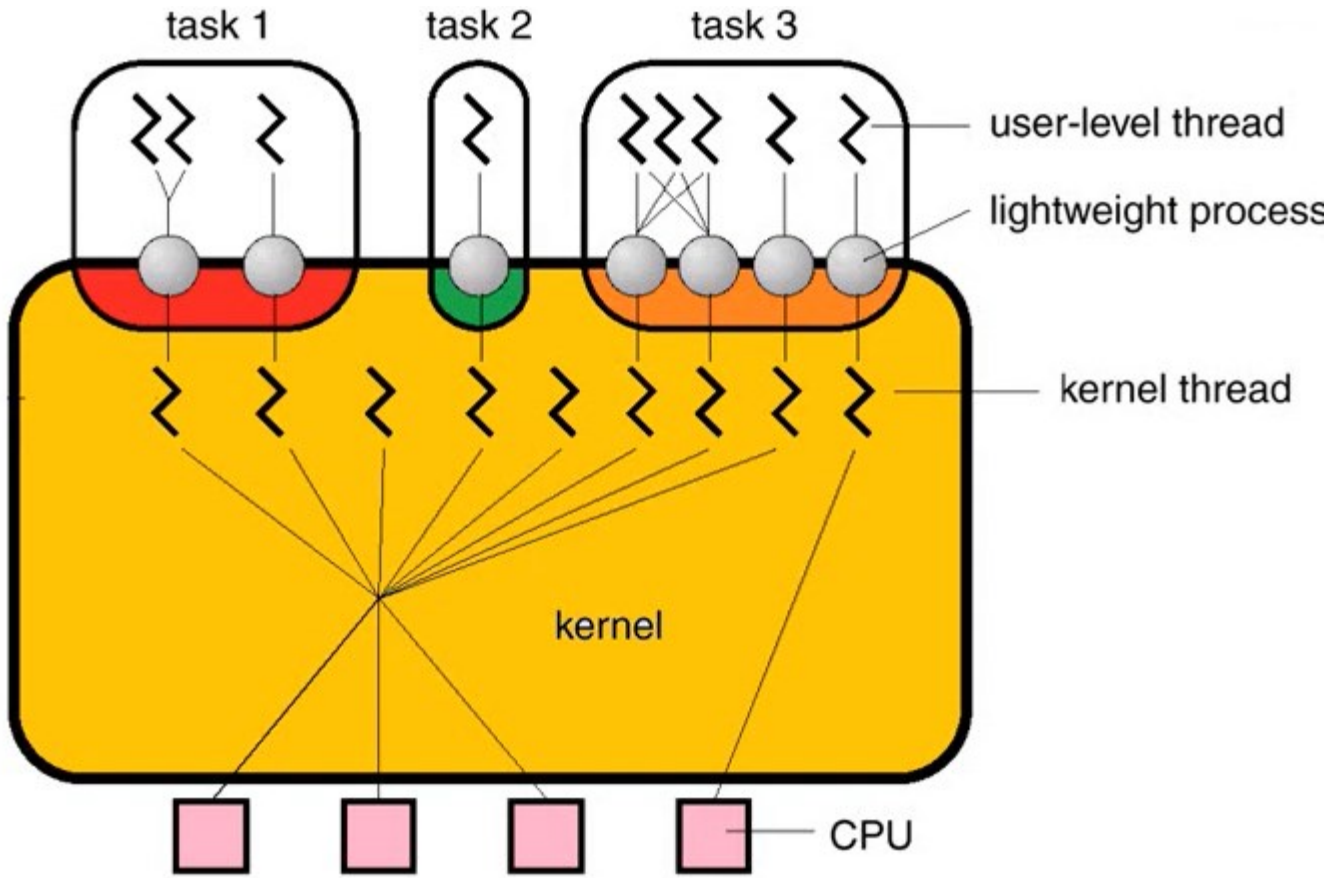
روش ترکیبی (Hybrid thread)

- روش ترکیبی مزایای هر دو رویکرد نخ های کاربر و نخ های هسته را ترکیب می کند و معایب آن ها را کاهش می دهد.

- عملکرد: نگاشت بین نخ سطح کاربر و سطح هسته
  - تعدادی نخ کاربر به تعدادی نخ هسته نگاشت می شوند.
  - در واقع سیستم عامل تعدادی نخ هسته را برای اجرای نخ های کاربر اختصاص می دهد.
  - فرآیند سبک وزن (Light Weight Process) ، نگاشتی بین نخ سطح کاربر و نخ سطح هسته است.
  - هر LWP، یک یا تعداد بیشتری نخ را حمایت می کند و به یک نخ سطح هسته نگاشت می کند.
  - مزایای سیستم ترکیبی،
    - چندین نخ درون یک برنامه می توانند به طور موازی برروی چند پردازنده اجرا شوند.
    - مسدود شدن یک نخ، مسدودی کامل فرآیند را به دنبال ندارد.

- این روش می تواند ۳ مدل مختلف را ایجاد کند: یک به یک، چند به یک، چند به چند.

روش ترکیبی


task 1
task 2
task 3
user-level thread
lightweight process
kernel thread
kernel
CPU

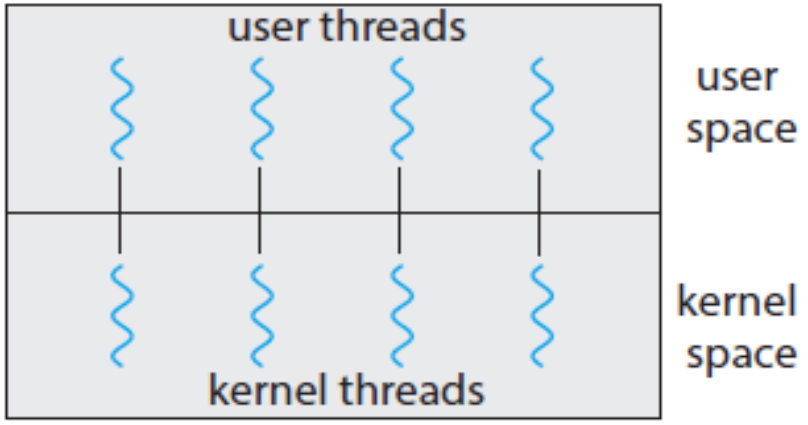
روش ترکیبی

- مدل نخ های ترکیبی تلاش می کند تعادلی بین عملکرد و انعطاف پذیری ایجاد کند. با استفاده از نخ های هسته، امکان بهره مندی از مزایای چند پردازشی واقعی فراهم می شود و با استفاده از نخ های کاربر، تعویض نخ ها سریع تر انجام می شود و پاسخگویی برنامه بهبود می یابد.

- پیاده‌سازی و مدیریت نخ‌های ترکیبی پیچیده‌تر از مدل‌های نخ‌های کاربر یا هسته خالص است.

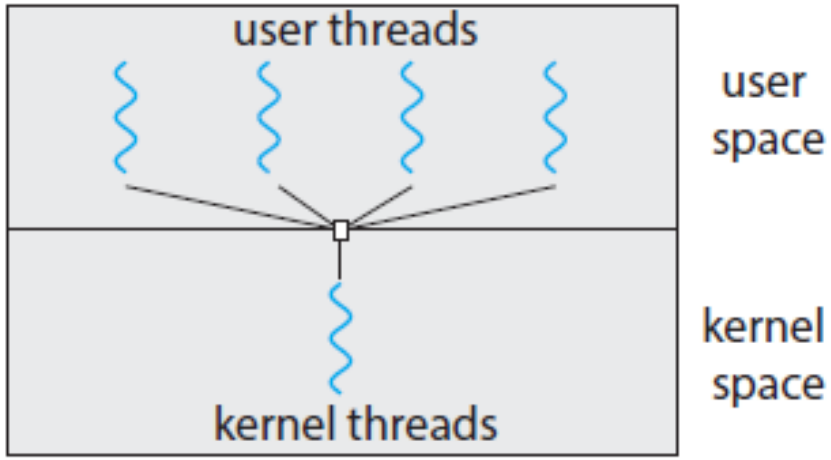
مدل های چند نخی

- مدل یک به یک (One-to-One model)
  - هر نخ سطح کاربر به یک نخ سطح هسته متصل می شود
  - بین نخ های کاربر و نخ های هسته یک رابطه یک به یک وجود دارد.
  - امکان اجرای همزمان نخ ها برروی پردازنده های مختلف وجود دارد.
  - مسدود شدن یک نخ سبب مسدود شدن کل فرآیند نمی شود.
  - ایجاد نخ های زیاد، سربار زیادی به سیستم تحمیل می کند.

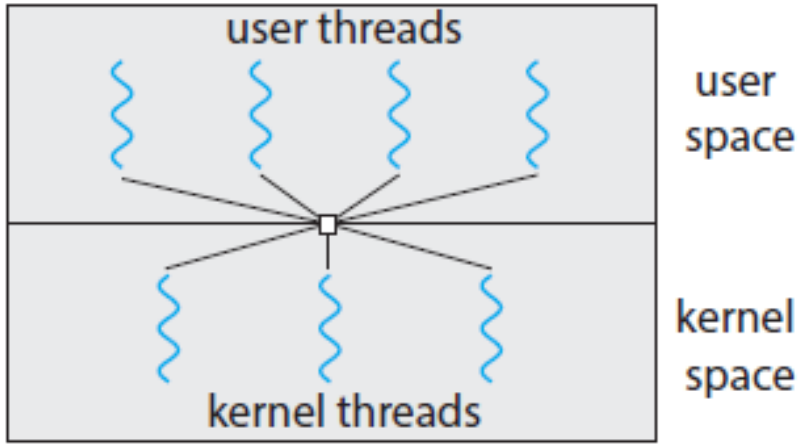


مدل های چند نخی

- مدل چند به یک (Many-to-One model)
  - تعداد زیادی نخ کاربر با یک نخ هسته در ارتباط هستند.
  - مدیریت نخ ها در سطح کاربر انجام می شود که سریع تر و کارآمدتر است.
  - قابلیت جابجایی نخ ها بین سیستم عامل های مختلف وجود دارد.
  - مسدود شدن یک نخ کاربر، باعث مسدود شدن نخ های مرتبط با آن نخ هسته می شود.
  - امکان استفاده از چند پردازنده ای وجود ندارد.



مدل های چندنخی



- مدل چند به چند (Many-to-Many model)
  - چندین نخ سطح کاربربه تعداد کمتری نخ سطح هسته نگاشت می شود.
  - تعداد نخ های هسته می تواند کمتر یا مساوی تعداد نخ های سطح کاربر باشد.
  - در مدیریت نخ ها انعطاف پذیری بالایی وجود دارد.
  - موازی سازی نخ ها بر پردازنده های مختلف امکان پذیر است.
  - یک فراخوان مسدود کننده، کل سیستم را مسدود نمی کند.
  - امکان استفاده از نخ های زیاد در سطح کاربر، بدون اعمال بار زیادی بر هسته فراهم است.
  - پیچیدگی پیاده سازی و مدیریت نخ ها، می تواند بالا باشد.
  - این احتمال وجود دارد که عملکرد سیستم در شرایطی غیرقابل پیش بینی باشد.

انواع فرآیندها

- فرآیندهای مستقل
  - داده ها را با سایر فرآیندها به اشتراک نمی گذارند و تحت تاثیرسایر فرآیندها قرار نمی گیرند.

- فرآیندهای همکار
  - فرآیندهایی که می توانند بر یکدیگر تاثیر بگذارند، یا تحت تاثیر قرار گیرند.
  - معمولا داده ها را به اشتراک می گذارند.

- فرآیندها نیاز است که با فرآیندهای دیگر در ارتباط باشند.

ارتباط بین فرآیندها (Inter process communication-IPC)

- ارتباط میان فرآیندها یکی از بخش های مهم سیستم عامل است که به فرآیندها اجازه می دهد با یکدیگر همکاری داشته و وظایف پیچیده را انجام دهند.
- **دلایل ارتباط بین فرآیندها**
  - اشتراک اطلاعات
  - افزایش سرعت محاسبات
  - مادولار بودن
  - راحتی
  - ارائه خدمات
  - مدیریت منابع

ارتباط بین فرآیندها (Inter process communication-IPC)

- اشتراک اطلاعات: بسیاری از برنامه ها برای انجام دستورات خود نیاز به اشتراک اطلاعات با یکدیگر دارند.

- افزایش سرعت محاسبات: تقسیم دستورات بین فرآیندهای مختلف با هدف اجرای موازی آنها

- مادولاربودن: طراحی سیستم مبتنی بر ماژول ها و استفاده از ماژول های مختلف برای انجام وظایف پیچیده و در نظر گرفتن فرآیندهای جداگانه برای انجام وظایف ماژول‌ها

- راحتی: هر کاربر بتواند در هر لحظه بر روی چندین برنامه به طور موازی کار کند.

ارتباط بین فرآیندها (Inter process communication-IPC)

- ارائه خدمات: برخی از فرآیندها خدمات خاصی را به سایر فرآیندها ارائه می دهند. مثلا یک فرآیند چاپگر خدمات چاپ را به دیگر فرآیندها ارائه کند.

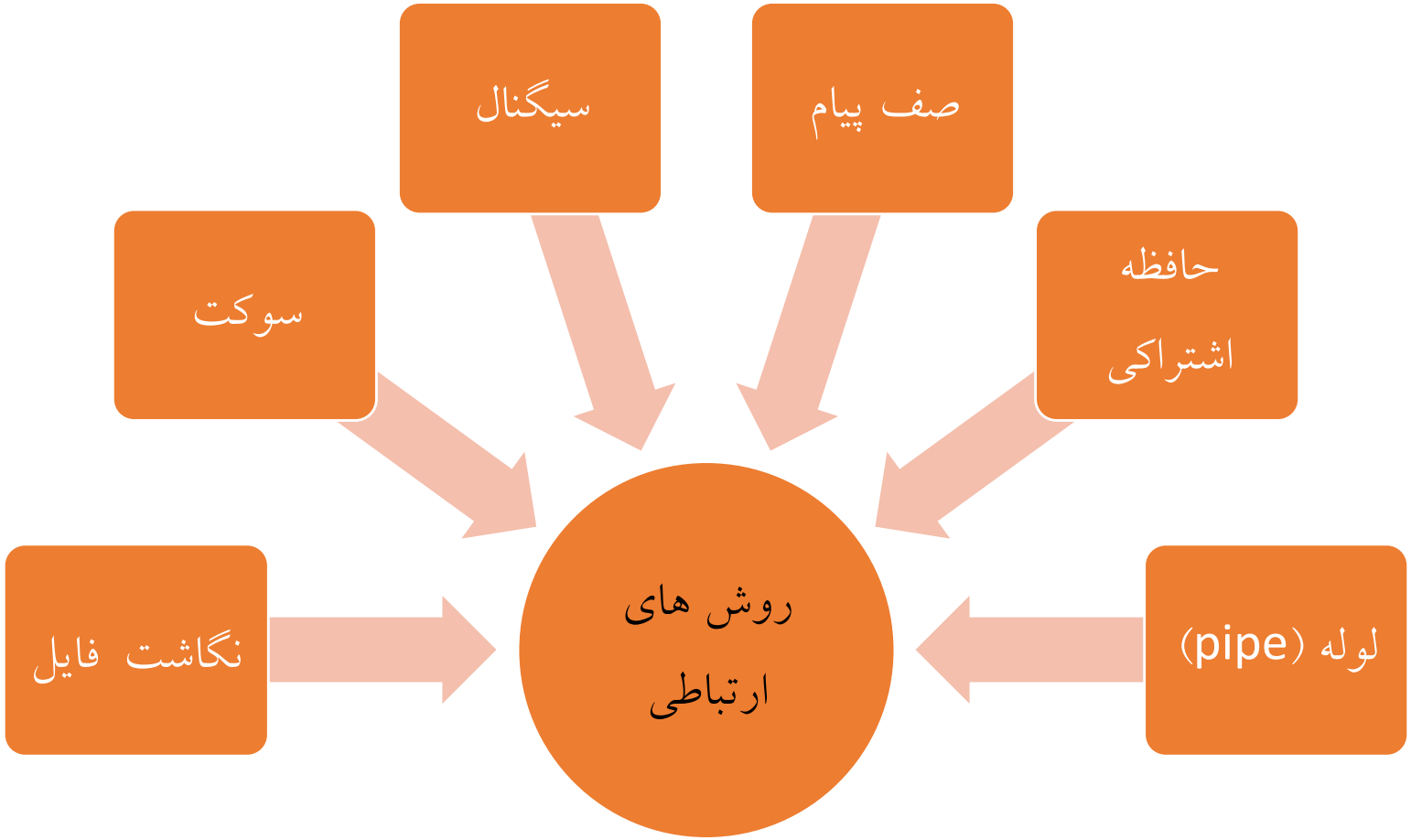
- مدیریت منابع: تعدادی از فرآیندها برای مدیریت منابع سیستم عامل استفاده می شوند. مانند: حافظه، I/O، فایل ها.

ارتباط بین فرآیندها (Inter process communication-IPC)

- **در ارتباط میان فرآیندها ۳ مساله اهمیت دارد:**

۱. چگونگی ارتباط میان فرآیندها
۲. فرآیندها در بخش منابع و داده های اشتراکی درگیر نشوند.
۳. نظم و ترتیب اجرای فرآیندها

روش های ارتباط بین فرآیندها



همزمانی فرآیندها

- در فرآیندهای همکار که بر یکدیگر تاثیرگذار هستند، چگونگی و ترتیب تحویل CPU به فرآیندها و زمان های سوئیچ متن، در پاسخ نهایی تاثیر دارد.

- همگام سازی (همزمانی)
  - اگر بین فرآیندها وابستگی وجود داشته باشد ترتیب درست انجام فرآیندها اهمیت زیادی دارد و باید رعایت شود.

- شرایط رقابتی (Race condition)
  - فرآیندهایی که بطور همزمان به داده اشتراکی دسترسی دارند و نتایج به ترتیب اجرا فرآیندها بستگی داشته باشد، این شرایط را مسابقه می نامند.
  - باید راهکاری ارائه کرد که شرایط مسابقه رخ ندهد.

همزمانی فرآیندها (مثال)

- فرض کنید دو پروسه P1 و P2 بدین صورت باشد:
- متغیر سراسری a برای هر دو فرآیند در ابتدا صفر است. چند اجرای همروند از فرآیندهای P1 و P2 به دست آورید.

```
P1:
    a=1;
```

```
P2:
    b=a;
    c=a;
```

a=1, b=1, c=1

b=0, a=1, c=1

b=0, c=0, a=1

همزمانی فرآیندها

- **ناحیه بحرانی** (Critical section)
  - بخشی از فرآیند که به حافظه اشتراکی (یا منبع اشتراکی) دسترسی دارد را ناحیه بحرانی می گویند.

- همکاری فرآیندها باید بگونه ای باشد که از رخ دادن «شرایط مسابقه» جلوگیری شود و ارتباط صحیحی بین آنها برقرار باشد.بنابراین باید شرایط زیر برقرار باشد:
  - **انحصار متقابل** (Mutual exclusion): هنگام اجرای بخش بحرانی یک فرآیند، فرآیند دیگری نباید در ناحیه بحرانی باشد.
  - **پیشرفت**: برای ورود به ناحیه بحرانی، فقط فرآیندهایی که هنوز ناحیه بحرانی خود را اجرا نکرده اند، در تصمیم گیری می توانند دخیل باشند.
  - **انتظار محدود** (Bounded waiting): یک فرآیند برای ورود به ناحیه بحرانی نباید بطور نامحدود در حالت انتظار قرار گیرد.

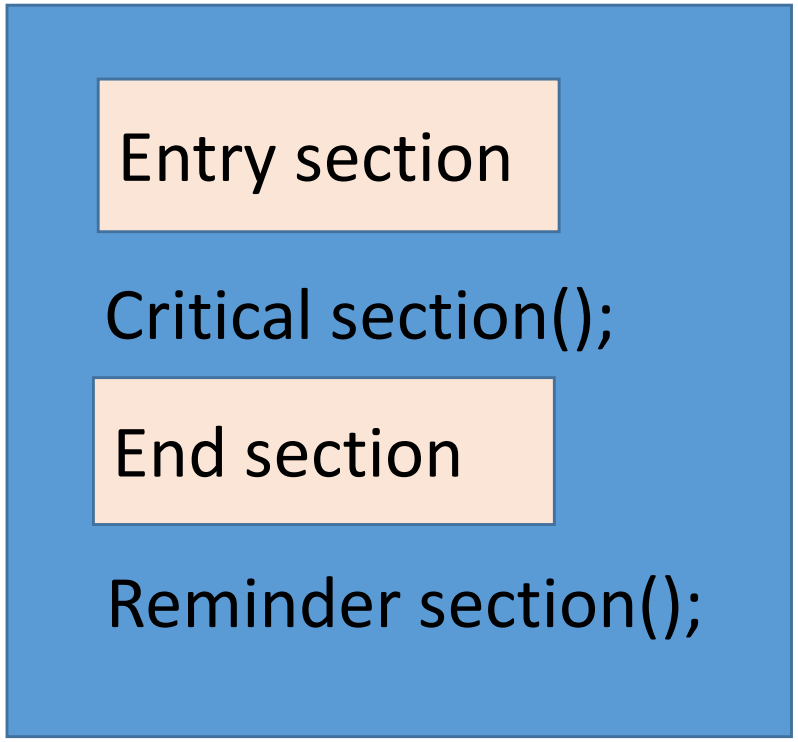
همزمانی فرآیندها

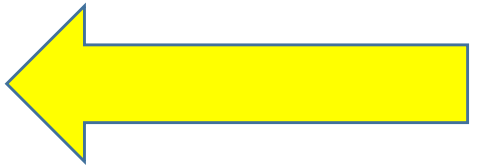
بن بست :
یک فرآیند تا ابد منتظر ورود به ناحیه بحرانی است.

گرسنگی:
به مدت نامعلوم و نامشخص منتظر فرآیندهای دیگر باشد.

ناحیه بحرانی

- هر فرآیند برای ورود به بخش بحرانی باید اجازه بگیرد.
- بخشی از کد فرآیند که «اجازه گرفتن» را پیاده سازی می کند را بخش ورودی (Entry section) گویند.
- بخش بحرانی می تواند با بخش خروجی یا exit section دنبال شود.
- ادامه کد فرآیند را بخش باقیمانده گویند. (remainder section)





ساختار کلی فرآیند

انحصار متقابل

- روش های پیاده سازی انحصار متقابل را می توان به صورت زیر دسته بندی کرد:

1. نرم افزاری
2. مبتنی بر سخت افزار
3. مبتنی بر سیستم عامل
4. مبتنی بر زبان برنامه سازی (کامپایلر)

رویکردهای نرم‌افزاری

- روش های نرم افزاری بدون حمایت دستورالعمل های سخت افزاری، سیستم عامل و زبان های برنامه سازی، توسط خود برنامه ها بکار می رود.
- متغیرهای قفل
- الگوریتم دکر (شامل چند تلاش)
  - تلاش اول: تناوب قطعی
  - تلاش دوم،تلاش سوم، تلاش چهارم و تلاش پنجم.
- الگوریتم پترسون
  - استفاده از دو متغیر مشترک

متغیرهای قفل

- یک متغیر مشترک به نام قفل (Lock) وجود دارد. مقدار اولیه آن صفر است.
- فرآیند برای ورود به ناحیه بحرانی، متغیر قفل را چک می کند، در صورتی که برابر صفر باشد، آن را یک کرده و وارد ناحیه بحرانی می شود.
- اگر متغیر Lock برابر یک باشد، منتظر می ماند و نمی تواند وارد ناحیه بحرانی شود.
- صفر: فرآیندی در ناحیه بحرانی نیست.
- یک: یعنی فرآیندی در ناحیه بحرانی اش قرار دارد.

متغیرهای قفل

P0

```
While (Lock==1);
Lock = 1 ;

Critical section;

Lock = 0;

Reminder section;
```

P1

```
While (Lock==1);
Lock = 1 ;

Critical section;

Lock = 0;

Reminder section;
```

متغیرهای قفل

- سناریوی اول:

- P0 : check Lock → P0: Lock=1 → P0: Enter Critical Section → P1 : Check Lock (Waiting)→ P0: Lock=0→ P0: Reminder Section→ P1: Check Lock→ P1: Lock=1→ P1: Enter Critical→ P0: Reminder Section→ P1: Lock = 0→ ….

- سناریوی دوم:

- P0: Check Lock (False) → P1: Check Lock (False) → P0: Lock=1 → P1: Lock=1 → P0:Enter Critical Section → P1: Enter Critical Section → ….

- طبق سناریوی دوم، هر دو فرآیند همزمان وارد ناحیه بحرانی شده اند.

الگوریتم دکر

- اولین روش نرم افزاری توسط دکر Decker مبتنی بر دو فرآیند برای حل مساله انحصار متقابل ارائه شد.

- در ۵ مرحله تلاش به نتیجه رسید و توانست هر ۳ شرط(انحصار متقابل، پیشرفت، انتظار محدود) را برآورده کرد.

الگوریتم دکر (تلاش اول)

- **تناوب قطعی**
- دو فرآیند دارای یک متغیر مشترک هستند که نوبت فرآیند را تعیین می کند و می تواند ۰ یا ۱ باشد.
- فرض دو فرآیند P0 و P1 و متغیر مشترک turn را داریم.

```
P0:
{
While (True) {
        while (turn != 0);
         Critical_Section();
          turn = 1;
          Reminder Section();
          }
}
```

```
P1:
{
While (True) {
        while (turn != 1);
         Critical_Section();
          turn = 0;
          Reminder Section();
          }
}
```

الگوریتم دکر (تلاش اول)

- تست مداوم یک متغیر تا زمانی که حاوی یک مقدار مشخصی شود، را Busy Waiting (انتظار مشغول) می نامند.
- باید از این شرایط اجتناب کرد زیرا وقت CPU را هدر می دهد.
- با توجه به این ویژگی، این روش زمانی مناسب است که زمان انتظار کوتاه باشد.

- **نکته:** شرط پیشرفت رعایت نمی شود.
- مثال: فرآیند P1 وارد ناحیه بحرانی شده است. فرآیند P0 نیز فرآیند بحرانی خود را قبلا طی کرده است و الان در بخش Reminder است. فرآیند P1 ناحیه بحرانی را به اتمام رسانده و turn را ۰ می کند و بخش Reminder خود را اجرا کرده و سریعا مجدد می خواهد وارد ناحیه بحرانی خود شود و بدلیل اینکه Turn برابر صفر است نمی تواند وارد شود و این در حالی است که فرآیند P0 نیز در بخش بحرانی نیست.

الگوریتم دکر (تلاش دوم)

- به جای turn از دو متغیر Flag استفاده می کند. برای هر فرآیند یک کلید مجزا فرض شده است تا اگر فرآیندی قصد ورود به ناحیه بحرانی خود را ندارد، مانع فرآیند دیگر برای ورود به ناحیه بحرانی نشود.

```
P0:
{
While (True) {
        while (Flag [1] );
         Flag [0] = true;
         Critical_Section();
          Flag [0] = False ;
          Reminder Section();
          }
}
```

```
P1:
{
While (True) {
        while (Flag [0] );
         Flag [1] = true;
         Critical_Section();
          Flag [1] = False ;
          Reminder Section();
          }
}
```

الگوریتم دکر (تلاش دوم)

- شرط انحصار متقابل برآورده نمی شود. طبق سناریوی زیر هر دو فرآیند P0 و P1 می توانند همزمان در ناحیه بحرانی خود باشند.

P0: While (Flag [1])→P1:While (Flag [0])→P0: Flag [0]=True;→P1:Flag[1] = True; → P0:Critical_section(); → P1 : Critical_section();

- شرط انتظار محدود رعایت نمی شود.

P0: Critical_section()→P1: while(Flag [0]);→P0: اتمام ناحیه بحرانی, Flag [0]=False, Reminder_section(); While(Flag [1]); Flag [0]=True;P1: while(Flag [0]);→P0:.....

الگوریتم دکر (تلاش سوم)

- تغییر مختصری در تلاش دوم اعمال شده است.
- دو دستور ابتدایی جابجا شده اند.

```
P0:
{
While (True) {
        Flag [0] = true;
        while (Flag [1] );
        Critical_Section();
         Flag [0] = False ;
         Reminder Section();
         }
}
```

```
P1:
{
While (True) {
        Flag [1] = true;
        while (Flag [0] );
        Critical_Section();
          Flag [1] = False ;
          Reminder Section();
          }
}
```

الگوریتم دکر (تلاش سوم)

- شرط انتظار محدود برآورده نمی شود.

- P0: Flag [0] = true;→ P1: Flag [1] = true;→P0:while (Flag [1]);→P1:while (Flag [0]);→ P0:while (Flag [1]);→P1:while (Flag [0]);→ P0:while (Flag [1]);→P1:while (Flag [0]);→ P0:while (Flag [1]);→P1:while (Flag [0]);→ P0:while (Flag [1]);→P1:while (Flag [0]);→ P0:while (Flag [1]);→P1:while (Flag [0]);→ P0:while (Flag [1]);→P1:while (Flag [0]);→…

الگوریتم دکر (تلاش چهارم)

```
P0:
{
While (True) {
        Flag [0] = true;
        while (Flag [1] )
        {
         Flag [0] = false;
         delay ();
         Flag [0] = true;
        }
        Critical_Section();
         Flag [0] = False ;
         Reminder Section();
         }
}
```

```
P1:
{
While (True) {
        Flag [1] = true;
        while (Flag [0] )
         {
          Flag [1] = false;
          delay ();
          Flag [1] = true;
         }
        Critical_Section();
          Flag [1] = False ;
          Reminder Section();
          }
}
```

الگوریتم دکر (تلاش چهارم)

- انتظار محدود برآورده نمی شود.

- ممکن است که فرآیند P0 در حلقه مربوط به delay خود بماند و در مقابل فرآیند دیگر P1 خیلی سریع ناحیه بحرانی خود را اجرا کرده و باقیماند کد را نیز اجرا و مجدد وارد ناحیه بحرانی خود شود. فرآیند P0 همواره در حالت انتظار مانده و اجازه ورود به ناحیه بحرانی ندارد.

الگوریتم دکر (تلاش پنجم)

```
P0:
{
While (True) {
        Flag [0] = true;
        while (Flag [1] )
        {
         if (turn == 1) {
         Flag [0] = False;
         While (turn == 1) do;
         Flag [0] = true; }
        }
        Critical_Section();
         turn = 1 ;
         Flag [0] = False ;
         Reminder Section();
         }
}
```

```
P1:
{
While (True) {
        Flag [1] = true;
        while (Flag [0] )
        {
         if (turn == 0) {
         Flag [1] = False;
         While (turn == 0) do;
         Flag [1] = true; }
        }
        Critical_Section();
         turn = 0 ;
         Flag [1] = False ;
         Reminder Section();
         }
}
```

در این روش، متغیر **Turn** و **Flag** را همزمان استفاده می کند.

مقایسه الگوریتم دکر

الگوریتم دکر	شرط انحصار متقابل	شرط پیشروی	شرط انتظار محدود
تلاش اول	√	×	√
تلاش دوم	×	√	×
تلاش سوم	√	√	×
تلاش چهارم	√	√	×
تلاش پنجم	√	√	√

الگوریتم پترسون

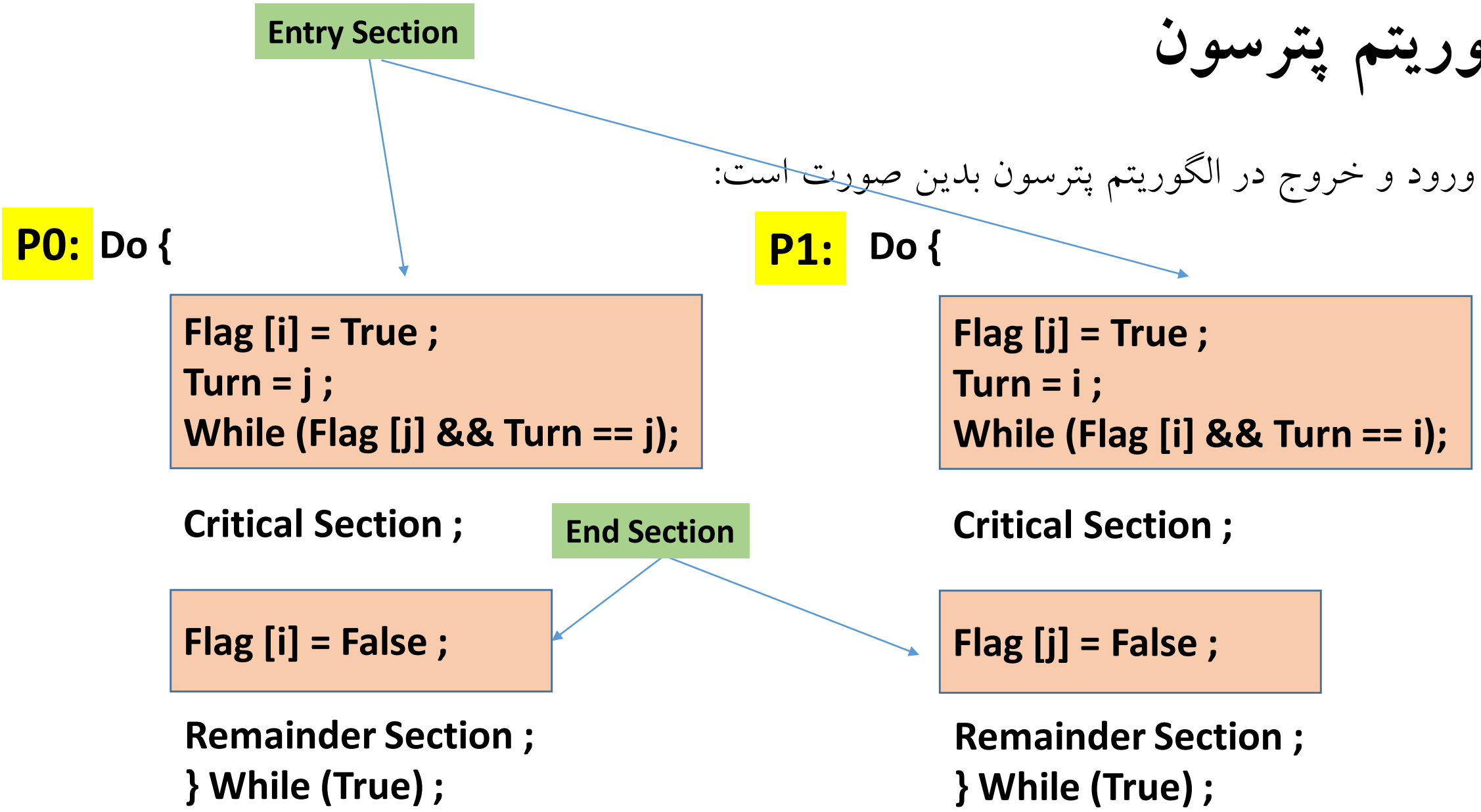
1. در این الگوریتم از دو متغیر مشترک استفاده می شود: turn و Flag
2. زمانی که یک فرآیند (Pi) آماده است که وارد ناحیه بحرانی خود شود، متغیر Flag خود را True می کند. سپس،
3. فرآیند Pi مقدار متغیر Turn را به شماره فرآیند مقابل (Pj) مقداردهی می کند.
4. فرآیند Pi در صورتی وارد ناحیه بحرانی خود می شود که نوبت و پرچم Flag به ترتیب برابر بااندیس فرآیند و True باشند.

نکته ۱: در این الگوریتم شرط انحصار متقابل، پیشروی و انتظار محدود برقرار است.

نکته ۲: الگوریتم برای دو فرآیند قابل اجرا است. می توان برای اجرای فرآیندهای بیشتر تعمیم داد.

الگوریتم پترسون

- کد ورود و خروج در الگوریتم پترسون بدین صورت است:



رویکردهای سخت افزاری

- دستورات سخت افزاری و قابلیت CPU، امکانی فراهم می کند که محتویات حافظه را تغییر داده و در حین اجرای آن دستور، پردازنده غیرقابل وقفه است.

- از این قابلیت برای حل مساله ناحیه بحرانی می توان بهره برد.
  - غیرفعال کردن وقفه ها
  - دستور TSL (Test-and-Lock)
  - دستور Swap (Compare-and-Swap)

غیرفعال کردن وقفه

- هر فرآیند بلافاصله پس از ورود به ناحیه بحرانی اش، کلیه وقفه‌ها را از کار می‌اندازد و درست قبل از خروج از ناحیه بحرانی دوباره همه وقفه ها را فعال می‌کند.

- با خاموش کرن وقفه ها، CPU نمی‌تواند از فرآیندی به فرآیند دیگر سوئیچ کند.
  - بنابراین فرآیند بدون مداخله دیگر فرآیندها ناحیه بحرانی خودش را اجرا می کند.

- ایراد
- اگر کاربر فراموش نماید که دوباره وقفه را فعال نماید، سیستم از کار خواهد افتاد. بنابراین غیرفعال کردن وقفه توسط برنامه کاربر عاقلانه نیست.
- در سیستم چندپردازنده‌ای، غیرفعال شدن وقفه توسط کاربر، فقط برای پردازنده عمل می‌کند و برای دیگر پردازنده‌ها تاثیری ندارد و آنها می‌توانند به ناحیه مشترک دسترسی داشته باشند.

دستور TSL

- معمولا CPU دارای دستوری به نام TSL است که:
  - محتویات یک کلمه از حافظه را خوانده (متغیر Lock)
  - و در یک ثبات قرار می‌دهد. (ثبات RX)
  - و یک مقدار غیرصفر (مثلا عدد ۱) را در همان آدرس ذخیره می‌نماید. (Lock=1)
- دستور اتمیک است.
  - هیچ فرآیند و CPU دیگری نمی‌توانند به این کلمه حافظه دسترسی داشته باشند تا زمانی که اجرای دستور به پایان برسد.
  - در واقع گذرگاه حافظه قفل می شود که کسی اجازه دسترسی نداشته باشد.

TSL RX, Lock

دستور TSL

```
Enter_Region:
        TSL Register, Lock
        CMP Register, #0
        JNE  Enter_Region;
        RET
```

```
Leave_Region:
        MOVE Lock, 0
        RET
```

Process

```
While (True)  {
      Enter_Region(Lock);
      Critical section;
      Leave_Region(Lock);
      Remainder section;
       }
```

دستور TSL

- شرط انحصار متقابل را برآورده می کند.
- انتظار مشغولی (Busy waiting) را دارد که خوب نیست. زیرا وقت CPU را هدر می دهد.

- نکته: اگر فرآیندهای اولویت دار در سیستم داشته باشیم. استفاده از این روش، بدلیل مساله انتظار مشغولی، مشکلاتی را ایجاد می کند.

- **مثال:**
- فرآیندی با اولویت (L) در حال اجرای ناحیه بحرانی خود است (یعنی Lock=1). فرآیندی با اولویت (H) (یعنی اولویت بالا) وارد سیستم شده و می خواهد اجرا گردد. پردازنده از فرآیند (L) گرفته می‌شود و CPU به فرآیند (H) داده می شود. حال این فرآیند در حالت حلقه انتظار یعنی مرحله اجازه ورود به ناحیه بحرانی خود قرار می گیرد و منتظر می‌ماند تا اجازه ورود به ناحیه بحرانی را پیدا کند. زیرا Lock=1 است و اجازه ورود ندارد. چون اولویت بالایی دارد، CPU را نیز پس نمی دهد.

روش مبتنی بر سیستم عامل (سمافور)

- برای حل مساله ناحیه بحرانی استفاده از یک ابزار همزمانی به نام سمافور (Semaphore) پیشنهاد شد.

- سمافور
  - یک متغیر صحیح است.
  - با دو عملیات wait و signal قابل دسترسی است.
  - وقتی فرآیندی مقدار سمافور را تغییر می‌دهد هیچ فرآیند دیگری نمی‌تواند همان سمافور را تغییر دهد.

```
Wait (s):
        While S<=0  no-op;
                S -- ;
```

```
Signal (s):
                S ++ ;
```

سمافور

- از سمافور برای حل مساله بحرانی n فرآیند می توان استفاده کرد. در اینجا یک سمافور بنام mutex با مقدار اولیه ۱ به اشتراک گذاشته می شود و هر فرآیند p دارای کد زیر است:

```
Do {
    Wait (mutex);

    Critical Section ;

    Signal (mutex);

    Remainder Section ;
} While (True) ;
```

سمافور

- روش ارائه شده برای حل مساله ناحیه بحرانی و همزمانی، دارای مشکل busy waiting است.

- **پیشنهاد برای رفع مشکل**
- اگر فرآیند اجازه ورود به ناحیه بحرانی اش را ندارد، دیگر در حلقه while معطل نشده و وقت CPU را نمی گیرد و آن فرآیند، بلوکه یا مسدود می شود. (اصطلاحا sleep است.)
  - بنابراین، فرآیند دیگری می تواند CPU را در اختیار داشته باشد.

- فرآیند بلوکه شده، با استفاده از تابع Signal می تواند دوباره به شرایط اجرا برگردد و در حالت آماده قرار گیرد. (wake up رخ می دهد.)

سمافور

```
Struct  Semaphore{
    int value;
    Struct  Process List;
}
Semaphore S;
```

```
Wait ( S):
{
   S.value  - - ;
   if  (S.value < 0)  {
        Add process Pi to  S.List;
        Block (Pi);
    }
}
```

```
Signal (s):
{
  S.value ++;
  if (S.value <= 0) {
       Remove a process of S.List;
        Wake up (process);
   }
}
```

پیاده سازی انحصار متقابل به کمک سمافور، شرط های انحصار متقابل، پیشرفت و انتظار محدود را برآورده می کند.

مثال ۱

- دو فرآیند به نام‌های P1 و P2 داریم که می‌توانند بطور همروند اجرا شوند.
- فرآیند P1 شامل یک دستور S1 است.
- فرآیند P2 شامل یک دستور S2 است.
- فرض کنید S2 فقط باید بعد از S1 اجرا شود.
- با توجه به این شرایط، کد مناسبی مبتنی بر سمافور برای اجرای این دو فرآیند ارائه نمایید.

سمافور مشترک به نام Synch = 0 تعریف و کد زیر پیشنهاد می‌شود:

```
P1:
S1 ;
Signal (Synch) ;
```

```
P2:
Wait (Synch);
S2;
```

مثال ۲

• با توجه به قابلیت همروندی و داشتن ۳ فرآیند زیر، چه ترتیب اجرایی می‌تواند رخ دهد؟
فرض کنید دو سمافور S و Q دارای مقدار اولیه صفر هستند.

P1:
Instruction A;
Signal (S) ;

P2:
Wait (Q) ;
Instruction B ;

P3:
Wait (S) ;
Instruction C;
Signal (Q) ;

P1 → P3 → P2

بررسی مشکلات کلاسیک

- **کاربرد همگام سازی**
- **مساله تولید کننده و مصرف کننده**
  - تولید کننده، نوعی داده را تولید کرده و در بافر قرار می دهد.
  - مصرف کننده، داده ها را یکی یکی از بافر برمی دارد.
  - مصرف کننده و تولید کننده می توانند در هر زمان به بافر دسترسی داشته باشند.
  - چند تولید کننده همزمان داده در بافر اضافه نکنند.
  - مصرف کننده و تولید کننده همزمان به بافر دسترسی نداشته باشند.
- **مساله شام فیلسوف ها**
- **مساله خوانندگان و نویسندگان**

تولید کننده و مصرف کننده

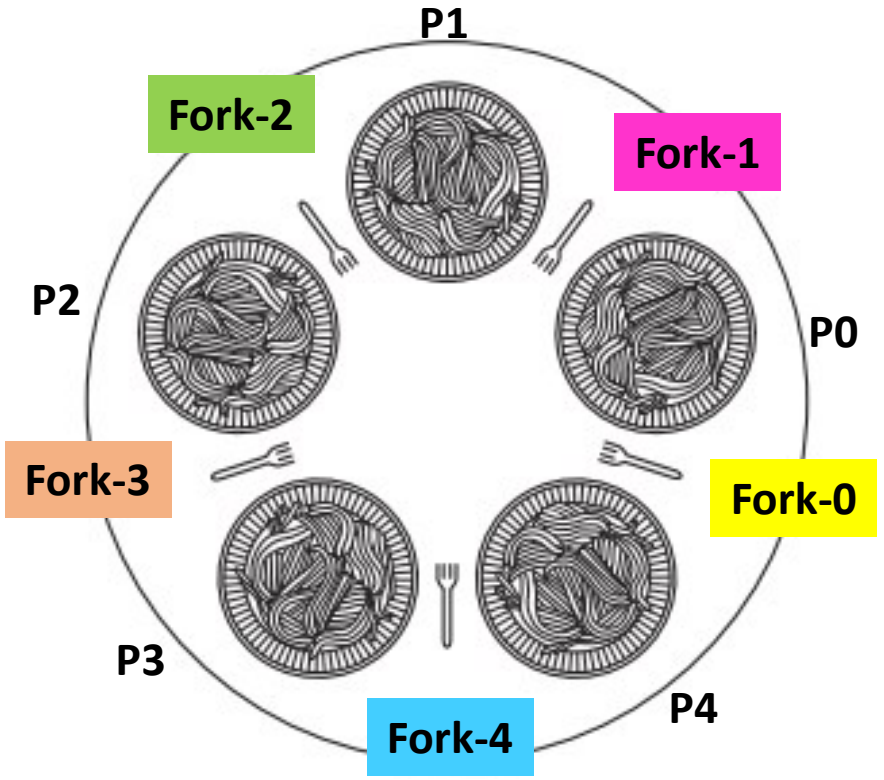
- سمافور mutex
  - با مقدار اولیه ۱، برای انحصار متقابل در نظر گرفته شده است. تولید کننده و مصرف کننده همزمان به بافر دسترسی نداشته باشند.
- سمافور full
  - با مقدار اولیه ۰، برای شمارش تعداد خانه های پر بافر.
- سمافور empty
  - با مقدار اولیه n برای شمارش تعداد خانه های خالی بافر

تولید کننده و مصرف کننده

```
Void producer ()
{
Int item;
While (True) {
        item= produce();
        wait (empty);
        wait (mutex);
        insert (item);
        signal (mutex) ;
        signal (full);
         }
}
```

```
Void consumer ()
{
Int item;
While (True) {
        wait (full);
        wait (mutex);
        item= remove();
        signal (mutex);
        signal (empty);
        consume();
         }
}
```

شام فیلسوف ها



- این مساله شامل ۵ فیلسوف است.
- هر فیلسوف قادر به دو عمل خوردن و فکر کردن است.
- برای عمل خوردن نیاز به دو چنگال است.
- فیلسوف در ابتدا چنگال سمت چپ را بر می دارد و سپس چنگال سمت راست را بر می دارد.
- در این مساله، **انحصار متقابل**، **بن بست** و **گرسنگی** باید بررسی شود.

شام فیلسوف ها

```
Semaphore room=4;
Semaphore fork[5]={1};

Void philosopher (int i) {
 While (True) {
        think();
        wait (room);
        wait (fork [i]);
        wait (fork [ (i+1) % 5]);
        eat ();
        signal (fork [ (i+1) % 5]) ;
        signal (fork [i]);
        signal (room);
         }
}
```

سناریوی زیر را بررسی نمایید:
فرآیندهای زیر به ترتیب وارد صف آماده شده و برای اجرا، CPU در اختیار فرآیندها قرار داده می شود.

حالت اول: **P0, P3, P2, P4, P1**

P0, P3, P2, P4, P1

List

Process	Op	room	Fork[0]	Fork[1]	Fork[2]	Fork[3]	Fork[4]	Fo[0]	Fo[1]	Fo[2]	Fo[3]	Fo[4]	room
		4	1	1	1	1	1						
P0	Eat	3	0	0									
P3	Eat	2				0	0						
p2		1			0	-1					p2		
p4		0					-1					P4	
P1		-1											P1
P0		0	1	1									--
P3		1				0	0				--	--	
p1				0	-1					p1			
p4	eat		0										
P2	eat												
P4		2	1				1						
p2		3			0	1				--			
p1	eat	4		1	1								

P3,P2,P4,P1 **P2,P4,P1** **P4,P1** **P1** **P0,P3** **p1** **P1,p4, p2** **p4, p2** **P2** **p4** **p2** **p1**

مساله خوانندگان و نویسندگان

- این مساله برای شبیه سازی دسترسی به پایگاه‌های داده به کار می رود.
- همزمان چندین خواننده (فرآیند) می توانند از پایگاه داده، بخوانند.
- اگر یک نویسنده (فرآیندی) بخواهد در پایگاه داده بنویسید، در صورت استفاده خوانندگان از پایگاه داده در آن لحظه، نویسنده (فرآیند) اجازه دسترسی نخواهد داشت.
- نویسنده (فرآیند) باید به داده مشترک دسترسی انحصاری داشته باشد.
- با استفاده از سمافور می‌توان آن را پیاده سازی کرد.
  - **متغیر مشترک rc** (برای شمارش تعداد خوانندگان از پایگاه داده)
  - **سمافور db** (برای انحصار متقابل برای نویسنده به کار می رود)
  - **سمافور mutex** (دسترسی به متغیر مشترک rc را به صورت انحصار متقابل امکان پذیر می کند)

مساله خوانندگان و نویسندگان

Semaphore mutex = 1 Semaphore db = 1 int rc = 0

```
Reader :
While (TRUE)
{
    Wait (mutex);
     rc = rc +1;
     if (rc ==1) Wait(db);
     Signal (mutex);
     Read-DataBase ();
     Wait(mutex);
     rc = rc -1;
     if (rc == 0) Signal(db);
     Signal(mutex);
     Use-Data ();
}
```

```
Writer :
While (TRUE)
{
    Wait (db);
     Write-DataBase ();
     Signal (db);
}
```

مثال: بسط الگوریتم پترسون برای N فرآیند

- از مفهومی به نام الگوریتم دروازه ای استفاده می کند (Filter Algorithm)
- هر فرآیند برای ورود به ناحیه بحرانی باید از چند دروازه (سطح) عبور کند.
- اگر فرآیند مشاهده کند که کسی در سطح مساوی یا بالاتر از خودش هست و هنوز خودش آخرین نفر است باید در حالت انتظار بماند.
- ایراد: کارایی کاهش می یابد.

مثال: بسط الگوریتم پترسون برای N فرآیند

```
Define  N
Int Flag [N]
Int Turn [N]
Lock (Process(PID) )  {
        for (i = 0 ; i< N-1 ; i++)  {
                Turn [i] = PID;
                 Flag [PID] = i;
                 While ((exists  k != PID such that Flag [k] >= i &&  Trun [i] == PID) ;
}
Critical Section();
}
Unlock (Process (PID)) {
            Flag [PID] = -1;
}
```

زمانبندی

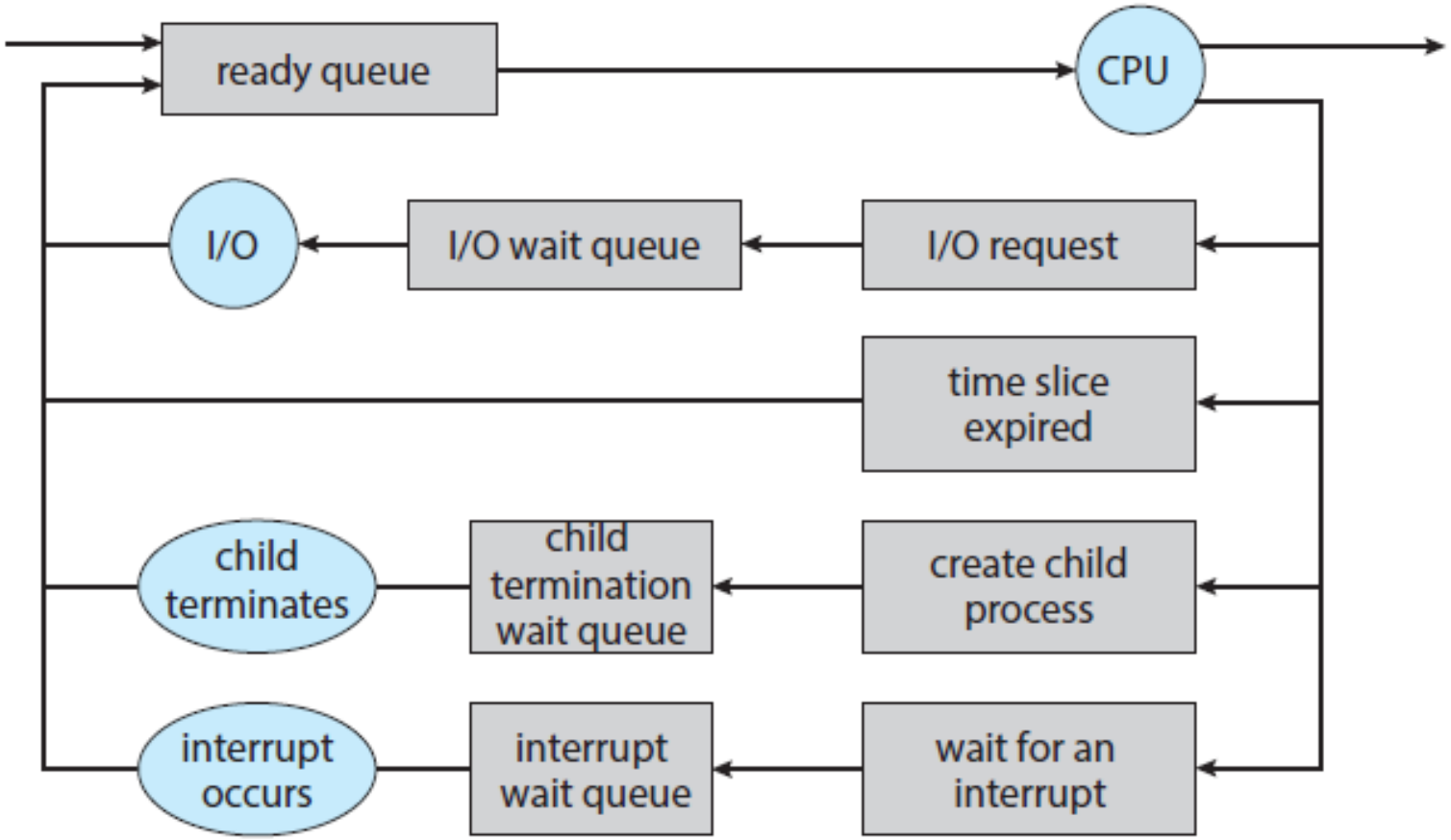
- **تعریف:** در سیستم های چندبرنامگی واقعی، فرآیندهای مختلف بر سر نوبت استفاده از CPU با هم رقابت می کنند. در وضعیتی که بیش از یک فرآیند در حالت آماده است ولی یک CPU وجود داشته باشد، سیستم عامل باید تصمیم بگیرد کدام فرآیند را اول اجرا کند.
- **زمانبند:** بخشی از سیستم‌عامل است تصمیم می‌گیرد کدام فرآیند اول اجرا شود را زمانبند گویند.
  - زمانبند از الگوریتم زمانبندی (Scheduling algorithm) استفاده می کند.

- انواع زمانبندی
  - زمانبند بلند مدت (Long term scheduler)
  - زمانبند میان مدت (Medium term scheduler)
  - زمانبند کوتاه مدت (short term scheduler)

زمانبندی

- زمانبند بلند مدت
  - اینکه کدام فرآیند وارد حافظه اصلی شود را زمانبند بلندمدت گویند. به آن اصطلاحا Job scheduler نیز گویند.
- زمانبند میان مدت
  - وظیفه آن جابجایی فرآیندها بین حافظه اصلی و حافظه ثانویه است.
  - کنترل چند برنامگی را بعهده دارد.
  - به آن swapper هم می گویند.
- زمانبند کوتاه مدت
  - وظیفه آن انتخاب فرآیندی است که باید CPU به آن تعلق بگیرد، به آن CPU scheduler می گویند.
  - تصمیم گیری در بازه های زمانی کوتاه

زمانبندی


ready queue
CPU
I/O
I/O wait queue
I/O request
time slice expired
child terminates
child termination wait queue
create child process
interrupt occurs
interrupt wait queue
wait for an interrupt

زمانبندی

- زمانبندی انحصاری، بدون قبضه، غیر قابل پس گرفتن (non preemptive)
  - فقط هنگامی CPU از پردازش در حال اجرا گرفته می شود که جهت عملیات I/O یا انتظار رخدادی، مسدود شود.

- زمانبندی غیرانحصاری، قابل پس گرفتن، با قبضه (preemptive)
  - این نوع زمانبندی، پس از تمام شدن برش زمانی معین، CPU از پردازش گرفته می شود.

معیارهای زمانبندی

- **عدالت (Fairness)**
  - هر فرآیند سهم عادلانه و منصفانه ای از CPU دریافت کند.
- **بهره وری از CPU (Utilization)**
  - تا انجایی که ممکن است CPU مشغول باشد.
- **بازدهی (Throughput)**
  - تعداد فرآیندهایی که در واحد زمان تکمیل می شوند، بازده نامیده می شود.
- **زمان گردش کار (Turnaround time)**
  - فاصله زمانی بین ارسال یک فرآیند تا زمان تکمیل آن زمان گردش کار نامیده می شود.
  - این زمان شامل مجموع زمانهایی است که فرآیند منتظر حافظه، انتظار در صف آماده، زمان اجرای CPU و زمان لازم برای I/O است.
- **زمان انتظار (Waiting time)**
  - مجموع زمانهایی است که فرآیند در صف آماده منتظر می باشد.
- **زمان پاسخ (Response time)**
  - فاصله زمانی از تقاضای یک کار تا تولید اولین پاسخ را زمان پاسخ گویند.

الگوریتم های زمانبندی

- **سیستم‌های دسته‌ای**
  - الگوریتم FCFS
  - الگوریتم کوتاه ترین کار اول
- **سیستم‌های محاوره‌ای**
  - الگوریتم زمانبندی اولویت
  - الگوریتم نوبت چرخشی
  - الگوریتم صف چندسطحی
  - الگوریتم صف چندسطحی با فیدبک
- **سیستم بلادرنگ**
  - الگوریتم پویا (نرخ یکنواخت)

الگوریتم FCFS

- فرآیندی که اول درخواست CPU می‌کند، CPU اول به آن تخصیص داده می‌شود.
- پیاده‌سازی بوسیله صف FIFO است.
- هر فرآیندی که وارد صف آماده شد به انتهای صف اضافه می‌شود.

- **ویژگی‌ها**
- متوسط زمان انتظار در این الگوریتم طولانی است.
- انحصاری است و گرسنگی ندارد.
- آسان ترین الگوریتم زمانبندی است و سربار ناچیزی دارد، زیرا نیازی به اطلاعات قبلی درباره فرآیندها ندارد. .

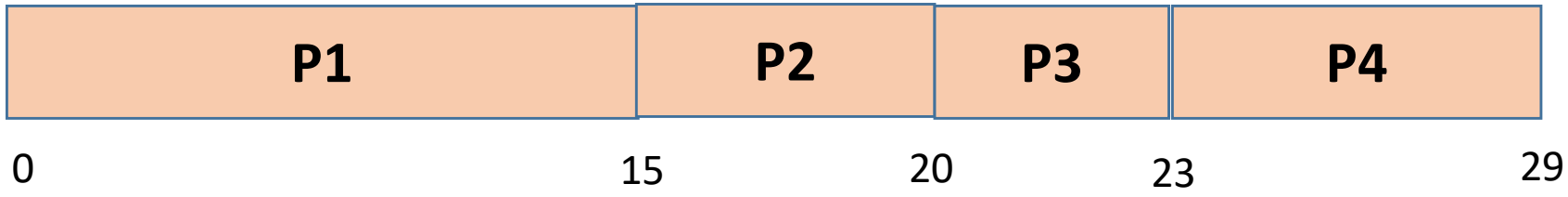
مثال

- فرض کنید فرآیندهای زیر در زمان صفر در صف آماده قرار گرفته‌اند، روش زمانبندی FCFS به کار می‌رود. میانگین زمان انتظار را بدست آورید.

پردازش	زمان پردازش (ms)
P1	15
P2	5
P3	3
P4	6

$$\text{میانگین زمان انتظار} = \frac{(0-0)+(15-0)+(20-0)+(23-0)}{4}$$

$$= 14.5 \text{ ms}$$



نمودار گانت Gantt Chart

الگوریتم کوتاهترین کار اولSJF

- CPU در اختیار فرآیندی قرار می گیرد که کمترین زمانCPU را بگیرد.
- باید زمان اجرای هر فرآیند را از قبل بدانیم.
- اگر دو فرآیند دارای زمان های اجرای مساوی باشند، الگوریتم FIFO اجرا می شود.
- این الگوریتم کوتاهترین زمان متوسط انتظار را دارد.
- این الگوریتم در سیستم‌عامل دسته‌ای قابل پیاده‌سازی است. زیرا هر کاربر حداکثر زمان اجرای آن را اعلام می‌کند.

الگوریتم کوتاهترین کار اولSJF

- الگوریتم SJF می‌تواند انحصاری یا غیرانحصاری باشد.

- الگوریتم SJF را نمی‌توان برای زمانبند کوتاه مدت بکار برد زیرا زمان اجرای فرآیندها از قبل معلوم نیست.
  - یک راه حل، استفاده از پیش بینی تقریبی است.

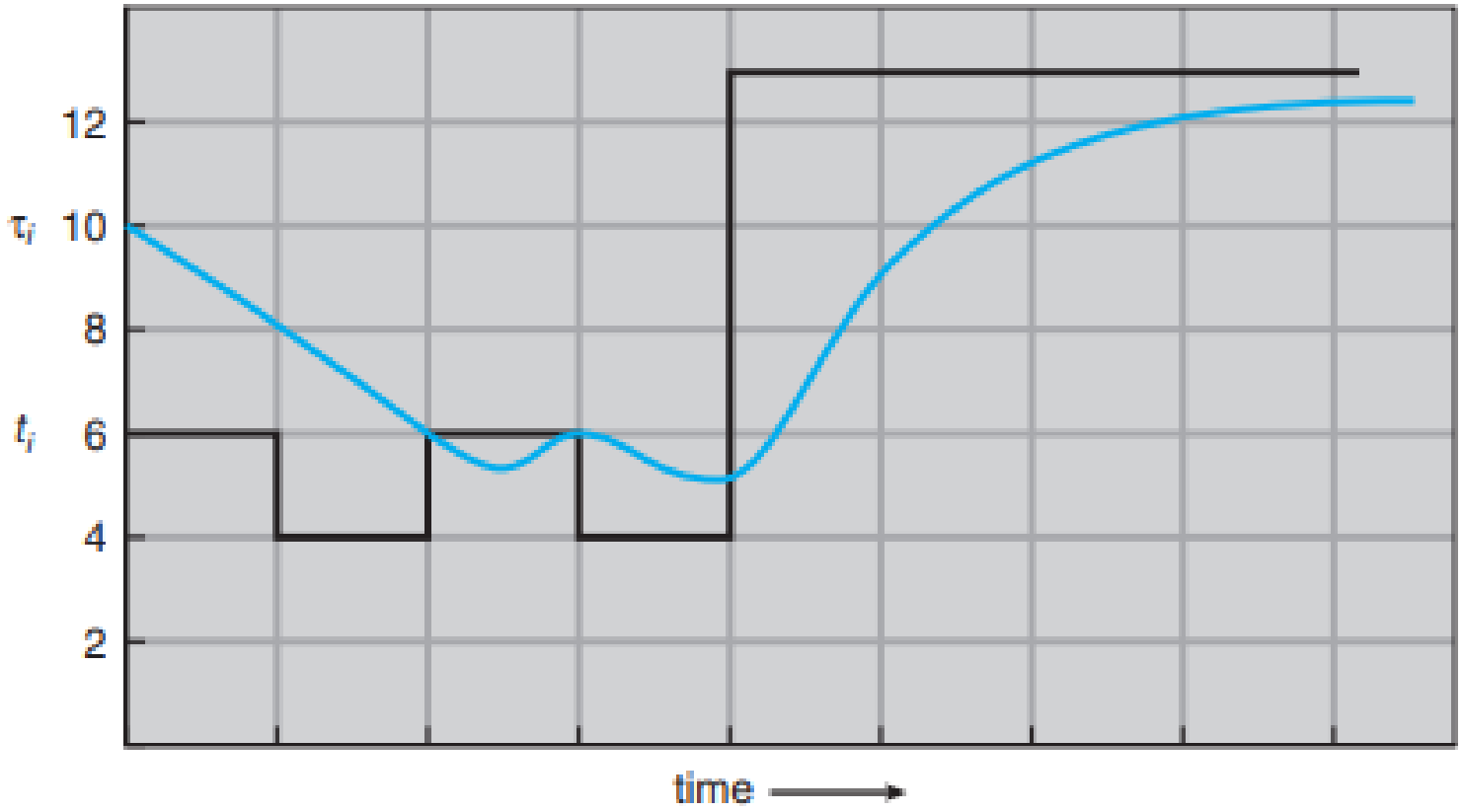
- پیش بینی تقریبی
  - با توجه به زمانی که قبلا فرآیند گرفته، زمان اجرای بعدی فرآیند را با رابطه زیر محاسبه می کنند:
  - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

الگوریتم کوتاهترین کار اولSJF

- پیش بینی تقریبی
  - با توجه به زمانی که قبلا فرآیند گرفته، زمان اجرای بعدی فرآیند را با رابطه زیر محاسبه می کنند:
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

- طول زمان اجرا بعدی به صورت میانگین نمایی طولهای محاسبه شده زمان های محاسباتی قبلی پیش بینی می شود.
- در این رابطه tn طول n امین زمان اجرا است.
- τ_n سابقه گذشته را ذخیره می کند.

مثال (پیش بینی زمان اجرا)



(t_i)		6	4	6	4	13	13	13	...
(τ_i)	10	8	6	6	5	9	11	12	...

مثال

- فرض کنید پیش بینی اولیه ۱۰ باشد. ضریب وزنی را 0.5 و زمان های واقعی اجرا به ترتیب برابر است با ٦، ٨ و ٢.
- مقدار پیش بینی را محاسبه نمایید.
- پاسخ:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$
$$\tau_1 = 0.5 \times 6 + (0.5)10 = 3 + 5 = 8$$

$$\tau_2 = 0.5 \times 8 + (0.5)8 = 4 + 4 = 8$$
$$\tau_3 = 0.5 \times 2 + (0.5)8 = 1 + 4 = 5$$

الگوریتم SRT (Shortest Remaining Time)

- کوتاهترین زمان باقیمانده
  - الگوریتم غیر انحصاری (preemptive) است.
  - مشابه SJF کار می‌کند اما، بر اساس زمان باقیمانده فرآیند کار می کند.
- معایب
  - امکان گرسنگی برای کارهای طولانی
  - باید زمان اجرای کارها مشخص باشد.
- مزایا
  - حداقلی بودن زمان انتظار و زمان برگشت
  - مناسب کارهای اولویت‌داری است که دارای کوتاهترین زمان اجرا هستند.

الگوریتم نوبت چرخشی (Round Robbin-RR)

- شبیه به الگوریتم FCFS است با این تفاوت که به هر پردازش حداکثر به میزان مشخصی CPU را در اختیارش قرار می‌دهد.
  - به این مقدار زمانی، کوانتوم زمانی (time quantum) یا برش زمانی (time slice) می‌گویند.

- اگر اجرای یک فرآیند در مدت کوانتوم زمان به اتمام نرسد، تایمر یک وقفه به سیستم‌عامل می‌دهد و سیستم عامل با (context switching)، CPU را از فرآیند گرفته و فرآیند به انتهای صف آماده منتقل می‌شود تا مجدد در نوبت خود، CPU به آن تخصیص داده شود.

  - همچنین از ابتدای صف آماده، فرآیند جدیدی برای اجرا انتخاب می شود.

الگوریتم بالاترین نسبت پاسخ (Highest Response – HRRN Ratio Next)

- نوعی زمانبندی انحصاری (non preemptive) است.
- برای برطرف کردن معایب الگوریتم زمانبندی SJF پیشنهاد شد.
- در این روش، اولویت‌ها دینامیک هستند و مطابق رابطه زیر تعیین می‌شوند:

$$\text{اولویت} = \frac{\text{زمان انتظار} + \text{زمان سرویس}}{\text{زمان سرویس}}$$

- کارهای کوتاهتر اولویت بیشتری دارند و زودتر اجرا می‌شوند. اما با افزایش زمان انتظار، کارهای طولانی نیز اولویت بیشتری کسب می‌کنند و می‌توانند اجرا شوند و مساله قحطی‌زدگی رخ نمی‌دهد.

الگوریتم زمانبندی اولویت (Priority scheduling)

- به هر فرآیند یک اولویت داده می شود و CPU به فرآیندی که دارای بالاترین اولویت است داده می شود.

- اگر اولویت ها یکسان باشد زمانبندی به صورت FCFS خواهد بود.

- اولویت فرآیندها با شماره هایی که به آنها داده می شود ، مشخص می گردد.

- الگوریتم می تواند انحصاری یا غیرانحصاری باشد.

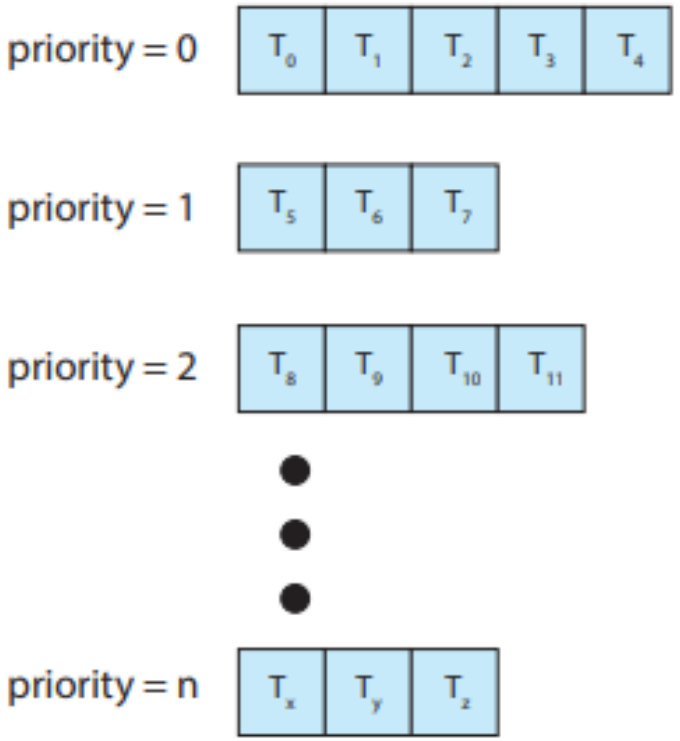
الگوریتم زمانبندی اولویت

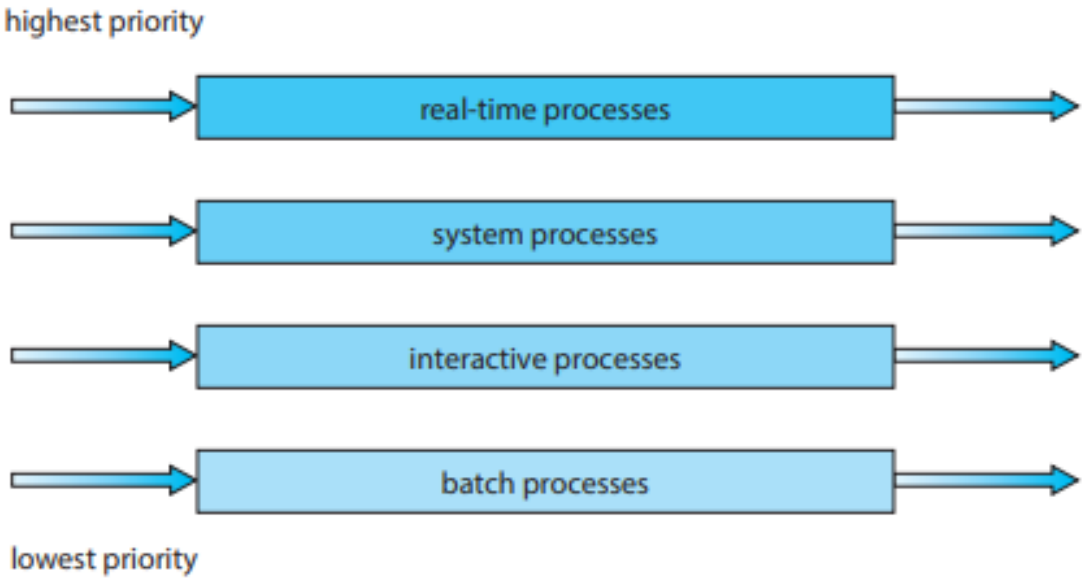
- اولویت فرآیند ممکن است به صورت خارجی یا داخلی تعریف شود.
- اولویت داخلی مبتنی است بر:
  - زمان اجرا، مقدار حافظه مورد نیاز برای فرآیند، تعداد فایل هایی که باز می کند.
- اولویت خارجی مبتنی است بر:
  - مهم بودن فرآیند، فرآیند مربوط به چه شخصی یا چه ارگانی است، یا بوسیله نظرات سیاسی.

- مساله قحطی زدگی در الگوریتم زمانبندی اولویت
- فرآیندهایی با اولویت بالا ممکن است نگذارند که فرآیندهای با اولویت پایین هرگز CPU را در اختیار بگیرند.
  - **راه حل: فرآیندهایی که دارای اولویت کمی هستند بعد از اینکه مدت زیادی منتظر بمانند، به تدریج اولویت آنها را افزایش می‌دهند. مثلا به ازای گذشت هر بازه زمانی، اولویت فرآیندها افزایش می‌یابد. به این تکنیک ageing گویند.**

الگوریتم صف چندسطحی (Multi Level Queue-MLQ)

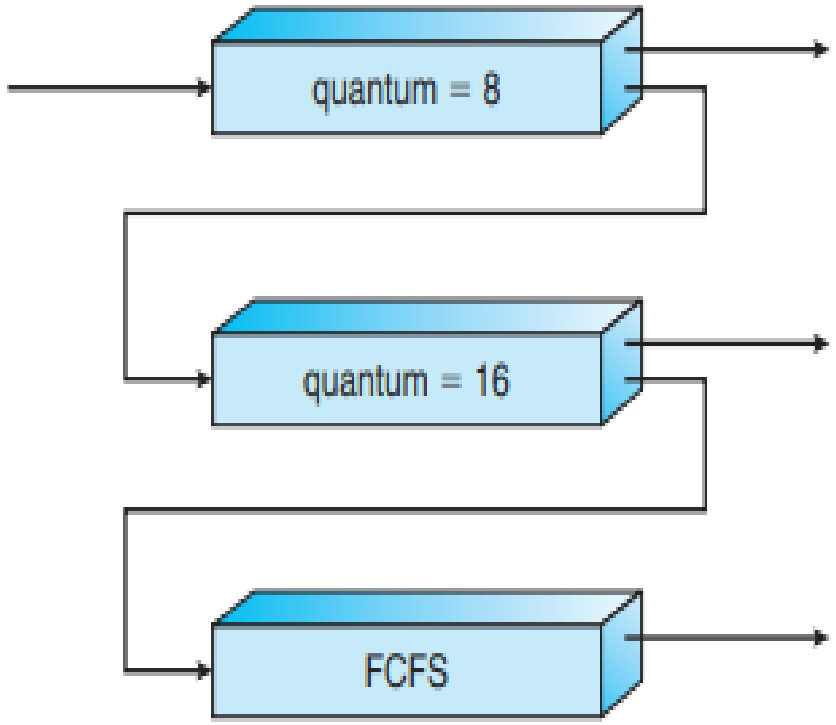
- زمانی که بتوان فرآیندها را در دسته های متفاوتی طبقه بندی کرد، این روش مناسب است.
- هر فرآیند در صف خاصی قرار می گیرد و هر صف برای خود از روش زمانبندی متفاوتی استفاده می کند. اولویت صف ها نیز متفاوت می باشد.





الگوریتم صف چندسطحی با فیدبک (MLFQ)

- الگوریتم غیرانحصاری (preemptive)
- این روش اجازه می دهد فرآیندها از صفی به صف دیگر منتقل شوند.
- از سناریوی های مختلفی می توان استفاده کرد.



مثال:

فرآیندهای صف ۱ اجرا می شوند.
سپس فرآیندهای صف ۲ اجرا می شوند.
در پایان نیز فرآیندهای صف ۳، اجرا می شوند.
برای مثال اگر فرآیندی در صف ۱، اجرای آن خاتمه نیافت به انتهای صف ۲ اضافه می‌شود. همین شرایط برای صف سوم نیز برقرار است.

زمانبندی شانسی (Lottery Scheduling)

- سیستم‌عامل به هر فرآیند تعدادی عددی نسبت می‌دهد. هر چه اولویت فرایند بیشتر باشد، تعداد اعداد بیشتری به آن نسبت داده می‌شود.
- سپس یک عدد تصادفی تولید شده و فرآیندی که آن عدد را در اختیار داشته باشد، می‌تواند CPU را در اختیار داشته باشد.
- هر چه اولویت فرآیند بیشتر باشد، شانس بیشتری برای داشتن CPU را خواهد داشت.

زمانبندی بلادرنگ

- سیستم بلادرنگ (Real time)، زمان نقش مهمی دارد.
  - بلادرنگ سخت
  - بلادرنگ نرم
- فرآیندها در سیستم بلادرنگ عمر کوتاهی دارند.
- وقایع در سیستم بلادرنگ
  - وقایع متناوب (Periodic Events)
    - در فواصل زمانی مساوی رخ می دهد.
  - وقایع نامتناوب (Aperiodic Events)
    - به صورت تصادفی و غیرقابل پیش بینی رخ می دهد.

زمانبندی بلادرنگ

وقایع دوره ای باعث فعال‌سازی وظایف دوره‌ای (Periodic Tasks) می‌شود. مانند:
نمونه برداری از دما در هر 100ms
نکته: به دلیل منظم بودن، قابلیت پیش بینی بالایی دارند و تحلیل زمانبندی در آنها ساده تر است.

وقایع غیردوره‌ای (Aperiodic) باعث فعال‌سازی وظایف غیرمتناوب می‌شود. مانند:
ورود یک بسته شبکه با زمان نامشخص
نکته: مهلت زمانی سخت ندارند، اما تا حد امکان باید سریع پاسخ داده شوند.

زمانبندی بلادرنگ

- الگوریتم های زمانبندی بلادرنگ
  - پویا: در زمان اجرا تصمیم های زمانبندی (اعمال تقدم ها) انجام می شود.
  - ایستا: تصمیمات زمانبندی (اعمال تقدم ها) قبل از شروع اجرای سیستم انجام می شود.
- **الگوریتم های زمانبندی پویا (با تقدم ثابت)**
- الگوریتم نرخ یکنواخت (Rate Monotonic-RM)
- **الگوریتم های زمانبندی پویا (با تقدم پویا)**
- الگوریتم ابتدا زودترین مهلت (Earliest deadline first-EDF)
- الگوریتم کمترین سستی (Least laxity)

زمانبندی بلادرنگ

- الگوریتم نرخ یکنواخت (Rate Monotonic-RM)
  - زمانبندی ثابت اولویت است.(fixed priority)
- به هر فرآیند یک اولویت متناسب با فرکانس رخداد واقعه راه انداز آن نسبت داده می شود.
- زمانبند فرآیند با بالاترین اولویت را انتخاب می کند.
- غیرانحصاری است.
- فرکانس بالاتر، اولویت بالاتری فرآیند خواهد داشت.
- الگوریتم بهینه است.

زمانبندی بلادرنگ (مثال)

- الگوریتم نرخ یکنواخت (Rate Monotonic-RM)

- P1: T1=20 ns
- P2: T2= 60 ns
- P3: T3 = 40 ns
  - در این مثال، برای نمونه به P1 تقدم ۱۰ داده می شود. به فرآیند P3 تقدم ۵ و به P2 تقدم ۲ داده می‌شود.
  - تقدم ها ثابت هستند.
- الگوریتم ابتدا زودترین مهلت (Earliest deadline first-EDF)
  - فرآیندها بر اساس مهلت زمانی خود مرتب می شوند.
  - زمانبند فرآیندی را انتخاب می کند که فرصتش از همه کمتر است (نزدیکترین مهلت را دارد.)
  - الگوریتم اولویت پویا است. (dynamic priority)

زمانبندی بلادرنگ

- الگوریتم ابتدا زودترین مهلت (Earliest deadline first-EDF)
  - فرآیندها بر اساس مهلت زمانی خود مرتب می شوند.
  - زمانبند فرآیندی را انتخاب می کند که فرصتش از همه کمتر است (نزدیکترین مهلت را دارد.)
  - الگوریتم اولویت پویا است. (dynamic priority)

زمانبندی بلادرنگ (مثال)

- الگوریتم ابتدا زودترین مهلت (Earliest deadline first-EDF)

- P1: deadline = 20 ns
- P2: deadline = 30 ns
- P3: deadline = 10 ns

  - تقدم P3 بالاترین است، سپس تقدم P1 و در پایان تقدم P2.
  - تقدم ها پویا هستند.

زمانبندی بلادرنگ

- الگوریتم کمترین سستی (Least laxity)
- **سستی:**
- برای همه فرآیندها، مقدار زمان باقیمانده محاسبه می‌شود. این مقدار، مدت زمانی است که فرآیند می تواند آماده باقی بماند و اجرا نشود. این زمان را سستی می‌گویند.
- مثال: فرآیندی 200 ms زمان اجرا نیاز دارد. 250 ms زمان مهلت دارد که اجرا شود، مقدار زمان سستی برابر 50 ms است.

- فرآیندی را انتخاب می‌کند که کوچکترین زمان سستی را دارد

زمانبندی بلادرنگ

در سیستم‌های بلادرنگ، مجموعه‌ای از فرآیندها زمانی می‌توانند زمانبندی شوند که مقدار بهره‌وری CPU، در رابطه زیر برقرار باشد.

- Ci: مدت زمان اجرای فرآیند i برروی CPU
- Ti: دوره تناوب فرآیند
- N: تعداد فرآیندهای موجود

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

بن‌بست (Deadlock)

- یک فرآیند برای اجرا به مجموعه ای از منابع نیاز دارد. در محیط چندبرنامه ای اگر منابع آزاد نباشند، فرآیند باید منتظر باشد تا منبع مورد نیازش آزاد گردد. اگر این انتظار بی نهایت طول بکشد، **بن بست** رخ داده است.

- مدل سیستم
  - یک سیستم کامپیوتر شامل تعداد محدودی از منابع مانند حافظه، CPU، فایل و دستگاه های ورودی و خروجی است که میان فرآیندها تقسیم می شوند.

- هر فرآیند برای استفاده از منبع، مراحل زیر را دنبال می کند:
  - درخواست (request که با فراخوانی سیستم انجام می‌شود.)
  - استفاده از منبع به عنوان اجرا شدن فرآیند
  - آزاد کردن (release که با فراخوانی سیستم انجام می شود.)

بن‌بست (Deadlock)

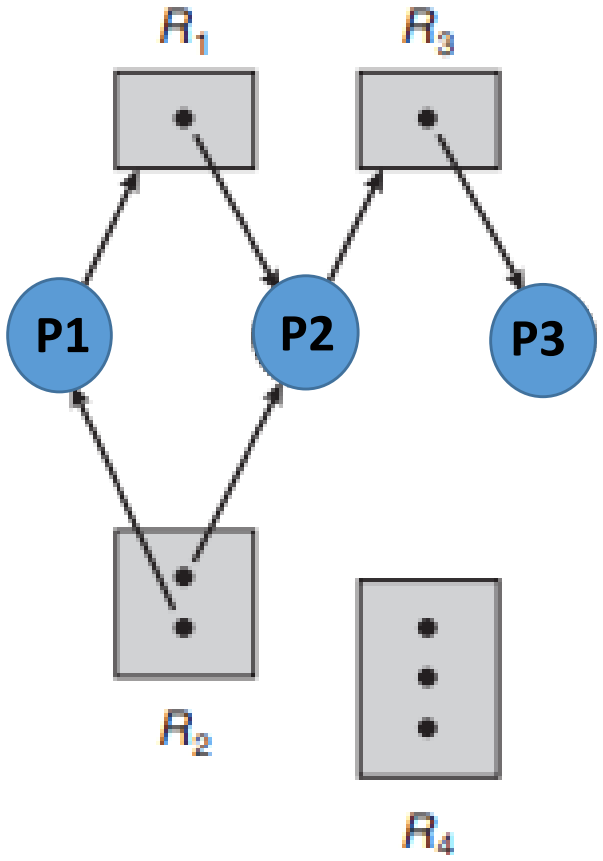
مجموعه‌ای از فرآیندها هستند که هر کدام یک منبع را گرفته‌اند و منتظر منبع دیگری هستند که آن منبع در اختیار فرآیند دیگری قرار دارد. فرآیندها نمی‌توانند ادامه دهند و سیستم در حالت بن‌بست قرار دارد.

شرایط لازم برای بن‌بست

- کافمن نشان داد که، در صورت برقراری ٤ شرط زیر بطور همزمان، بن‌بست می‌تواند رخ دهد: (شرط لازم اما کافی نیست)

1. انحصار متقابل
    1. یک منبع حداقل به صورت غیر اشتراکی وجود دارد که در هر لحظه فقط یک فرآیند می‌تواند از آن استفاده نماید. فرآیندها دیگر، در صورت نیاز باید درخواست شان به تعویق بیفتد.
2. نگهداری و منتظر ماندن
    1. فرآیند منبعی را در اختیار دارد و می‌تواند برای منابع جدید درخواست دهد. در واقع منتظر گرفتن منابع دیگری است که در اختیار دیگر فرآیندها هست.
3. غیرقابل پس گرفتن
    1. نمی توان منبع را از فرآیندی پس گرفت و باید کار فرآیند به اتمام برسد که منبع را آزاد کند.
4. انتظار چرخشی
    1. زنجیره ای از فرآیندها وجود دارد و هر کدام منتظر منبعی است که توسط عضو دیگر زنجیره، قبضه شده است.

مدل‌سازی بن‌بست (گراف تخصیص منابع)



- مجموعه ای از فرآیندها P و منابع R را داریم.

P= {P0, P1, P2, ...}

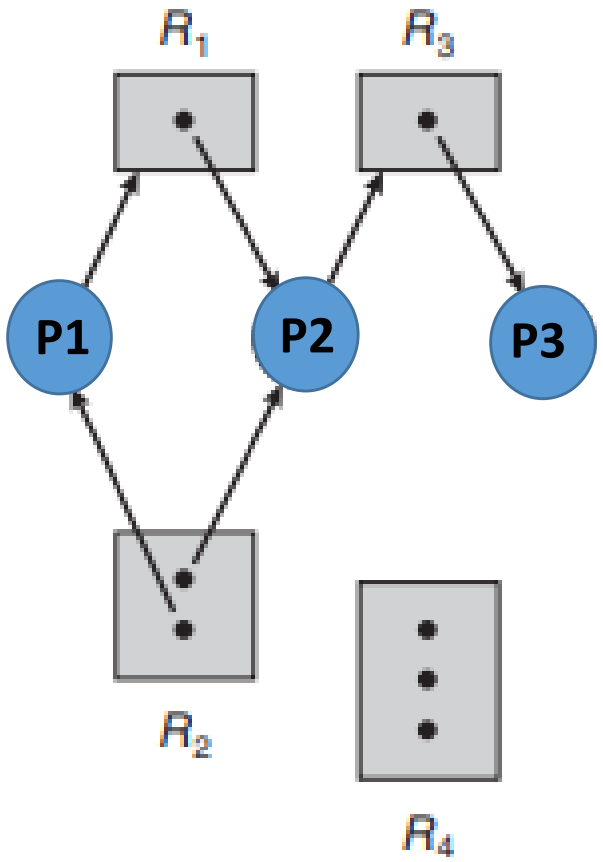
R= {R1, R2, R3, ...}

- در این گراف، رئوس عبارت است از فرآیندها و منابع.
- فرآیندها با دایره و منابع با مستطیل ترسیم می‌کنیم.
- بردار E شامل مجموعه‌ای از یال‌ها است.

E= {P1→R1, P2→R3, R1→P2, R2→P1, R3→p3}

- بطور کلی یال Pi→Rj یعنی فرآیند Pi درخواست دریافت منبع Rj را دارد.
- یال Ri→Pj یعنی منبع Ri در اختیار فرآیند Pj قرار دارد.

گراف تخصیص منابع

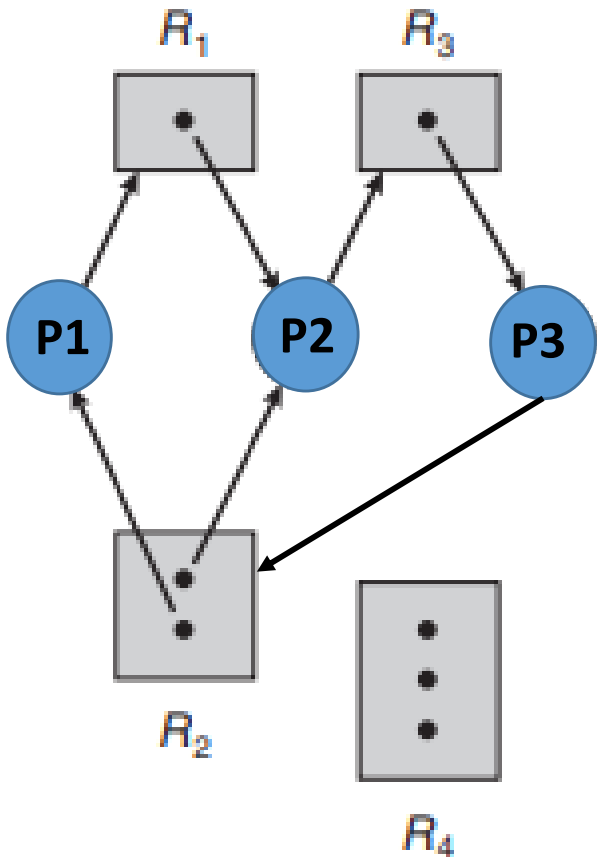


ا داشتن گراف تخصیص منابع می‌توان نشان داد که اگر گراف سیکل ندارد در این صورت هیچ فرآیندی در سیستم در بن‌بست نیست و اگر گراف دارای سیکل است امکان وجود بن‌بست داریم.

گر از هر نوع منبع، فقط یک نمونه وجود داشته باشد، در این صورت وجود سیکل یعنی بن‌بست رخ داده است.

- تصویر مقابل سیکل ندارد. پس در آن بن‌بست رخ نمی‌دهد.

گراف تخصیص منابع



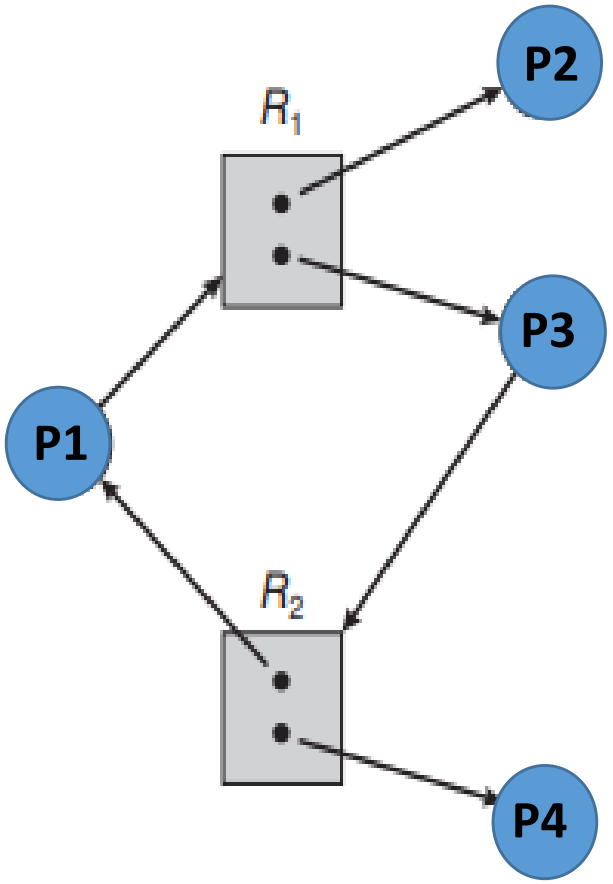
- تصویر مقابل سیکل دارد. پس در آن ممکن است بن‌بست رخ دهد.
- سیکل‌ها عبارت است از:

P1→R1→P2→R3→P3→R2→P1

P2→R3→P3→R2→P2

با بررسی گراف، بن‌بست رخ می‌دهد.

گراف تخصیص منابع



- تصویر مقابل سیکل دارد. ممکن است در آن بن‌بست رخ دهد.
- سیکل عبارت است از:

P1→R1→P3→R2→P1

در اینجا بن‌بست رخ نمی‌دهد. مثلا:

(۱). هر وقت فرآیند P4 به اتمام رسید منبع R2 را آزاد (رها) می‌کند.

(۲). فرآیند P3 که منبع R2 را تقاضا کرده است می‌تواند از منبع R2 آزاد شده استفاده کند. بعد از اتمام P3 منبع R2 و R1 رها می‌شود.

(۳). فرآیند P1 نیز می‌تواند اجرا شود. پس بن‌بست نداریم.

مدیریت بن‌بست

(۱). پیشگیری یا جلوگیری از بن‌بست (deadlock prevention)

- روشی باید به کار گرفته شود که یکی از چهار شرط لازم برای بن‌بست رخ ندهد.

(۲). اجتناب یا پرهیز از بن‌بست (deadlock avoidance)

- درخواست‌ها و آزادشدن منابع آتی بررسی شده و تصمیم گرفته می‌شود که آیا درخواست جاری اگر پاسخ داده شود منجر به بن‌بست خواهد شد یا نه

(۳). کشف یا تشخیص بن‌بست (deadlock detection)

- می‌تواند بن‌بست رخ دهد. الگوریتمی ارائه می شود که بن‌بست را کشف کند.

(٤). ترمیم بن‌بست (deadlock recovery)

- از بین بردن فرآیندی در چرخه (Kill)، پس گرفتن منبع

- **(۵) نادیده گرفتن مساله**
  - الگوریتم شترمرغ (Ostrich)
  - هیچ عملی انجام نمی‌شود و در صورت رخ دادن بن‌بست و قفل شدن سیستم، آن را می‌توان دستی ریست کرد.

مدیریت بن بست ((۱). پیشگیری یا جلوگیری از بن‌بست)

- در این دیدگاه، تلاش می‌شود که شرایط چهارگانه که ضرورت بن‌بست هست را لغو کند.

  - منابع به صورت اشتراکی استفاده شوند. مثلا خواندن همزمان یک فایل توسط چند فرآیند، انحصار متقابل را لغو می‌کند.
  - هر پردازش قبل از آغاز اجرا، کلیه منابعش را درخواست کرده و در صورتی که همه آنها را در ابتدا داشت، اجرا می‌شود.
  - اگر پردازشی تقاضای منبعی داشت اما سیستم قادر به تخصیص آن منبع نباشد، دیگر منابع که در دستش هست را باید رها کند.
  - برای جلوگیری از انتظار چرخشی، به هر منبع یک شماره صحیح نسبت داده می شود. هر فرآیند فقط می‌تواند منابع را در جهت صعودی شماره هایش درخواست کند.

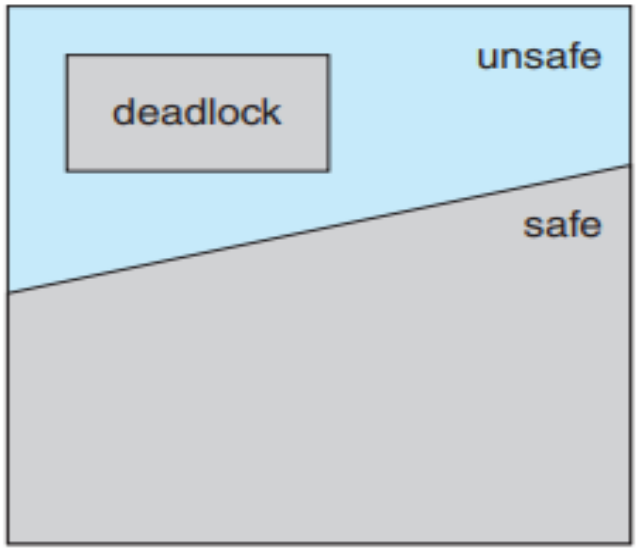
- معایب
  - بهره وری پایین
  - قحطی زدگی (Starvation)

این روش ها، همیشه کاربرد ندارند

مدیریت بن بست ((۲). اجتناب از بن‌بست)

- در این دیدگاه، کاری می‌کند که سیستم همواره در حالت امن باشد.
  - اگر سیستم بتواند منابع مورد درخواست را به ترتیبی تخصیص دهد که از بروز بن‌بست اجتناب شود، می‌گوییم سیستم امن است.
  - اگر سیستم ناامن باشد، امکان رخ دادن بن‌بست وجود دارد.

- در الگوریتم اجتناب از بن بست، هرگاه فرآیندی تقاضای منبعی را کند سیستم باید تصمیم بگیرد که آیا فورا منبع به آن تخصیص داده شود یا فرآیند باید صبر کند.



یکی از الگوریتم های اجتناب از بن بست، الگوریتم بانکداران است (Banker’s Algorithm)

الگوریتم بانکداران

- الگوریتم شبیه رفتار یک بانکدار با مشتریانش است. یک بانکدار تمام سرمایه خودش را به مشتریان تخصیص نمی‌دهد و طوری عمل می‌کند که بتواند نیازهای کلیه مشتریانش را برآورد کند.
- **فرضیات الگوریتم**

تعداد فرآیند: n تعداد انواع منابع: m

بردار Available : بردار به طول m که تعداد منابع از هر نوع را نشان می‌دهد.

ماتریس Max : در اندازه n×m، ماکزیمم نیاز هر فرآیند را نشان می‌دهد.

ماتریس Allocation :در اندازه n×m، تعداد منابع از هر نوع که به هر فرآیند تخصیص یافته را بیان می‌کند.

الگوریتم بانکداران

- محاسبه ماتریس Need در اندازه n×m، که نشان‌دهنده نیازهای حال حاضر است

Need = Max - Allocation

1) در ماتریس Need به دنبال سطری هستیم که کوچکتر از Available باشد.
2) اگر چنین سطری وجود داشت، تخصیص انجام شده، در غیراین‌صورت، الگوریتم خاتمه می‌یابد.
3) بعد از اتمام اجرا، منابع آزاد می‌شوند و بروزرسانی بردار Available را داریم.
4) تکرار مراحل ۱ تا ۳، تا زمانی که همه فرآیندها خاتمه یابند.
5) پایان

مثال

- سیستمی با ۳ منبع A، B و C که به ترتیب از هر کدام، ۱۰، ۵ و ۷ عدد داریم. با فرض ۵ فرآیند و داشتن وضعیت زیر در زمان t0، آیا سیستم در حالت امن قرار دارد؟
- **پاسخ**
- با الگوریتم بانکداران وضعیت سیستم را بررسی می کنیم.

Allocation

Pi	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
p4	0	0	2

Max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

مثال

- Available <10-2-3-2, 5-1-1, 7- 2-1-2> = <3, 3, 2>
- Need = Max- Allocation
- P1, P3 < Available

Allocation

Pi	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
p4	0	0	2

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Max

Need

Pi	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
p4	4	3	1

مثال

- Available <10-2-3-2, 5-1-1, 7- 2-1-2> = <3, 3, 2>

- تخصیص منابع به P1 و اجرای P1 .
- رها کردن منابع و بروزرسانی Available = <5, 3, 2>
- P3, p4 < Available
- تخصیص منابع به P3 و اجرای آن.
- رها کردن منابع و بروزرسانی Available = <7,4,3>، سپس تخصیص و اجرای P4 .
- رها کردن و بروزرسانی منابع Available = <7, 4, 5>
- اجرای P2 و رهاسازی منابع و بروز رسانی، Available =< 10, 4, 7 > و در نهایت اجرای P0

Allocation

Pi	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
p4	0	0	2

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Max

Need

Pi	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
p4	4	3	1

مدیریت بن‌بست ((۳). کشف بن‌بست)

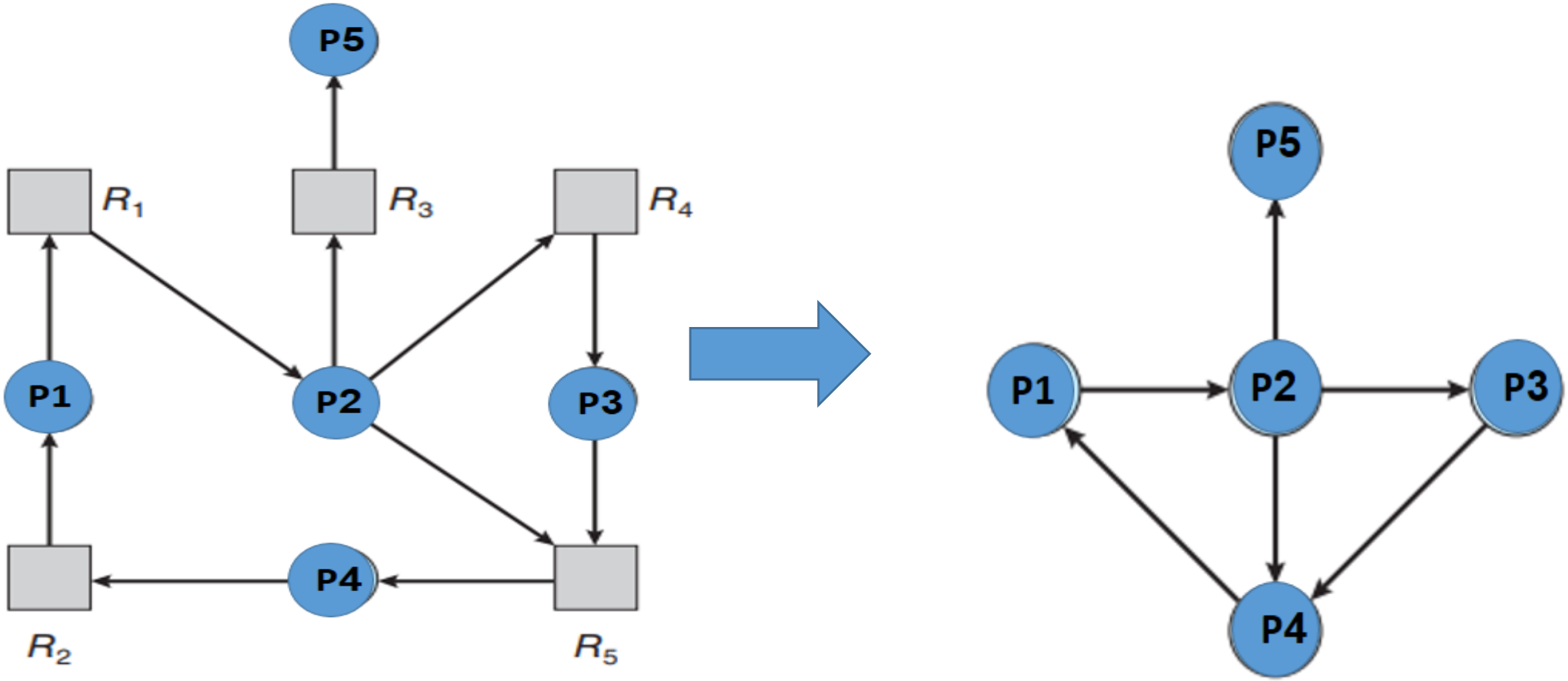
- در این رویکرد ممکن است بن بست رخ دهد. بنابراین سیستم بررسی می کند که آیا بن بست رخ داده است یا نه.

- **دو روش برای تشخیص بن بست وجود دارد:**

1. رویکرد مبتنی بر گراف انتظار در حالت وجود یک منبع از هر نوع
2. رویکردی مشابه با الگوریتم بانکداران در حالت وجود چند منبع از هر نوع

کشف بن بست (گراف انتظار)

از گراف تخصیص منابع، گراف انتظار را ترسیم می‌کنیم.
گره‌های منبع حذف شده و یال‌ها را به طور صحیحی ترکیب می‌کنیم.
اگر درگراف انتظار حلقه وجود داشته باشد، بن‌بست وجود دارد



کشف بن بست (الگوریتم بانکداران)

- مثال: سیستمی با ۵ فرآیند و سه نوع منبع مختلف A، B و C که از هر کدام ۷، ۲ و ٦ عدد داریم، را در نظر بگیرید. اگر ماتریس تخصیص Allocation و ماتریس نیازهای فعلی هر فرآیند (به نام ماتریس Request) به صورت زیر باشد. آیا سیستم در حالت بن بست است یا خیر؟ .

Allocation

Pi	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
p4	0	0	2

Request

Pi	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
p4	0	0	2

رویکردی مبتنی بر الگوریتم بانکداران

- پاسخ:

Available = <0, 0, 0>

Allocation

Pi	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
p4	0	0	2

P0 : <0, 0, 0> ➔ اجرا→ Available=<0, 1, 0>

P2: <0, 0, 0> → اجرا→ Available =<3, 1, 3>

P3: <1, 0, 0>→ اجرا→ Available = <5, 2, 4>

P1: <2, 0, 2>→ اجرا→ Available =<7, 2, 4>

P4: <0, 0, 2> → اجرا→ Available = <7, 2, 6>

Request

Pi	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
p4	0	0	2