«به نامِ نگارنده‌یِ الگوریتمِ هستی؛ که نظمِ خلقت، زیرِ نگاهِ اوست.»

جزوه آموزشی سیستم عامل

استاد:

شاهرخ امانی

بازنویسی و تنظیم:

پریا امانی

سال تحصیلی:1404-1405

فصل اول: مفاهیم پایه و ساختار سیستم عامل

- تعریف سیستم عامل و مدیریت منابع
- انواع سیستم عامل: (تک‌کاربره، چندکاربره، توزیع‌شده و Real-Time)
- معماری سخت‌افزار: چرخه CPU و ثبات‌های PC و PSW
- مفهوم وقفه (Interrupt): روال ISR، پشته (Stack) و آدرس بازگشت
- هسته سیستم عامل (Kernel) و (Context Switch)

فصل دوم: مدیریت و چرخه حیات فرآیندها

- روش‌های ورودی/خروجی: (Programmed, Polling, Interrupt, DMA)
- انواع فرآیند CPU-Bound و I/O-Bound و شناسنامه فرآیند PCB
- نمودار حالات فرآیند: تولد، آماده، اجرا، انتظار، پایان و مفهوم تعلیق/ Suspend
- مفهوم نخ : Threadموازی‌سازی، نخ‌های سطح کاربر و هسته

فصل سوم: زمان‌بندی پردازنده (CPU Scheduling)

- معیارهای ارزیابی
- انواع زمان‌بندی: انحصاری (Non-Preemptive) و غیرانحصاری (Preemptive)
- الگوریتم‌های زمان‌بندی:
- الگوریتم FCFS و LCFS
- الگوریتم نوبت‌دهی چرخشی (Round Robin)
- الگوریتم‌های SJF و SRT
- صف چندسطحی بازخوردی (MLFQ) و اولویت‌بندی

فصل چهارم: بن‌بست (Deadlock)

- شرایط وقوع و روش‌های پیشگیری: (حذف تصاحب و انتظار، حذف عدم قبضه)
- حذف انتظار چرخشی: ترتیب خطی منابع (Linear Ordering)
- اجتناب از بن‌بست: الگوریتم بانکداران

فصل پنجم: همگام‌سازی و انحصار متقابل

- ناحیه بحرانی (Critical Section) و چالش‌های نرم‌افزاری: (الگوریتم دکر و راه‌حل‌های ناقص)
- حمایت سخت‌افزاری: دستورات TSL، SWAP و از کار انداختن وقفه‌ها
- سمافور (Semaphore): * توابع Wait و Signal
- انواع سمافور (عمومی و دودویی) و انحصار متقابل با سمافور
- مسائل کلاسیک همگام‌سازی: (مثل خوانندگان و نویسندگان)

فصل اول

مفهوم سیستم عامل

۱. سیستم عامل (Operating System) چیست؟

سیستم عامل واسطی (Interface) است بین کاربر و سخت‌افزار. بدون سیستم عامل، شما باید برای هر بار استفاده از هارد یا کارت گرافیک، کدهای پیچیده باینری می‌نوشتید.

- **وظیفه اصلی:** مدیریت منابع (CPU، حافظه، دیسک) و فراهم کردن محیطی برای اجرای نرم‌افزارها.

- **مثال:** سیستم عامل مثل مدیر یک هتل است. مسافران (نرم‌افزارها) لازم نیست بدانند آشپزخانه چطور کار می‌کند؛ آن‌ها فقط سفارش می‌دهند و مدیر هتل هماهنگ می‌کند که چه کسی، چه زمانی و از کدام اتاق استفاده کند.

۲. انواع سیستم عامل (OS Types)

الف) تک‌کاربره (Single-user)

سیستمی که در هر لحظه فقط یک نفر می‌تواند از آن استفاده کند (مثل DOS یا نسخه‌های قدیمی ویندوز).

ب) چندکاربره (Multi-user)

سیستمی که اجازه می‌دهد چندین کاربر هم‌زمان به منابع دسترسی داشته باشند (مثل سرورهای دانشگاه یا لینوکس). سیستم عامل باید مراقب باشد که اطلاعات کاربران با هم قاطی نشود.

ج) توزیع شده (Distributed)

در این مدل، محاسبات روی چندین کامپیوتر مجزا انجام می‌شود اما کاربر فکر می‌کند فقط با یک سیستم کار می‌کند.

• مثال: وقتی در گوگل سرچ می‌کنید، هزاران کامپیوتر در حال پردازش هستند، اما شما فقط یک صفحه ساده می‌بینید.

د) درنگ‌زمان (Real-Time)

در این سیستم‌ها، «زمان» حیاتی است. پاسخ باید در یک بازه زمانی دقیق داده شود، وگرنه فاجعه رخ می‌دهد.

• **مثال:** سیستم کنترل ترمز ABS ماشین یا هدایت موشک. اگر سیستم عامل ترمز با ۲ ثانیه تاخیر عمل کند، فاجعه رخ می‌دهد.

۳. چرخه کار CPU (Fetch-Decode-Execute)

هر دستوری که در کامپیوتر اجرا می‌شود، این سه مرحله را طی می‌کند:

1. واکشی (Fetch): دستور از حافظه (RAM) برداشته شده و وارد CPU می‌شود.

2. رمزگشایی CPU : (Decode) می‌فهمد این دستور چیست؟ (جمع است؟ ضرب است؟ یا انتقال داده؟).

3. اجرا: (Execute) عملیات انجام می‌شود.

۴. ثبات‌های حیاتی (Registers)

ثبات وضعیت برنامه (PSW - Program Status Word)

این یک ثبات بسیار مهم است که وضعیت فعلی پردازنده را نشان می‌دهد. شامل پرچم‌هایی (Flags) است که می‌گوید: آیا آخرین محاسبه صفر شد؟ آیا خطا رخ داد؟ آیا در حالت کاربر هستیم یا هسته (Kernel) ؟

شمارنده برنامه (PC - Program Counter)

این ثبات، آدرس دستور بعدی که باید اجرا شود را نگه می‌دارد. مثل نشان‌گر کتاب که یادت می‌اندازد الان کدام خط هستی و خط بعدی کجاست.

۵. وقفه (Interrupt) و روال بررسی آن (ISR)

وقفه یعنی متوقف کردن موقت کار فعلی CPU برای رسیدگی به یک رویداد فوری.

• روال بررسی وقفه : (Interrupt Service Routine - ISR) یک قطعه کد کوچک در سیستم عامل است که می‌گوید در صورت بروز هر وقفه، چه کاری انجام شود.

• پشته :(Stack) وقتی وقفه رخ می‌دهد، CPU باید بداند کجا بوده تا بعداً برگردد. اطلاعات فعلی (مثل مقدار ثبات‌ها) را در محلی به نام Stack ذخیره می‌کند.

• آدرس بازگشت: همان آدرسی است که در PC بود و حالا در Stack ذخیره شده تا بعد از تمام شدن وقفه، CPU به همان نقطه برگردد.

انواع وقفه:

1. **وقفه برنامه:** ناشی از خطای کد (مثل تقسیم بر صفر).

2. **وقفه سخت‌افزاری**: ناشی از خرابی یا رویدادهای فیزیکی مثل داغ شدن CPU

3. **وقفه ورودی/خروجی**: وقتی پرینتر کارش تمام می‌شود یا موس تکان می‌خورد.

• **مثال**: فرض کنید در حال مطالعه کتابی هستید (اجرای برنامه). تلفن زنگ می‌خورد (وقفه). شماره صفحه را یادداشت می‌کنی ذخیره در . (Stack) جواب تلفن را می‌دهی اجرای ISR بعد از اتمام تماس، از روی یادداشت به همان صفحه برمی‌گردی (آدرس بازگشت).

۶. هسته (Kernel)

قلب تپنده سیستم عامل است. بخشی از سیستم عامل که همیشه در حافظه RAM مقیم است و پایین‌ترین سطح دسترسی به سخت‌افزار را دارد.

۷. (Context Switch)

وقتی CPU از اجرای یک برنامه فرآیند A دست می‌کشد تا سراغ برنامه دیگری فرآیند B برود، باید تمام اطلاعات A را ذخیره و اطلاعات B را بارگذاری کند. به این جابجایی سنگین، Context Switch می‌گویند.

• **مثال:** مثل این است که درحال یادگیری ریاضی هستید، ناگهان باید شیمی بخوانید. باید فرمول‌های ریاضی را کنار بگذارید و مغز خود را روی شیمی تنظیم کنید. این «زمان هدر رفته» برای جابجایی، همان هزینه‌ی Context Switch است.

فصل دوم

۸. روش‌های ورودی/خروجی (I/O Methods)

1. Programmed I/O :پردازنده خودش مستقیماً تمام کارها را انجام می‌دهد و تا تمام نشدن کار سخت‌افزار، بیکار می‌ماند (بسیار کند).

2. Polling نظرسنجی : CPU مدام از سخت‌افزار می‌پرسد: «کارت تموم شد؟»، «کارت تموم شد؟».

• **مثال**: مثل اینکه مدام دم در بروی ببینی پستچی آمده یا نه. وقتت تلف می‌شود.

3. Interrupt-driven I/O : سخت‌افزار خودش بعد از اتمام کار، به CPU خبر می‌دهد.

• مثال: مثل اینکه بنشینی کارت را بکنی تا وقتی که زنگ در (وقفه) بخورد.

4. DMA (Direct Memory Access) : سخت‌افزار اجازه دارد مستقیم با RAM تبادل اطلاعات کند بدون اینکه مزاحم CPU شود. CPU فقط دستور شروع و پایان را می‌دهد.

۹. انواع فرآیندها از نظر مصرف منابع

- **CPU-Bound :** برنامه‌هایی که بیشتر با محاسبات سنگین سر و کار دارند (مثل رندر گرفتن ویدیو یا استخراج بیت‌کوین).
- **I/O-Bound :** برنامه‌هایی که بیشتر منتظر ورودی/خروجی هستند (مثل تایپ در Word یا گشت و گذار در اینترنت که منتظر دانلود داده است).

۱۰. بلوک کنترل فرآیند (PCB - Process Control Block)

هر فرآیند در سیستم عامل یک "شناسنامه" دارد که به آن PCB می‌گویند. مواردی که گفتی دقیقاً اجزای این شناسنامه هستند:

- **حالت فرآیند** :(Process State) در ادامه توضیح می‌دهم.
- **اطلاعات زمان‌بندی**: CPU چقدر به این فرآیند وقت داده؟ اولویتش چقدر است؟
- **مدیریت حافظه**: این فرآیند کجای RAM را اشغال کرده؟
- **اطلاعات حسابرسی** :(Accounting) شماره فرآیند (PID)، زمان شروع و مقدار کل استفاده از CPU.

۱۱. حالات فرآیند (Process States)

یک فرآیند در طول عمرش از این مراحل عبور می‌کند:

1. جدید (New) فرآیند در حال ایجاد شدن است.
2. پذیرش (Admitted) سیستم تصمیم می‌گیرد که آیا اجازه دارد وارد لیست فرآیندها شود یا نه (اگر RAM پر باشد، ممکن است پذیرفته نشود).
3. آماده (Ready) فرآیند در صف منتظر است تا CPU آزاد شود.

4. در حال اجرا (Running) دستورات فرآیند در حال اجرا توسط CPU است.

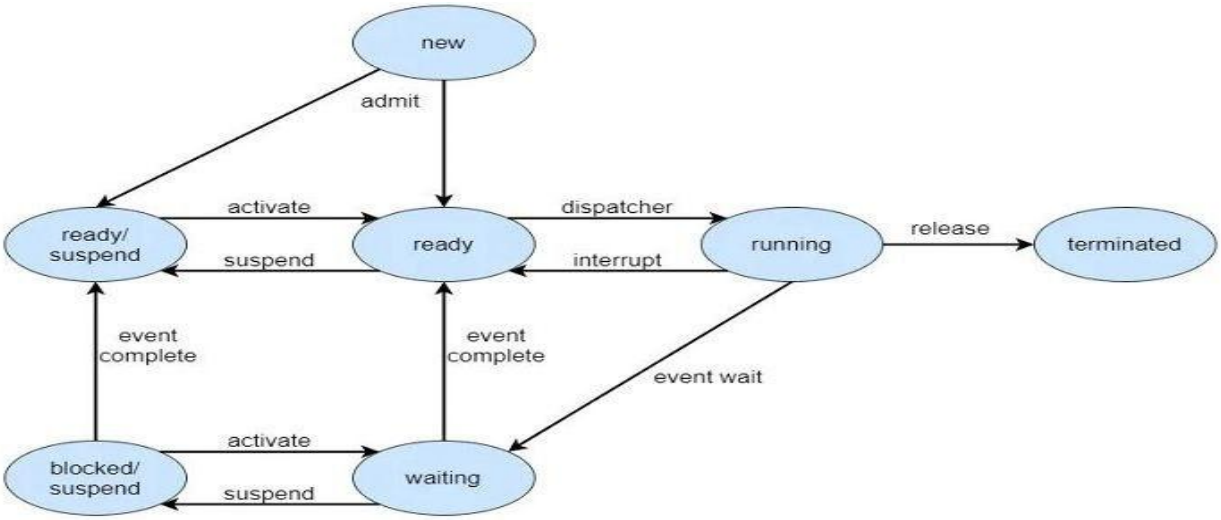
5. انتظار (Waiting/Blocked) فرآیند منتظر یک اتفاق (مثل کلیک موس یا خواندن فایل) است.

6. پایان (Terminated) کار فرآیند تمام شده و منابعش آزاد می‌شود.

این تصویر در واقع «نقشه سفر یک برنامه» در دل کامپیوتر است. فرض کن یک برنامه (مثل تلگرام) قرار است اجرا شود؛ این برنامه از لحظه تولد تا مرگ، از ایستگاه‌های مختلفی عبور می‌کند.

برای اینکه هر کسی متوجه شود، بیایید فرض کنیم سیستم عامل مثل یک «آشپزخانه بزرگ» است که فقط یک اجاق گاز (CPU) دارد و کلی سفارش غذا (برنامه‌ها).

در اینجا مرحله به مرحله این مسیر طبق عکس توضیح داده شده است:



مثال برای این تصویر دقیقاً همان چیزی است که به آن "نمودار انتقال حالت فرآیند" (Process State Transition Diagram) می‌گویند.

این نمودار قلب تپنده مدیریت فرآیند در سیستم عامل است و چرخه حیات یک برنامه را از لحظه تولد تا لحظه مرگ نشان می‌دهد.

بیایید این نمودار را به زبان بسیار ساده و مرحله‌به‌مرحله، مثل یک داستان دنبال کنیم:

داستان زندگی یک فرآیند (از تولد تا مرگ)

در این نمودار، دایره‌ها "حالت‌ها" و فلش‌ها "اتفاقاتی" هستند که باعث تغییر حالت می‌شوند.

۱. مرحله تولد: New

وقتی شما روی یک برنامه (مثل تلگرام یا یک بازی) دابل‌کلیک می‌کنید، فرآیند در حالت New قرار می‌گیرد. در این مرحله، سیستم عامل در حال تدارک مقدمات است.

۲. مرحله ورود به صف: Ready (آماده)

وقتی سیستم عامل برنامه را تایید کرد (Admit)، آن را به حافظه اصلی (RAM) می‌برد و در حالت Ready قرار می‌دهد.

• توضیح ساده: فرآیند در صف ایستاده و می‌گوید: "من حاضرم، فقط به من CPU بدهید تا کارم را شروع کنم."

۳. مرحله اجرا: Running (در حال اجرا)

وقتی نوبت به فرآیند می‌رسد، زمان‌بند سیستم عامل (Dispatcher) آن را انتخاب کرده و به CPU می‌سپارد. حالا فرآیند در حالت Running است.

• نکته: در هر لحظه روی هر هسته CPU، فقط یک فرآیند می‌تواند Running باشد.

۴. دو راهی حساس: چه اتفاقی حین اجرا می‌افتد؟

وقتی فرآیندی در حال اجراست(Running)، ممکن است ۳ اتفاق برایش بیفتد:

• **الف) وقفه** : (Interrupt) زمان اختصاص داده شده به فرآیند تمام می‌شود. سیستم عامل به او می‌گوید: "وقتت تمام شد، برگرد ته صف!" و فرآیند دوباره به حالت Ready برمی‌گردد.

• **ب) انتظار برای یک رویداد** : (Waiting) فرآیند به چیزی نیاز دارد که آماده نیست (مثلاً منتظر است کاربر دکمه‌ای را بزند یا فایلی از هارد خوانده شود). در این صورت به حالت Waiting (انتظار) می‌رود.

• **ج) پایان کار** : (Release) کار برنامه تمام می‌شود و به حالت Terminated (پایان) می‌رود.

مفهوم "تعلیق" - (Suspend) بخش مهم و امتحانی تصویر

دو دایره در سمت چپ وجود دارد که کلمه Suspend دارند. این یعنی سیستم عامل به دلیل شلوغیِ بیش از حد حافظه (RAM)، فرآیند را موقتاً به هارد دیسک منتقل کرده است تا جا باز شود.

۵. حالت Ready/Suspend (آماده‌ی معلق)

فرآیند آماده اجراست، اما چون RAM پر بوده، سیستم عامل آن را به هارد منتقل کرده است.

• **Activate:** وقتی RAM خلوت شود، فرآیند دوباره فعال شده و به صف Ready در RAM برمی‌گردد.

۶. حالت Blocked/Suspend (مسدودِ معلق)

فرآیند هم منتظر یک رویداد است (مثلاً منتظر کلیک کاربر) و هم به دلیل کمبود جا، به هارد تبعید شده است!

• Event Complete : اگر آن اتفاقی که فرآیند منتظرش بود بیفتد، فرآیند از حالت مسدود خارج شده و به Ready/Suspend می‌رود (یعنی هنوز در هارد است اما دیگر منتظر رویداد نیست، فقط منتظر جای خالی در RAM است).

Ready/Suspend و Blocked/Suspend این‌ها برای زمانی هستند که حافظه (RAM) خیلی پر شده و سیستم عامل مجبور است بعضی برنامه‌ها را موقتاً به هارد دیسک بفرستد تا جا باز شود.

• Suspend تعلیق : سیستم عامل می‌گوید: «فعلاً برو توی انباری (هارد) بخواب تا سرم خلوت شود».

• Activate فعال‌سازی : وقتی RAM خلوت شد، برنامه از انباری دوباره به حافظه اصلی برمی‌گردد.

خلاصه برای امتحان (کلید طلایی):

- **New: در حال ساخته شدن.**
- **Ready: در RAM، منتظر CPU.**
- **Running: در حال استفاده از CPU.**
- **Waiting: منتظر یک رویداد (مثل ورودی/خروجی).**
- **Terminated: تمام شده.**
- **Suspend : فرآیند به جای RAM، در هارد دیسک (Virtual Memory) نگهداری می‌شود تا RAM خالی شود.**

مثال برای درک بهتر: تصور کنید در بانک هستید.

وقتی فرم پر می‌کنید .(New)

وقتی در صف می‌ایستید .(Ready)

وقتی جلوی باجه با کارمند حرف می‌زنید (Running)

اگر کارمند بگوید برو فتوکپی بگیر و بیا، شما از باجه کنار می‌روید (Waiting)

اگر بانک خیلی شلوغ شود و نگهبان بگوید فعلاً برو توی حیاط منتظر بمان تا صدایت کنیم، شما Suspend شده‌اید.

تحلیل ذره‌بینی نمودار حالات فرآیند Process State Diagram

در این عکس، ما ۵ ایستگاه اصلی و ۲ ایستگاه فرعی داریم. هر فلش در این عکس، یک «اتفاق» است که در دنیای گانت چارت با یک عدد یا یک حرکت نمایش داده می‌شود.

مثال دیگری از تصویر بالا و مطالب کلیدی مشترک با گانت چارت ها:

۱. ایستگاه New ایجاد/تولد:

لحظه‌ای که شما دستور اجرای یک برنامه را می‌دهید.

• Admit پذیرش: این همان لحظه‌ای است که در جدول گانت چارت به آن (Arrival Timeزمان ورود) می‌گوییم. تا زمانی که فرآیند Admit نشود، اصلاً وارد محاسبات ما نمی‌شود.

۲. ایستگاه Ready صف آماده :

فرآیند در RAM است و تمام منابع را دارد، جز CPU .

• ارتباط با گانت چارت: تمام فرآیندهایی که در جدول می‌بینی (P1, P2, ...) ، در زمان ورودشان به این ایستگاه می‌رسند.

• Dispatcher اعزام‌کننده : این همان «دست غیبی» است که یک فرآیند را از صف انتخاب می‌کند و روی گانت چارت می‌کشد.

۳. ایستگاه Running در حال اجرا :

فرآیند روی CPU است.

• ارتباط با گانت چارت: این همان مستطیلی است که روی نمودار می‌کشی. طول این مستطیل برابر با (Burst Time زمان اجرا) فرآیند است.

• Interrupt وقفه: در الگوریتم‌های Preemptive غیرانحصاری، سیستم عامل فرآیند را از وسطِ کار بیرون می‌کشد و دوباره به دایره Ready برمی‌گرداند.

۴. ایستگاه Waiting انتظار /مسدود:

فرآیند زنده است اما منتظر یک اتفاق خارجی (مثل ورودی/خروجی) است.

• نکته کلیدی: در گانت چارت‌های ساده، ما معمولاً فرض می‌کنیم فرآیند به این دایره نمی‌رود. اما اگر برود، یعنی CPU بیکار می‌ماند یا به سراغ فرآیند بعدی می‌رود.

• Event Complete: وقتی انتظار تمام شود، فرآیند هرگز مستقیم به CPU برنمی‌گردد! بلکه اول باید برود در صف Ready تا دوباره نوبتش شود.

۵. ایستگاه Terminated پایان:

فرآیند کارش تمام شده.

• ارتباط با گانت چارت: عددِ انتهای مستطیل هر فرآیند، لحظه رسیدن به این ایستگاه است که به آن Exit/Completion Time می‌گوییم.

اصطلاحاتی که قبل از حل گانت چارت باید در ذهنت حک شود:

اگر کسی این عکس را خوب بفهمد، در گانت چارت با این کلمات غریبی نمی‌کند:

1. **Arrival Time زمان ورود:** لحظه عبور از فلش Admit و ورود به دایره .Ready

2. **Burst Time زمان اجرا :** کل زمانی که فرآیند نیاز دارد در دایره Running بماند تا کارش تمام شود.

3. **Waiting Time زمان انتظار:** مجموع زمان‌هایی که فرآیند در دایره Ready معطل شده تا فلش Dispatcher او را به سمت CPU ببرد.

4. **Preemption حق تقدم/بیرون انداختن :** همان فلش Interrupt که فرآیند را از دایره Running به Ready برمی‌گرداند در روش‌هایی مثل Round Robin.

5. **Context Switch تعویض متن**: عملیاتی که وقتی یک فرآیند از Running خارج و فرآیند دیگری از Ready وارد می‌شود، رخ می‌دهد (ذخیره شناسنامه قبلی و باز کردن شناسنامه جدید).

ارتباط Suspend با گانت چارت:

اگر استاد مسئله‌ای داد که در آن حافظه پر بود، فرآیند به دایره‌های Ready/Suspend یا Blocked/Suspend می‌رود. در گانت چارت، این یعنی یک وقفه طولانی‌مدت؛ چون فرآیند از RAM خارج شده و به هارد رفته است. بازگشت او فلش Activate زمان زیادی می‌برد.

بخش ترید(Thread)

۱. مفهوم نخ (Thread) یا فرآیند سبک‌وزن

فرض کنید یک آرایه بسیار بزرگ (مثلاً ۱۰ میلیون عدد) دارید.

- **حالت اول (بدون نخ):** یک فرآیند تکی شروع می‌کند از ابتدا تا انتهای آرایه را مرتب می‌کند. اگر این کار ۱۰ دقیقه طول بکشد، برنامه شما در این ۱۰ دقیقه «قفل» می‌شود (Not Responding)
- **حالت دوم:** با استفاده از نخ (Parallelism) ما آرایه را به ۵ قسمت تقسیم می‌کنیم و به ۵ «نخ» می‌سپاریم. این ۵ نخ به‌صورت موازی (Parallel) روی ۵ بخش کار می‌کنند. زمان اجرا تقریباً به یک‌پنجم کاهش می‌یابد.

تعریف کتابی: نخ کوچک‌ترین واحد اجرایی است که درون یک فرآیند قرار دارد. نخ‌ها در منابع فرآیند (مثل حافظه) با هم شریک هستند اما هر کدام مسیر اجرای خود را دارند.

مثال کاربردی: نرم‌افزار Word

وقتی در Word تایپ می‌کنید، چندین نخ همزمان کار می‌کنند:

1. **نخ اول:** حروف تایپ شده را روی صفحه نشان می‌دهد.
2. **نخ دوم:** غلط‌های املایی را زیرش خط قرمز می‌کشد.

3. نخ سوم: هر یک دقیقه یک نسخه‌ی پشتیبان (Autosave) می‌گیرد.

اگر نخ‌ها نبودند، وقتی سیستم داشت Autosave می‌گرفت، تایپ کردن شما متوقف می‌شد!

۲. رابطه نخ‌ها و فرآیندها (ترکیبات ممکن)

این رابطه را در جدول زیر خلاصه کرده ایم:

توضیح به زبان ساده	نام ساختار	حالات
مدل قدیمی مثل dos، درهرلحظه فقط یک کار انجام میشود.	Single/threaded	یک فرایند/یک نخ
مدل امروزی مثل chrome، یک برنامه چندین کار	Multi/ threaded	یک فرآیند/چند
اجرای همزمان چند برنامه مستقل مثلا بازی کردن وآهنگ گوش دادن	Multi/process	چند فرآیند/هرکدام یک نخ
سیستم های پیشرفته که چندین برنامه سنگین را با نخ های داخلی شان مدیریت میکنند.	ترکیبی(modern os)	چندفرایند/هرکدام چند نخ

۳. سطح اجرای نخ‌ها (User vs Kernel)

نخ‌ها بسته به اینکه چه کسی آن‌ها را مدیریت می‌کند به دو دسته تقسیم می‌شوند:

الف) نخ سطح کاربر :(User-Level Thread) توسط کتابخانه‌های برنامه‌نویسی مدیریت می‌شوند. سیستم عامل از وجود آن‌ها بی‌خبر است. سریع هستند اما اگر یکی از آن‌ها مسدود (Block) شود، کل فرآیند متوقف می‌شود.

ب) نخ سطح هسته : Kernel-Level Thread مستقیماً توسط سیستم عامل مدیریت می‌شوند. اگر یک نخ مسدود شود، سیستم عامل می‌تواند نخ دیگری از همان فرآیند را اجرا کند.

فصل سوم

۴. زمان‌بندی پردازنده (CPU Scheduling)

چون معمولاً تعداد فرآیندها خیلی بیشتر از تعداد هسته‌های CPU است، سیستم عامل باید مثل یک «ناظم» تصمیم بگیرد که چه کسی اول از CPU استفاده کند.

معیارهای ارزیابی زمان‌بندی (Criteria)

سیستم عامل برای اینکه بفهمد زمان‌بندی‌اش خوب است یا نه، این ۴ فاکتور را چک می‌کند:

1. **Max CPU Utilization:** یعنی تا حد امکان اجازه ندهیم CPU بیکار بماند (باید همیشه ۱۰۰٪ در حال کار باشد).
2. **Max Throughput (توان عملیاتی):** تعداد فرآیندهایی که در یک واحد زمان (مثلاً یک ثانیه) به پایان می‌رسند، حداکثر باشد.
3. **Min Turnaround Time (زمان بازگشت):**

کل زمانی که طول می‌کشد تا یک فرآیند از لحظه ورود تا لحظه اتمام کارش انجام شود، حداقل باشد.

4. Min Waiting Time (زمان انتظار):

فرآیندها باید کمترین زمان ممکن را در «صف انتظار» (Ready Queue) بمانند.

۵. انواع روش‌های زمان‌بندی

الف) انحصاری (Non-Preemptive)

در این حالت، وقتی CPU به یک فرآیند داده می‌شود، آن فرآیند تا زمانی که کارش تمام نشود یا خودش داوطلبانه CPU را رها نکند، آن را نگه می‌دارد. هیچ‌کس نمی‌تواند به زور CPU را از او بگیرد.

•مثال: مثل نانوایی سنتی که یک نفر ۱۰۰ تا نان می‌خواهد و تا نان‌هایش تمام نشود، نفر بعدی نمی‌تواند نان بگیرد.

ب) غیرانحصاری (Preemptive)

در اینجا سیستم عامل مقتدر است! اگر یک فرآیند مهم‌تر بیاید یا زمانِ فرآیند فعلی تمام شود، سیستم عامل CPU را به زور از او می‌گیرد و به دیگری می‌دهد.

•**مثال**: مثل صف عابربانک که سیستم بگوید هر نفر فقط ۲ دقیقه حق استفاده دارد؛ بعد از ۲ دقیقه نفر بعدی می‌آید، حتی اگر کار نفر قبلی تمام نشده باشد.

۶. محاسبه میانگین زمان انتظار

برای اینکه بفهمیم کدام روش بهتر است، «میانگین زمان انتظار» را حساب می‌کنیم.

•زمان انتظار هر فرآیند: زمان اتمام+زمان مورد نیاز +زمان ورود

•میانگین: مجموع زمان‌های انتظار تقسیم بر تعداد فرآیندها.

نکته: هرچه این عدد کوچک‌تر باشد، سیستم عامل در مدیریت فرآیندها موفق‌تر بوده است.

مبحث زمان‌بندی پردازنده و نمودار گانت:

الف) گانت‌چارت (Gantt Chart) چیست؟

تعریف خیلی ساده

گانت‌چارت یک خط زمان (Timeline) است که نشان می‌دهد:

- CPU
- در چه بازه‌ای از زمان
- در اختیار کدام فرایند (Process) بوده است.

یعنی «پردازنده از زمان ۰ تا آخر، مشغول انجام کار کدام فرایند بوده؟»

چرا گانت‌چارت مهم است؟

چون با استفاده از گانت‌چارت می‌توانیم:

- زمان انتظار (Waiting Time)
- زمان بازگشت (Turnaround Time)
- ترتیب اجرای فرایندها

را خیلی دقیق و قابل محاسبه به دست بیاوریم.

بدون گانت‌چارت، حل CPU Scheduling تقریباً غیرممکن است.

ب) تعریف و نقش نمودار گانت (Gantt Chart)

تعریف علمی: نمودار گانت یک مدل گرافیکی برای نمایش زمان‌بندی بصری است که در آن فعالیت‌ها بر روی یک محور افقی (محور زمان) ترسیم می‌شوند. در سیستم‌عامل، این نمودار توالی و ترتیب اجرای فرآیندها توسط واحد پردازش مرکزی ((CPUرا به تصویر می‌کشد.

نقش در سیستم‌عامل: این نمودار به مدیر سیستم یا دانشجو اجازه می‌دهد تا تخصیص منابع را بررسی کند. به کمک آن می‌توان فهمید در هر لحظه دقیقاً کدام فرآیند در حال اجرا بوده، چه

زمانی کارش تمام شده و هر فرآیند چه مدت در صف انتظار معطل مانده است. این ابزار، بهترین راه برای محاسبه کارایی الگوریتم‌های مختلف است.

پ) معرفی و تشریح الگوریتم‌های زمان‌بندی

در اینجا ۶ روش اصلی برای نوبت‌دهی به فرآیندها آورده شده است. هر روش ابتدا از نظر علمی تعریف شده و سپس با یک مثال ساده توضیح داده شده است:

اول: روش FCFS یا (First-Come, First-Served)

«اولین ورودی، اولین سرویس»

• مثال ساده: دقیقاً مثل صف نانوایی؛ فرقی نمی‌کند کسی ۱ نان می‌خواهد یا ۱۰۰ نان، هر کس زودتر آمده باشد اول نان می‌گیرد.

دوم: روش LCFS یا (Last-Come, First-Served)

«آخرین ورودی، اولین سرویس»

• مثال ساده: مثل یک کمد لباس باریک؛ آخرین لباسی که روی بقیه لباس‌ها قرار می‌دهید، اولین لباسی است که موقع نیاز از کمد برمی‌دارید.

سوم: روش RR یا (Round Robin)

«نوبت‌دهی چرخشی»

• مثال ساده: مثل صندلی بازی یا مسابقه‌ای که در آن به هر نفر فقط ۲ دقیقه وقت می‌دهند تا حرف بزند و بعد نوبت را به نفر بعدی می‌دهند.

چهارم: روش SJF یا (Shortest Job First)

«اولویت با کوتاه‌ترین کار»

• مثال ساده: در صف صندوق فروشگاه، کسی که فقط یک بطری آب دارد را جلوتر از کسی که چرخ‌دستی‌اش پر است راه می‌اندازند تا صف سریع‌تر حرکت کند.

پنجم: روش SRT یا (Shortest Remaining Time)

«کوتاه‌ترین زمان باقی‌مانده»

• مثال ساده: شما در حال شستن ظرف‌ها هستید، کسی می‌آید و می‌گوید فقط یک قاشق برای شستن دارد؛ شما کار را متوقف می‌کنید، قاشق او را می‌شویید و بعد به ادامه کار خودتان برمی‌گردید.

ششم: روش MLFQ یا (Multi-Level Feedback Queue)

• تعریف علمی «صف چندسطحی بازخوردی»

• مثال ساده: مثل تفکیک مشتریان در بانک؛ یک صف برای کارهای بانکی سریع (مثل واریز) و یک صف برای کارهای زمان‌بر (مثل افتتاح حساب) که مراجعین بر اساس نوع کارشان جابجا می‌شوند.

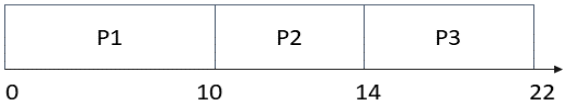
اول: روش FCFS یا (First-Come, First-Served)

• تعریف علمی: این نام مخفف عبارت «اولین ورودی، اولین سرویس» است. در این الگوریتم، اولویت با فرآیندی است که زودتر وارد صف آماده شده باشد. این روش از نوع «انحصاری» است؛ یعنی تا زمانی که کار فرآیند فعلی تمام نشود، پردازنده به فرآیند دیگری داده نمی‌شود.

• مثال ساده: دقیقاً مثل صف نانوایی؛ فرقی نمی‌کند کسی ۱ نان می‌خواهد یا ۱۰۰ نان، هر کس زودتر آمده باشد اول نان می‌گیرد.

FCFS

Process	Arrtime	Burst time
P1	0	10
P2	0	4
P3	0	8



$$\frac{(10-0-0)+(14-4-0)+(22-8-0)}{3} = \frac{24}{3} = 8$$

1. لحظه ۰: فرآیند P1 وارد CPU می‌شود. چون ۱۰ واحد زمان نیاز دارد، تا لحظه ۱۰ روی CPU می‌ماند. نقطه پایان P1 = 10

2. لحظه ۱۰: حالا نوبت P2 است. چون ۴ واحد زمان نیاز دارد، از لحظه ۱۰ شروع کرده و در لحظه ۱۴ کارش تمام می‌شود .{10 + 4 = 14} نقطه پایان P2 = 14

3. لحظه ۱۴: نوبت به P3 می‌رسد. ۸ واحد زمان نیاز دارد، پس از لحظه ۱۴ شروع کرده و در لحظه ۲۲ به کارش پایان می‌دهد .{14 + 8 = 22} نقطه پایان P3 =22

گام دوم: محاسبه زمان انتظار (Waiting Time)

Process	Arrival time	Burst time
P1	0	3
P2	1	3
P3	4	3
P4	6	2



$$\frac{(3-3-0)+(6-3-1)+(9-3-4)+(11-2-6)}{4} = \frac{7}{4} = 1.75$$

1. لحظه ۰: فرآیند P1 وارد می‌شود. چون تنهاست، بلافاصله روی CPU می‌رود. ۳ واحد زمان نیاز دارد، پس تا لحظه ۳ کارش طول می‌کشد. نقطه پایان P1 =3

2. لحظه ۳: کار P1 تمام شد. حالا به صف نگاه می‌کنیم؛ P2 در لحظه ۱ رسیده بود و منتظر است. پس نوبت اوست. ۳ واحد زمان نیاز دارد، از ۳ شروع می‌کند و در لحظه ۶ تمام می‌شود {6 = 3 + 3} نقطه پایان P2 =6

3. لحظه ۶: کار P2 تمام شد. در این لحظه، P3 که در لحظه ۴ رسیده بود و P4 که همین الان در لحظه ۶ رسیده) در صف هستند. چون P3 زودتر آمده قانون (FCFS) ، اول او اجرا می‌شود. ۳ واحد زمان نیاز دارد، پس در لحظه ۹ تمام می‌شود .{9 = 3 + 6} نقطه پایان P3 =9

4. لحظه ۹: نوبت به آخرین نفر یعنی P4 می‌رسد. ۲ واحد زمان نیاز دارد، از ۹ شروع کرده و در لحظه ۱۱ به کارش پایان می‌دهد {11 = 2 + 9} نقطه پایان P4 = 11

گام دوم: محاسبه زمان انتظار (Waiting Time)

فرمول: زمان ورود – زمان اجرا – زمان پایان = زمان انتظار

- فرآیند {0} = (0 - 3 - 3) :P1

(توضیح: چون در لحظه صفر آمد و همان لحظه هم اجرا شد).

- فرآیند {2} = (1 - 3 - 6) :P2

(توضیح: در لحظه ۱ آمد ولی تا لحظه ۳ که P1 تمام شود منتظر ماند).

- فرآیند {2} = (4 - 3 - 9) :P3

(توضیح: در لحظه ۴ آمد ولی تا لحظه ۶ که P2 تمام شود منتظر ماند).

- فرآیند {3} = (6 - 2 - 11) :P4

(توضیح: در لحظه ۶ آمد ولی تا لحظه ۹ که P3 تمام شود منتظر ماند).

گام سوم: محاسبه میانگین زمان انتظار (Waiting Time)

دوم: الگوریتم LCFS - آخرین ورودی، اولین سرویس

نام کامل: Last-Come, First-Served

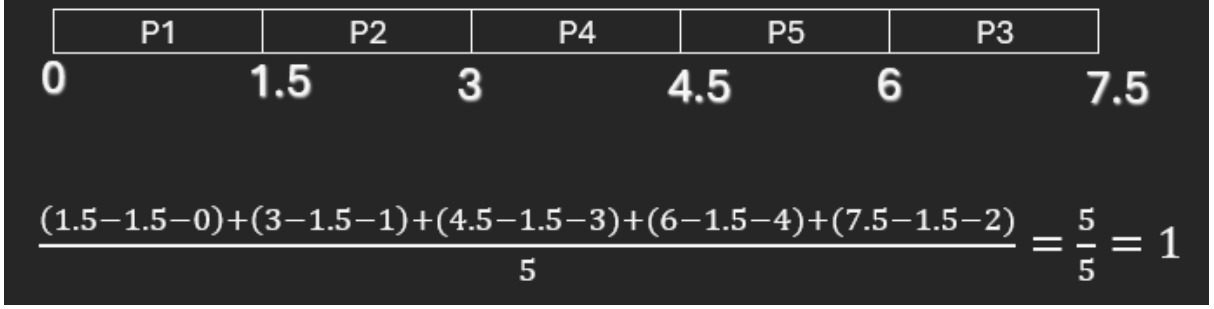
معنی: آخرین فرآیندی که وارد صف شده است، اولین فرآیندی است که پردازش می‌شود.

تعریف علمی: این روش دقیقاً برعکس FCFS عمل می‌کند. زمانی که CPU آزاد می‌شود، سیستم‌عامل به صف نگاه کرده و فرآیندی را انتخاب می‌کند که دیرتر از همه رسیده باشد. این الگوریتم مشابه ساختار داده‌ای) Stackپشته) است.

مثال عامیانه: تصور کن یک دسته بشقاب را روی هم می‌چینی؛ بشقابی که آخر از همه روی بقیه گذاشتی (آخرین ورودی)، اولین بشقابی است که برای شستن برمی‌داری (اولین سرویس).

LCFS

Process	Arrival time	Burst time
P1	0	1.5
P2	1	1.5
P3	2	1.5
P4	3	1.5
P5	4	1.5



گام اول: رسم Gantt Chart نمودار گانت

برای رسم، باید خیلی با دقت عمل کنیم چون نوبت‌ها بر اساس آخرین ورودی‌هاست:

1. لحظه ۰: فرآیند P1 وارد می‌شود. چون در این لحظه هیچ‌کس دیگری در صف نیست، کارش را شروع می‌کند. ۱.۵ واحد زمان نیاز دارد، پس در لحظه ۱.۵ تمام می‌شود.

2. لحظه ۱.۵: کار P1 تمام شد. حالا سیستم‌عامل به صف نگاه می‌کند. چه کسانی رسیده‌اند؟ P2در لحظه ۱ رسیده بود. چون فقط P2 در صف است، نوبت اوست. ۱.۵ واحد اجرا می‌شود و در لحظه ۳ تمام می‌شود.

3. لحظه ۳: حالا نکته طلایی اینجاست! کار P2 تمام شد. در این لحظه چه کسانی در صف هستند؟ P3 در لحظه ۲ رسیده بود و P4 در لحظه ۳ رسیده بود . طبق قانون LCFS آخرین ورودی ، چون P4 دیرتر از P3 آمده، اول او اجرا می‌شود! ۱.۵ واحد اجرا می‌شود و در لحظه ۴.۵ تمام می‌شود.

4. لحظه ۴.۵: کار P4 تمام شد. حالا چه کسانی در صف هستند؟ P3 که خیلی وقت است منتظر مانده و P5 که در لحظه ۴ رسیده بود . باز هم طبق قانون، چون P5 دیرتر آمده، اول او می‌رود روی CPU ۱.۵ واحد اجرا شده و در لحظه ۶ تمام می‌شود.

5. لحظه ۶: در نهایت فقط P3 در صف مانده است. نوبت به او می‌رسد، ۱.۵ واحد اجرا شده و در لحظه ۷.۵ کل کار به پایان می‌رسد.

گام دوم: محاسبه زمان انتظار (Waiting Time)

دقیقاً مثل همان روشی که روی تخته دیدی:

فرمول: زمان ورود - زمان اجرا - زمان پایان = زمان انتظار

- فرآیند P1: (1.5 - 1.5 - 0) = 0
- فرآیند P2: (3 - 1.5 - 1) = 0.5
- فرآیند P4: (4.5 - 1.5 - 3) = 0
- فرآیند P5: (6 - 1.5 - 4) = 0.5
- فرآیند P3: (7.5 - 1.5 - 2) = 4

گام سوم: محاسبه میانگین زمان انتظار

(تفاوت با FCFS)

• **تفاوت اصلی** : در روش قبلی (FCFS)، هر کس زودتر می‌آمد زودتر کارش راه می‌افتاد. اما اینجا در LCFS، فرآیند P3 چون در بدترین زمان ممکن رسید، قربانی شد و با اینکه زودتر از P4 و P5 آمده بود، آخر از همه کارش تمام شد و ۴ واحد انتظار کشید!

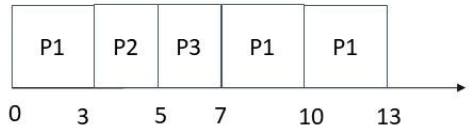
سوم: روش RR یا (Round Robin)

«نوبت‌دهی چرخشی»

• مثال ساده: مثل صندلی بازی یا مسابقه‌ای که در آن به هر نفر فقط ۲ دقیقه وقت می‌دهند تا حرف بزند و بعد نوبت را به نفر بعدی می‌دهند.

ROUND ROBIN (RR)
Quantum=3

Process	Arrival	Burst time
P1	0	9
P2	0	2
P3	0	2



$$\frac{(13-9-2)+(5-2-0)+(7-2-0)}{3} = \frac{12}{3} = 4$$

تعریف علمی: در این روش، به هر فرآیند یک زمان مشخص به نام (Quantum کوانتوم) داده می‌شود. فرآیند روی CPU می‌رود و اگر کارش تمام نشد، بعد از اتمام کوانتوم، CPU از او گرفته شده و به نفر بعدی داده می‌شود. این روش کاملاً عادلانه است.

گام اول: رسم (Gantt Chart نمودار گانت)

۱. از ۰ تا ۳: نوبت P1 است. چون کوانتوم ۳ است، با اینکه ۹ واحد کار دارد، فعلاً فقط ۳ واحد انجام می‌دهد و ۶ واحدش باقی می‌ماند.

۲. از ۳ تا ۵: نوبت P2 است. چون کل کارش ۲ واحد است (کمتر از کوانتوم ۳)، کارش کلاً تمام می‌شود. (نقطه پایان P2 =5)

۳. از ۵ تا ۷: نوبت P3 است. او هم ۲ واحد کار دارد و تمام می‌شود. (نقطه پایان P3 =7)

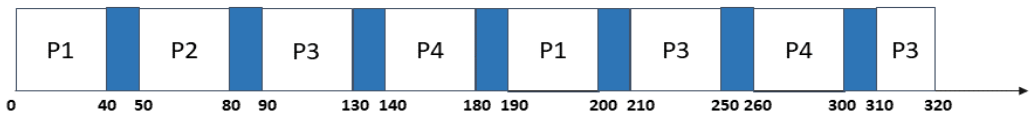
۴. از ۷ تا ۱۰: دوباره نوبت P1 است. ۳ واحد دیگر انجام می‌دهد و ۳ واحد دیگرش باقی می‌ماند.

۵. از ۱۰ تا ۱۳: باز هم نوبت P1 است (چون کسی در صف نیست) و ۳ واحد آخر را تمام می‌کند. نقطه پایان P1 = 13

ROUND ROBIN(RR)

QUANTUM=40_contex switch=1

Process	Arrival time	Burst time
P1	0	50
P2	0	30
P3	0	90
P4	0	80



$$\frac{200+80+320+300}{4} = \frac{900}{4} = 225$$

الگوریتم Round Robin_ (RR) مثال دوم با (Context Switch)

نکته علمی مهم: در دنیای واقعی، وقتی CPU می‌خواهد از یک فرآیند سراغ فرآیند بعدی برود، زمانی صرف «تعویض متن» یا Context Switch می‌شود. در این مدت CPU عملاً هیچ کار مفیدی انجام نمی‌دهد (زمان تلف شده).

کوانتوم: 40 | هزینه جابجایی (Context Switch): 10

گام اول: رسم Gantt Chart با در نظر گرفتن زمان تلف شده

۱. از ۰ تا ۴۰: اجرای P1 ۱۰ واحدش باقی می‌ماند.

۲. از ۴۰ تا ۵۰: Context Switch زمان تلف شده – شکل ابی رنگ داخل تصویر.

۳. از ۵۰ تا ۸۰: اجرای P2 چون کارش ۳۰ تا بود و از کوانتوم (۴۰) کمتر بود، همین‌جا تمام می‌شود.

۴. از ۸۰ تا ۹۰: Context Switch.

۵. از ۹۰ تا ۱۳۰: اجرای P3 ۵۰ واحدش باقی می‌ماند.

۶. از ۱۳۰ تا ۱۴۰: Context Switch.

۷. از ۱۴۰ تا ۱۸۰: اجرای P4 ۴۰ واحدش باقی می‌ماند.

۸. **از ۱۸۰ تا ۱۹۰:** Context Switch.

۹. از ۱۹۰ تا ۲۰۰: برگشت به P1 برای انجام ۱۰ واحد باقی‌مانده.کار P1 اینجا تمام می‌شود = ۲۰۰.

۱۰. ادامه به همین ترتیب برای بقیه...

گام دوم: محاسبه زمان پایان فرآیندها

تفاوت این دو مثال (برای درک بهتر)

- در مثال اول ، جابجایی بین فرآیندها آنی بود.

• در مثال دوم ، بعد از هر بار که یک نفر CPU را ترک می‌کند، یک وقفه ۱۰ واحدی داریم. این ۱۰ واحدها باعث می‌شوند کلِ زمانِ اتمامِ کارها (Completion Time) به شدت عقب بیفتد.

چهارم: روش SJF یا (Shortest Job First)

«اولویت با کوتاه‌ترین کار»

الگوریتم - SJF کوتاه‌ترین کار اول

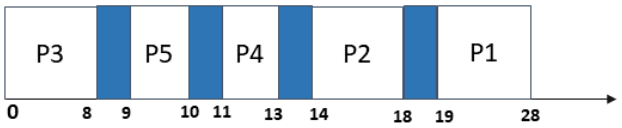
نام کامل: Shortest Job First

• **مثال ساده:** در صف صندوق فروشگاه، کسی که فقط یک بطری آب دارد را جلوتر از کسی که چرخ‌دستی‌اش پر است راه می‌اندازند تا صف سریع‌تر حرکت کند.

SJF

contex switch=1

Process	Arrival	Burst time
P1	0	9
P2	2	4
P3	0	8
P4	3	2
P5	5	1



$$\frac{(28-9-0)+(18-4-2)+(8-8-0)+(13-2-3)+(10-1-5)}{5} = \frac{43}{5} = 8.6$$

(SJFکوتاه‌ترین کار اول) یک نکته‌ی بسیار مهم دارد که همان Context Switch یا «هزینه‌ی جابجایی» است.

الگوریتم - SJF کوتاه‌ترین کار اول (با هزینه جابجایی)

منطق علمی: در این روش، CPU به صف نگاه می‌کند و بین تمام کسانی که «همین الان» در صف هستند، آن کسی را انتخاب می‌کند که زمان اجرایش از همه کمتر است. این کار باعث می‌شود میانگین انتظار به حداقل برسد.

نکته تمرین: در این مثال، یک هزینه جابجایی یا SC = 1 داریم. یعنی هر بار که CPU بخواهد از یک فرآیند به فرآیند دیگر برود، ۱ واحد زمان تلف می‌شود.

گام اول: رسم Gantt Chart نمودار گانت

۱. **لحظه ۰:** هر دو فرآیند P1 و P3 وارد شده‌اند. چون P3 با ۸ واحد کوتاه‌تر از P1 با ۹ واحد است، P3 شروع می‌کند و تا لحظه ۸ اجرا می‌شود.

۲. لحظه ۸ تا ۹: کار P3 تمام شد. حالا ۱ واحد زمان برای جابجایی SC تلف می‌شود بخش هاشورخورده.

۳. **لحظه ۹:** حالا کی در صف است؟ .P1(9), P2(4), P4(2), P5(1) کوتاه‌ترین P5 است. ۱ واحد اجرا می‌شود و در لحظه ۱۰ تمام می‌شود.

۴. **لحظه ۱۰ تا ۱۱**: دوباره ۱ واحد جابجایی.

۵. لحظه ۱۱: بین باقی‌مانده‌ها، P4 با ۲ واحد کوتاه‌تر است. اجرا می‌شود و در لحظه ۱۳ تمام می‌شود.

۶. لحظه ۱۳ تا ۱۴: دوباره ۱ واحد جابجایی.

۷. **لحظه ۱۴:** حالا بین P1 و P2، فرآیند P2 کوتاه‌تر است. ۴ واحد اجرا شده و در لحظه ۱۸ تمام می‌شود.

۸. **لحظه ۱۸ تا ۱۹**: آخرین جابجایی .

۹. **لحظه ۱۹:** در نهایت P1 با ۹ واحد اجرا می‌شود و در لحظه ۲۸ کل کار تمام می‌شود.

گام دوم: محاسبه زمان انتظار با فرمول استاد

(زمان پایان – زمان اجرا – زمان ورود – زمان‌های جابجایی مسیر = انتظار)

- فرآیند P1: (28 - 9 - 0)=19
- فرآیند P2:(18-4-2)=12

- فرآیند P3: (8 - 8 - 0) = 0
- فرآیند P4:(13-2-3)=8
- فرآیند P5: (10-1-5)=4

گام سوم: میانگین زمان انتظار.

توضیح :

در این روش، سیستم مثل یک مدیر زرنگ عمل می‌کند؛ اول کارهای ۱ دقیقه‌ای و ۲ دقیقه‌ای را راه می‌اندازد تا صف خلوت شود. اما چون به ازای هر تعویض کار، ۱ دقیقه هم وقتش تلف می‌شود، زمان پایانِ کارِ بزرگ (P1) خیلی عقب افتاد و به ۲۸ رسید.

پنجم: روش SRT یا (Shortest Remaining Time)

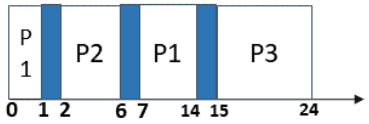
«کوتاه‌ترین زمان باقی‌مانده»

• **مثال ساده**: شما در حال شستن ظرف‌ها هستید، کسی می‌آید و می‌گوید فقط یک قاشق برای شستن دارد؛ شما کار را متوقف می‌کنید، قاشق او را می‌شویید و بعد به ادامه کار خودتان برمی‌گردید.

SRT

contex switch=1

Process	Arrival	Burst time
P1	0	8
P2	1	4
P3	2	9



$$\frac{(6-4-1)+(14-8-0)+(24-9-2)}{3} = \frac{21}{3} = 7$$

گام اول: رسم گانت چارت (تحلیل لحظه به لحظه)

1. **لحظه ۰:** فقط P1 آمده. پس شروع می‌کند به کار.

2. **لحظه ۱:** فرآیند P2 از راه می‌رسد. حالا استاد می‌گوید: «نگه دار! ببینیم کی کوتاه‌تره؟»

• P1 تا الان ۱ واحد کار کرده، پس ۷ واحدش مانده.

• P2 کلاً ۴ واحد کار دارد.

• چون ۴ < ۷ است، پس P1 باید برود بیرون و P2 بیاید.

3. **لحظه ۱ تا ۲:** چون جابجایی داریم SC=1 ، این یک ثانیه CPU خالی می‌ماند.

4. لحظه ۲: فرآیند P2 شروع می‌شود. (در همین لحظه P3 هم می‌آید ولی چون زمانش ۹ است و خیلی طولانی است، همان‌جا در صف می‌ماند).

5. لحظه ۶: کار P2 تمام می‌شود 6 = 4 + 2

6. لحظه ۶ تا ۷: دوباره جابجایی SC=1 داریم. CPU خالی می‌ماند.

7. لحظه ۷: حالا بین P1 که ۷ واحدش مانده بود و P3 که ۹ واحد دارد ، کی کمتره؟
معلومه P1 پس P1 دوباره می‌آید روی CPU و تا لحظه ۱۴ کارش را تمام می‌کند {14 = 7 + 7}

8. لحظه ۱۴ تا ۱۵: باز هم جابجایی SC=1

9. لحظه ۱۵: در نهایت نوبت به P3 سنگین می‌رسد. ۹ واحد کار دارد، پس در لحظه ۲۴ کار تمام است {24 = 9 + 15}

گام دوم: محاسبه انتظار (متد پایان – اجرا – ورود)

• انتظار P2: (6 - 4 - 1) = 1

(توضیح: در لحظه ۶ تمام شد، ۴ واحد کار کرد، در لحظه ۱ هم آمده بود. آن ۱ واحد معطلی بخاطر جابجایی اول بود).

• انتظار P1: (14 - 8 - 0) + 1 = 7

(توضیح: ۸ واحد کار کرد، در لحظه ۰ آمد، کلاً تا لحظه ۱۴ طول کشید. اما ۱ واحد هم وسطش بخاطر جابجایی معطل شد).

• انتظار P3: (24 - 9 - 2) = 13

(توضیح: در لحظه ۲ آمد، اما تا لحظه ۱۵ که نوبتش شد، ۱۳ واحد در صف Ready سماق مکید!).

گام سوم: میانگین

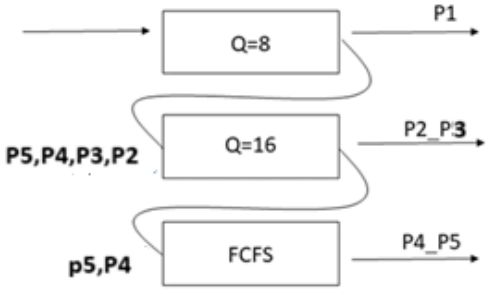
ششم: روش MLFQ یا (Multi-Level Feedback Queue)

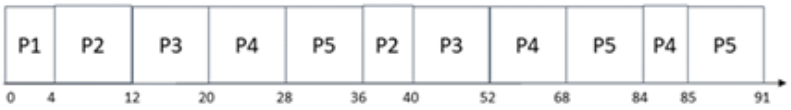
• تعریف علمی «صف چندسطحی بازخوردی»

• مثال ساده: مثل تفکیک مشتریان در بانک؛ یک صف برای کارهای بانکی سریع (مثل واریز) و یک صف برای کارهای زمان‌بر (مثل افتتاح حساب) که مراجعین بر اساس نوع کارشان جابجا می‌شوند.

MLFQ

Process	Arrival time	Burst time
P1		4
P2		12
P3		20
P4		25
P5		30





$$\frac{(4-4)+(40-12)+(52-20)+(85-25)+(91-30)}{5} = \frac{181}{5} = 36.2$$

زمان‌بندی MLFQ و اولویت‌بندی HRRN

در این مدل، سیستم‌عامل فرآیندها را در چندین صف با قوانین مختلف می‌چرخاند. در تصویر سه صف (سه جعبه وسط تصویر) درج شده است:

1. **صف اول (بالا):** با کوانتوم ۸ {q=8}
2. **صف دوم (وسط):** با کوانتوم ۱۶ {q=16}
3. **صف سوم (پایین):** با قانون FCFS هر کس زودتر بیاید.

گام اول: منطق حرکت فرآیندها (فلش‌های درج شده در تصویر)

• هر فرآیند ابتدا وارد صف اول می‌شود. اگر در ۸ واحد زمان تمام نشود، تنبیه شده و به صف دوم سقوط می‌کند.

• اگر در صف دوم هم در ۱۶ واحد زمان تمام نشود، به صف سوم (ته صف) فرستاده می‌شود.

گام دوم: بررسی فرآیندها طبق جدول استاد

استاد زمان اجرای ۵ فرآیند را نوشته که در طول مسیر تغییر می‌کنند (اعداد خط‌خورده):

• P1 ۴ واحد کار دارد. (در همان صف اول تمام می‌شود).

• P2 ۱۲ واحد کار دارد. (۸ واحد در صف اول انجام می‌دهد، ۴ واحد باقی‌مانده را در صف دوم تمام می‌کند).

• P3 ۲۰ واحد کار دارد. (۸ واحد در صف اول، ۱۲ واحد در صف دوم).

• P4 ۲۵ واحد کار دارد. (۸ در صف اول، ۱۶ در صف دوم، ۱ واحد باقی‌مانده به صف سوم می‌رود).

• P5 ۳۰ واحد کار دارد. (۸ در صف اول، ۱۶ در صف دوم، ۶ واحد باقی‌مانده به صف سوم می‌رود).

گام سوم: رسم گانت چارت (بخش پایین تخته)

۱. از ۰ تا ۴: فرآیند P1 اجرا و تمام می‌شود. (نقطه پایان = P1۴).

۲. از ۴ تا ۱۲: فرآیند P2 وارد می‌شود و ۸ واحد (کوانتوم صف اول) را کار می‌کند. چون هنوز ۴ واحدش مانده، فعلاً کنار می‌رود.

۳. از ۲۸ تا ۳۶: (بعد از چرخش‌های مختلف) نوبت به P5 می‌رسد.

۴. از ۳۶ تا ۴۰: نوبت به P2 می‌رسد تا آن ۴ واحد باقی‌مانده‌اش را در صف دوم تمام کند. (نقطه پایان = P2۴۰).

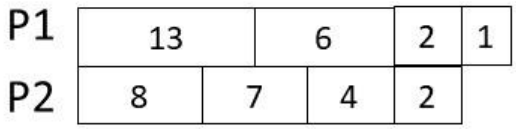
۵. از ۴۰ تا ۵۲: نوبت به P3 می‌رسد تا ۱۲ واحد باقی‌مانده‌اش را انجام دهد. (نقطه پایان P3 = ۵۲).

۶. لحظه ۶۸ به بعد: همان‌طور که در انتهای گانت چارت می‌بینی، فرآیندهای سنگین مثل P5 و P4 که به صف آخر FCFS سقوط کرده‌اند، در حال اجرا هستند. نقطه پایان P5=85 _ P4=91.

گام چهارم: محاسبه زمان انتظار (فرمول نهایی بالای تخته)

زمان بندی در سیستم چند پردازنده ای

{8,4,1,7,2,13,2,6}
{13,8,7,6,4,2,2,1}



این بخش مربوط به الگوریتم‌های زمان‌بندی Scheduling و نحوه پر کردن گانت‌چارت بر اساس اولویتِ زمان‌های پردازش است. بیا این منطق را به زبان خیلی ساده و مرحله‌به‌مرحله بازسازی کنیم تا در ذهنت تثبیت شود:

فرمول طلایی تقسیم کار (بر اساس عکس)

فرض کن یک مشت "کار" (اعداد داخل آکولاد) داری و می‌خواهی آن‌ها را بین دو کارگر {P_1} و {P_2} تقسیم کنی. هدف این است که کارها را طوری پخش کنی که هیچ‌کدام بیکار نمانند و تعادل برقرار شود.

1. مرتب‌سازی: ابتدا اعداد را از بزرگ به کوچک مرتب می‌کنیم: {13, 8, 7, 6, 4, 2, 2, 1}

2. شروع بازی: اولین عدد (بزرگترین کار) را همیشه به {P_1} می‌دهیم. (عدد ۱۳ در ردیف {P_1} قرار می‌گیرد).

3. نوبت‌دهی هوشمند (مقایسه مجموع):

• عدد دوم (۸) را به {P_2} می‌دهیم.

• حالا نگاه می‌کنیم: مجموع {P_1} شده ۱۳ و مجموع {P_2} شده ۸.

• کار بعدی همیشه به کسی می‌رسد که تا این لحظه کار کمتری انجام داده است تا عقب‌ماندگی‌اش جبران شود.

4. ادامه مسیر:

• چون ۸ کوچکتر از ۱۳ است، عدد بعدی (۷) به {P_2} می‌رسد. حالا مجموع {P_2} می‌شود ۱۵.

• حالا چون ۱۳ کوچکتر از ۱۵ است، نوبت {P_1} است که کار بعدی (۶) را بردارد. مجموع {P_1} می‌شود ۱۹.

• حالا چون ۱۵ کوچکتر از ۱۹ است، نوبت {P_2} است که عدد (۴) را بردارد. مجموع {P_2} می‌شود ۱۹.

• چون هر دو مساوی شدند (۱۹)، عدد بعدی (۲) به {P_1} می‌رسد و همین‌طور تا آخر ادامه می‌دهیم.

چرا این کار را می‌کنیم؟ (به زبان ساده)

این روش دقیقاً مثل صف نانوایی با دو شاطر است. هدف این است که هر دو شاطر با هم نان بپزند و یکی از آن‌ها خسته نشود در حالی که دیگری بیکار نشسته است. با این روش، در انتهای کار، مجموع زمان‌های {P_1} و {P_2} تقریباً با هم برابر می‌شود (در عکس هر دو حوالی ۲۱ و ۲۲ تمام شده‌اند).

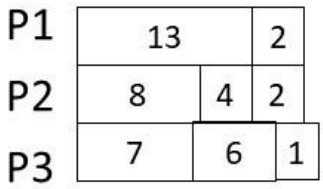
نکات کلیدی برای نوشتن در جزوه:

• **اولویت:** همیشه عدد بزرگتر اول پردازش می‌شود تا کارهای سنگین زودتر از گلوگاه خارج شوند.

• **تعادل بار** **Load Balancing**: هر وقت دیدی یک پردازنده مجموع اعدادش کمتر است، عدد بعدی را به همان بده.

{8,4,1,7,2,13,2,6}

{13,8,7,6,4,2,2,1}



بازی تقسیم کار بین ۳ نفر {P_1, P_2, P_3}

توی عکس دوم، لیست اعداد ثابت است {13, 8, 7, 6, 4, 2, 2, 1} و این بار جای ۲ نفر، بین ۳ نفر تقسیم شده است. قانون ثابت: «نوبت همیشه با کسیه که کارش از همه سبک‌تره!»

1. **غول مرحله آخر:** اول از همه عدد ۱۳ داده شده به {P_1}. فعلاً {P_1} سنگین‌ترین کار را دارد.

2. **نوبت ضعیف‌ترها:** حالا {P_2} و {P_3} بیکارن. عدد ۸ داده شده است به {P_2} و عدد ۷ داده شده است به {P_3}

3. **کی از همه کمتر کار کرده؟**

•تا اینجا: {P1 =13}، {P2 =8}، {P3 =7}.

•پس عدد بعدی یعنی ۶ میره برای {P_3} چون از همه کمتر کار کرده بود. حالا {P_3} جمعاً شد ۱۳ واحد کار.

.4دوباره مقایسه کن:

•حالا داریم: {P_1=13}، {P_2=8}، .{P_3=13}

•کی الان کمه؟ {P_2} که هنوز روی ۸ مونده! پس عدد بعدی یعنی ۴ رو میدیم به .{P_2} حالا {P_2} شد ۱۲ واحد.

5. دور آخر:

•الان وضعیت اینه: ={P_1=13}، {P_2=12}، .{P_3=13}

•عدد بعدی ۲ هست. کی کمتره؟ {P_2}پس ۲ رو میدیم به اون که بشه ۱۴.

•حالا {P_1} و {P_3} که ۱۳ هستن کمترن. عدد ۲ بعدی رو میدیم به {P_1}میشه ۱۵ و عدد ۱ آخری رو میدیم به {P_3}میشه.

نتیجه اخلاقی (چی بنویسیم؟)

توی عکس دوم اگه دقت کنی، تهِ نمودار گانت اعداد ۱۵، ۱۴ و ۱۴ رو می‌بینی. این یعنی با این روش «نوبتی دادن به اونی که کمتر کار کرده»، استاد تونسته کارها رو طوری پخش کنه که هر سه تا پردازنده تقریباً هم‌زمان (توی زمان ۱۴ و ۱۵) کارشون تموم بشه.

{8,4,1,7,2,13,2,6}

{13,8,7,6,4,2,2,1}

P1	1	2	6	13
P2	2	4	7	8

این دقیقاً ادامه همان منطق قبلی است، با این تفاوت که اینجا روش بهینه‌سازی برای کارهای کوچک‌تر را آموزش می‌دهد

روش دوم: تقسیم کار از کوچک به بزرگ (عکس جدید)

در این روش، ما می‌خواهیم برعکس قبلی عمل کنیم؛ یعنی اول "کارهای ریز" را انجام دهیم تا سریع‌تر تمام شوند.

1. مرتب‌سازی معکوس: ابتدا اعداد را از کوچک به بزرگ مرتب می‌کنیم: {1, 2, 2, 4, 6, 7, 8, 13}.

2. شروع با کوچک‌ترین: اولین عدد (۱) را به {P_1} می‌دهیم.

3. قانون مقایسه (ترازوی کار): کار بعدی همیشه به کسی می‌رسد که مجموع اعدادش (بارِ کاری‌اش) کمتر باشد.

مراحل انجام کار طبق تخته (سمت راست عکس):

- عدد ۱: به {P_1} داده می‌شود. (مجموع {P_1 = 1}
- عدد ۲: به {P_2} داده می‌شود. (مجموع {P_2 = 2}
- عدد ۲: حالا چون مجموع {P_1} از مجموع {P_2} کمتر است، این ۲ به {P_1} می‌رسد. مجموع جدید {P_1 = 3}
- عدد ۴: حالا چون مجموع {P_2} کمتر از {P_1} است، این ۴ به {P_2} می‌رسد. (مجموع جدید {P_2 = 6}
- عدد ۶: حالا مجموع {P_1} کمتر است، پس ۶ به {P_1} می‌رسد. (مجموع جدید {P_1 = 9}
- عدد ۷: حالا مجموع {P_2} کمتر است، پس ۷ به {P_2} می‌رسد. (مجموع جدید {P_2 = 13}
- عدد ۸: حالا مجموع {P_1} کمتر است، پس ۸ به {P_1} می‌رسد. (مجموع جدید {P_1 = 17}
- عدد ۱۳: در نهایت ۱۳ به {P_2} می‌رسد چون مجموعش کمتر بود.

فصل چهارم

مبحث بن بست(DEADLOCK)

بن‌بست (Deadlock) چیست؟

به زبان خیلی ساده، بن‌بست زمانی رخ می‌دهد که مجموعه‌ای از فرآیندها (برنامه‌ها) برای همیشه متوقف شوند، چون هر کدام منتظر منبعی (مثل رم یا چاپگر) هستند که در اختیار برنامه دیگری است و آن برنامه هم خودش منتظر است!

شرایط چهارگانه ایجاد بن‌بست

برای اینکه یک بن‌بست رخ دهد، باید ۴ شرط زیر همزمان برقرار باشند:

1. **انحصار متقابل (Mutual Exclusion):** منبع فقط در اختیار یک نفر باشد (مثلاً چاپگر همزمان نمی‌تواند دو متن مختلف را چاپ کند).

2. **نگهداری و انتظار (Hold & Wait):** یک برنامه منبعی را گرفته و در حالی که آن را رها نمی‌کند، منتظر منبع دیگری می‌ماند.

3. **عدم غصب (No Preemption):** سیستم‌عامل نمی‌تواند منبع را به زور از یک برنامه بگیرد؛ خود برنامه باید آن را آزاد کند.

4. **انتظار چرخشی (Circular Wait):** برنامه‌ها زنجیره‌وار منتظر هم بمانند (برنامه الف منتظر ب، ب منتظر ج، و ج منتظر الف).

مثال:

فرض کنید روی لپ‌تاپ خود دو برنامه باز کرده‌اید: نرم‌افزار ویرایش عکس (فتوشاپ) و نرم‌افزار ورد (Word)

1. فتوشاپ کارت گرافیک (GPU) را در اختیار گرفته تا یک عکس سنگین را رندر کند. 2. ورد چاپگر را رزرو کرده تا فایلی را چاپ کند. 3. حالا فتوشاپ برای خروجی نهایی نیاز دارد به چاپگر وصل شود (اما چاپگر دست ورد است).

2. همزمان ورد برای نمایش بهتر فونت‌ها نیاز به کارت گرافیک پیدا می‌کند (اما کارت گرافیک دست فتوشاپ است).

در این لحظه:

• فتوشاپ کارت گرافیک را ول نمی‌کند تا چاپگر آزاد شود.

• ورد هم چاپگر را ول نمی‌کند تا کارت گرافیک آزاد شود.

نتیجه: هر دو برنامه "هنگ" می‌کنند و هیچ‌کدام جلو نمی‌روند. این یعنی بن‌بست.

ابزار تشخیص: گراف تخصیص منابع (RAG)

در یادداشت‌های شما از دایره برای فرآیند (برنامه) و از مربع برای منبع استفاده شده است.

• اگر فلش از برنامه به سمت منبع باشد یعنی درخواست (Request)

• اگر فلش از منبع به سمت برنامه باشد یعنی منبع در اختیار آن برنامه است (Hold)

• نکته طلایی: اگر در این گراف یک "حلقه" (چرخه) ایجاد شود، یعنی ما با بن‌بست روبرو هستیم.

یک فرمول مهم برای پیشگیری

اگر مجموع درخواست‌های فرآیندها از تعداد منابع و فرآیندها کمتر باشد، بن‌بست رخ نمی‌دهد:

$$\sum_{i=1}^{n} Request(i) < m + n$$

{n} تعداد فرآیندها و {m} تعداد منابع است. به زبان ساده، اگر اشتهای برنامه‌ها برای گرفتن منابع از ظرفیت سیستم کمتر باشد، همه چیز امن است.

روشهای رفع بن بست به سه دسته تقسیم می شوند:

۱. پیشگیری از بن‌بست (Deadlock Prevention)

در این روش، ما سیستم را طوری طراحی می‌کنیم که اصلاً بن‌بست نتواند به وجود بیاید. برای این کار، باید حداقل یکی از آن ۴ شرطی که قبلاً خواندیم (نگهداری و انتظار، قبضه نکردن، انتظار چرخشی و انحصار متقابل) را از بین ببریم.

الف) حذف شرط نگهداری و انتظار (Hold and Wait)

توضیح: سیستم اجازه نمی‌دهد فرآیند یک منبع را نگه دارد و همزمان منتظر یکی دیگر بماند.

• روش برخورد: فرآیند باید در همان اول کار، تمام منابعی که تا آخر عمرش لازم دارد را اعلام کند. سیستم یا همه را یکجا به او می‌دهد، یا اگر حتی یکی کم بود، فرآیند را مسدود (Block) می‌کند و هیچ‌کدام را به او نمی‌دهد.

• **مثال چهارراه:** تصور کن به یک چهارراه می‌رسی. پلیس (سیستم‌عامل) به تو می‌گوید: "تا زمانی که کل مسیر تا انتهای خیابان بعدی برایت خالی نباشد، حق نداری حتی وارد چهارراه شوی". تو پشت چراغ قرمز می‌مانی (مسدود می‌شوی) تا کل مسیر یکجا برایت آزاد شود. این‌طوری هیچ‌وقت وسط چهارراه گیر نمی‌کنی که راه بقیه را ببندی.

ب) حذف شرط عدم قبضه (No Preemption)

توضیح: در حالت عادی منابع را نمی‌شود به زور از فرآیند گرفت. اما در پیشگیری، ما این قانون را می‌شکنیم.

• روش برخورد: اگر فرآیندی منبعی در دست دارد و درخواست منبع جدیدی بدهد که فعلاً موجود نیست، سیستم تمام منابع قبلی‌اش را به زور از او می‌گیرد (Preempt می‌کند). فرآیند باید دوباره از نو درخواست بدهد.

• مثال: فرض کن تو استاد هستی و یک کلاس در اختیار داری و درخواست یک کلاس دیگر هم داری مسئول آموزش کلاس جدید را در اختیار شما قرار نمیدهد و به شما میگوید اگر کلاس دیگری میخواهید کلاس قبلی خود را کاملا تخلیه کنید.

• مثال2

دو تا ماژیک در دست داری. حالا به مسئول آموزش می‌گویی "تخته‌پاک‌کن هم می‌خواهم". مسئول می‌گوید: "نمی‌شود! اول باید آن دو تا ماژیک را پس بدهی، بعد دوباره بیایی از اول درخواستِ دو تا ماژیک و یک تخته‌پاک‌کن را با هم بدهی".

۲. حذف شرط انتظار چرخشی (Circular Wait)

این دقیقاً همان بخشی است " بعدیش را بخواهد می‌دهد، قبلی را بخواهد نمی‌دهد".

• Linear Ordering (ترتیب خطی منابع):

• چیکار می‌کند؟ به هر منبع یک عدد (اولویت) می‌دهیم. فرآیندها فقط اجازه دارند منابعی را درخواست کنند که عددشان بیشتر از منابعی باشد که فعلاً در دست دارند.

• توضیح ساده: اگر منبع شماره ۵ دست توست، فقط می‌توانی شماره ۶ به بالا را بخواهی. اگر شماره ۳ را بخواهی، سیستم اجازه نمی‌دهد. این کار باعث می‌شود "حلقه" یا چرخه هیچ‌وقت درست نشود، چون مسیر درخواست‌ها همیشه رو به جلو (خطی) است.

• مثال: فرض کن به چاپگر شماره ۱ و به اسکنر شماره ۲ را داده‌ایم. اگر برنامه‌ای اول چاپگر را بگیرد، می‌تواند درخواست اسکنر بدهد. اما اگر برنامه‌ای اسکنر (شماره ۲) را گرفته باشد، دیگر حق ندارد درخواست چاپگر (شماره ۱) بدهد. این‌طوری هیچ‌وقت دو برنامه در انتظار هم نمی‌مانند.

۳. اجتناب از بن‌بست (Deadlock Avoidance)

الگوریتم بانکداران (Banker's Algorithm)

الگوریتم بانکداران ابزار اصلی سیستم‌عامل برای پیاده‌سازی مفهوم «اجتناب از بن‌بست» است. در این روش، سیستم پیش از تخصیص هر منبع، ابتدا محاسباتی انجام می‌دهد تا مطمئن شود با این تخصیص، راهی برای اتمام تمام فرآیندها باقی می‌ماند یا خیر.

۱. مفاهیم و پیش‌نیازهای محاسباتی

برای درک وضعیت سیستم، به سه جدول (ماتریس) نیاز داریم که بر اساس موجودی کل منابع {R_1=9, R_2=3, R_3=6} تکمیل می‌شوند:

• جدول ادعا : **(Claim)** سقف نهایی منابعی که هر فرآیند ممکن است در طول اجرای خود نیاز داشته باشد.

• جدول تخصیص :**(Allocation)** مقدار منابعی که در همین لحظه به هر فرآیند داده شده است.

• جدول نیاز :**(Need)** مقدار منابعی که هر فرآیند «هنوز» لازم دارد تا به پایان برسد. این جدول با تفریق ماتریس تخصیص از ماتریس ادعا به دست می‌آید **{Need = Claim - Allocation}** .

TOTAL RESOURCES {R1=9, R2=3, R3=6}
این جدول سقف نیاز هرفرآیند را نشان میدهد

(جدول نیاز) NEED MATRIX
این جدول حاصل تفریق دو جدول قبلی است
CLAIM- ALLOCATION= NEED

فرایند	R1	R2	R3	محاسبه ذهنی
P1	2	2	2	3-1,2-0,2-0
P2	0	0	1	6-6,1-1,3-2
P3	1	0	3	3-2,1-1,4-1
P4	4	2	0	4-0,2-0,2-2

(جدول تخصیص) ALLOCATION MATRIX
این جدول منابعی را نشان میدهد که هم اکنون در دست فرآیند هاست

فرایند	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2
جمع ستون ها	9	2	5

(جدول ادعا) CLAIM MATRIX
این جدول سقف نیاز هرفرآیند را نشان میدهد

فرایند	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

AVAILABLE= TOTAL-SUM OF ALLOCATION
AVAILABLE= {9-9, 3-2, 6-5}={0,1,1}
SAFE SEQUENCE (ترتیب امن): P2_P1_P3_P4

۲. تحلیل وضعیت انبار **(Available)**

اولین قدم برای خواندن جداول، محاسبه مقدار منابع آزاد در انبار سیستم است. ما مجموع منابعی که به همه فرآیندها داده شده جمع ستون‌های جدول **(Allocation)** را از کل منابع سیستم کم می‌کنیم:

• مجموع تخصیص یافته: **{R_1=9, R_2=2, R_3=5}**

• موجودی آزاد **{0, 1, 1} :(Available)**

۳. منطق پیدا کردن «ترتیب امن» **(Safe Sequence)**

سیستم به جدول **Need** نگاه می‌کند و آن را با موجودی آزاد **(Available)** مقایسه می‌کند. هر فرآیندی که نیازش کمتر یا مساوی موجودی انبار باشد، می‌تواند اجرا شود:

1. بررسی : **{P_2}** نیاز او **{0, 0, 1}** است که از موجودی **{1, 1, 1}**کمتر است؛ پس **{P_2}** اولین فرآیندی است که اجرا می‌شود.

2. آزاد سازی: وقتی کار{P_2} تمام شد، او تمام منابعی که قبلاً در اختیار داشت یعنی {6, 1, 2} را به انبار پس می‌دهد.

3. به‌روزرسانی: حالا انبار شارژ شده و موجودی جدید به **{6, 2, 3}** می‌رسد. اکنون سیستم می‌تواند با این منابع جدید، نیاز فرآیندهای سنگین‌تر مثل **{P_1}** یا **{P_3}** را تامین کند.

نتیجه‌گیری: اگر بتوانیم ترتیبی پیدا کنیم که همه فرآیندها یکی‌یکی به منابع برسند و تمام شوند در اینجا: {P_2 to P_1 to P_3 to P_4}، می‌گوییم سیستم در حالت امن است.

فصل پنجم

مبحث الگوریتم دکر (Dekker's Algorithm):

یکی از مهم‌ترین مباحث در بخش «انحصار متقابل» (Mutual Exclusion) است،

الگوریتم دکر اولین راه حل نرم‌افزاری کامل برای حل مشکل ناحیه بحرانی (Critical Section) است. زمانی که دو فرآیند می‌خواهند همزمان از یک منبع مشترک (مثل یک فایل یا چاپگر) استفاده کنند، این الگوریتم تضمین می‌کند که فقط یکی از آن‌ها وارد ناحیه بحرانی شود تا تداخلی پیش نیاید.

این الگوریتم در طول زمان تکامل یافت تا مشکلات راه‌حل‌های ساده‌تر قبلی را برطرف کند.

این الگوریتم شامل ۵ راه حل (یا ۵ تلاش) است که هر کدام برای رفع نقص نسخه‌ی قبلی ارائه نام راه‌حل‌های پنج‌گانه دکر:

نسخه	نام انگلیسی	نام فارسی	هدف
1	ALTERNATION	نوبت دهی قطعی (تناوب اجباری)	استفاده از یک متغیر برای نوبت دهی(نقص:سختگیرانه)
2	FLAG(BASIC)	اعلام آمادگی(پرچم)	استفاده از پرچم(FLAG) برای نشان دادن تمایل(نقص:عدم امنیت)
3	CHECKING OTHER FLAG	بررسی پرچم حریف	قبل از ورود پرچم(FLAG) دیگری را چک میکند (نقص:احتمال بن بست)
4	BACK-OFF	عقب نشینی موقت	اگر حریف آماده بود، پرچم (FLAG) خود

ترکیب نوبت (TURN) و پرچم (FLAG) برای حل قطعی مشکل	راه حل نهایی دکر	DEKKERS FINAL SOLUTION	5

توضیح منطق الگوریتم دکر

یک راه حل درست برای ناحیه بحرانی باید ۳ شرط اصلی را پاس کند:

1. Mutual Exclusion انحصار متقابل: همزمان دو نفر داخل نباشند.

2. Progress پیشرفت: اگر کسی داخل نیست و کسی می‌خواهد وارد شود، جلوی او گرفته نشود.

3. Bounded Waiting انتظار محدود: هیچ فرآیندی تا ابد پشت در منتظر نماند (گرسنگی رخ ندهد).

راه حل نهایی (نسخه ۵) به این صورت کار می‌کند:

• هر فرآیند یک پرچم (Flag) دارد که نشان می‌دهد "من می‌خواهم وارد شوم".

• یک متغیر به نام نوبت (Turn) وجود دارد که در صورت همزمانی، تعیین می‌کند حق تقدم با کیست.

• اگر هر دو همزمان بخواهند وارد شوند، فرآیندی که نوبتش نیست، پرچم خود را پایین می‌آورد تا دیگری وارد شود و پس از خروج او، نوبت به خودش برسد.

بسیار خب، در اینجا ۵ راه‌حل الگوریتم دکر برای حل مشکل ناحیه بحرانی (CriticalSection) را به ترتیب تکامل آن‌ها توضیح می‌دهم. این مبحث مستقیماً بعد از مفاهیم بن‌بست و برای حل چالش اشتراک منابع مطرح می‌شود.

بررسی گام‌به‌گام ۵ راه‌حل الگوریتم دکر

در تمام این روش‌ها، هدف این است که دو فرآیند {P_0} و {P_1} به شکلی مدیریت شوند که همزمان وارد ناحیه بحرانی نشوند.

۱. نوبت‌دهی قطعی (Alternation)

در این روش از یک متغیر مشترک به نام turn استفاده می‌شود. اگر turn == 0 باشد، فقط {P_0} اجازه ورود دارد و پس از خروج، نوبت را به ۱ تغییر می‌دهد.

• **نقص:** "سخت‌گیری" بیش از حد. اگر نوبت {P_0} باشد ولی او نخواهد وارد ناحیه بحرانی شود، {P_1} تا ابد پشت در می‌ماند؛ حتی اگر ناحیه بحرانی خالی باشد. (شرط Progress رعایت نمی‌شود).

۲. استفاده از پرچم (Flag - Basic)

در این نسخه، هر فرآیند یک پرچم اختصاصی دارد. وقتی فرآیندی می‌خواهد وارد شود، پرچم خود را True می‌کند.

• **نقص**: "امنیت پایین". ممکن است یک فرآیند پرچم خود را چک کند و ببیند دیگری نیست، اما قبل از اینکه خودش پرچم را بالا ببرد، فرآیند دیگر وارد شود. در نتیجه هر دو همزمان داخل ناحیه بحرانی می‌روند. (شرط Mutual Exclusion رعایت نمی‌شود).

۳. بررسی پرچم حریف قبل از ورود (Checking Other's Flag)

در این نسخه، فرآیند ابتدا پرچم خود را بالا می‌برد و بعد چک می‌کند که آیا حریف هم پرچمش بالاست یا خیر. اگر بالا بود، صبر می‌کند.

• **نقص**: بن‌بست (Deadlock) اگر هر دو فرآیند همزمان پرچم‌هایشان را بالا ببرند، هر دو به پرچم دیگری نگاه می‌کنند و منتظر می‌مانند تا حریف پرچمش را پایین بیاورد. نتیجه این است که هر دو تا ابد پشت در می‌مانند.

۴. عقب‌نشینی موقت (Back-off)

برای حل مشکل بن‌بست در نسخه قبلی، این قانون اضافه شد: "اگر پرچم حریف بالا بود، من پرچم خودم را پایین می‌آورم، کمی صبر می‌کنم و دوباره امتحان می‌کنم".

• **نقص:** قفل زنده .(Livelock) ممکن است دو فرآیند دقیقاً در یک زمان پرچم را بالا ببرند، هر دو ببینند دیگری هست، هر دو پایین بیاورند و دوباره دقیقاً همزمان بالا ببرند. این چرخه "تعارف کردن" ممکن است بارها تکرار شود و هیچ‌کدام وارد نشوند.

۵. راه‌حل نهایی دکر (Dekker's Final Solution)

این نسخه ترکیبی از نوبت (Turn) و پرچم (Flag) است.

• **منطق کار:** فرآیند پرچم خود را بالا می‌برد. اگر حریف هم آماده بود، به متغیر turn نگاه می‌کند. اگر نوبت او نبود، پرچمش را پایین می‌آورد تا حریف کارش را انجام دهد و پس از تمام شدن کار حریف، دوباره تلاش می‌کند.

این نسخه هر سه شرط (انحصار متقابل، پیشرفت و انتظار محدود) را کاملاً رعایت می‌کند.

• **نتیجه:**در نسخه اول مشکل اصلی عدم پیشرفت (PROGRESS)، درنسخه دوم مشکل اصلی نقص انحصار متقابل، در نسخه سوم مشکل اصلی وقوع بن بست (DEADLOCK)، در نسخه چهارم مشکل اصلی وقوع قفل زنده (LIVE LOCK) و در نسخه پنجم هیچ مشکلی وجود نداد و کاملا ایمن می باشد.

• برای متوجه شدن کامل در ادامه برای هرکدام از آن پنج تلاش یک کد فرضی و تحلیل خط به خط بیان شده است•

تلاش اول (تناوب قطعی)

در این روش از یک متغیر سراسری مشترک به نام turn استفاده شده که دو مقدار 0 یا 1 را می‌تواند بگیرد. مقدار اولیه turn برابر 0 است. بنابراین ابتدا P0 می‌تواند وارد ناحیه بحرانی شود.

در این حالت P1 در حلقه while منتظر می‌ماند تا P0 از ناحیه بحرانی خارج شده و turn را 1 کند. در این صورت P1 می‌تواند وارد ناحیه بحرانی شود. برنامه فرایندها به صورت زیر است:

P0(void)
{
while(TRUE)
{
while(turn != 0) ; /*wait*/
critical-section( );
turn = 1;
non-critical-section( );
}
}
	P1(void)
	{
	while(TRUE)
	{
	while(turn != 1) ; /*wait*/
	critical-section( );
	turn = 0;
	non-critical-section( );
	}
	}

۱- انحصار متقابل

راه حل گفته شده انحصار متقابل را تضمین می‌کند و امکان ندارد که P0 و P1 با هم وارد ناحیه بحرانی شوند. چون P0 در صورتی وارد ناحیه بحرانی می‌شود که مقدار turn برابر 0 باشد و P1 با مقدار 1 وارد می‌شود. بنابراین امکان ندارد که هر دو به turn نگاه کنند و هر دو با هم مقدارهای 0 و 1 را ببینند.

۲- پیشرفت

این روش شرط پیشرفت را رعایت نمی‌کند. چون اگر P1 وارد ناحیه بحرانی شود و کارش تمام شده و به بخش غیر بحرانی برود، در این حالت turn برابر 0 می‌باشد. حال نوبت P0 است که وارد ناحیه بحرانی شود، ولی می‌خواهد به مدت طولانی در ناحیه غیر بحرانی بماند. P1 به سرعت کارش در ناحیه غیر بحرانی تمام شده و قصد ورود مجدد به ناحیه بحرانی را دارد. اما چون turn برابر 0 است در حلقه انتظار می‌ماند تا بالاخره P0 وارد ناحیه بحرانی شده و بعد از خروج، turn را 1 کرده تا P1 بتواند وارد ناحیه بحرانی شود. در این سناریو، P1 توسط فرایندی منتظر مانده بود که در ناحیه بحرانی نبود و جلوی پیشرفتش را گرفته بود.

۳- انتظار محدود

در این روش قحطی نداریم، چون فرایندها به صورت یک در میان و نوبتی وارد ناحیه بحرانی می‌شوند. همچنین بن‌بست نیز نداریم. بنابراین شرط انتظار محدود رعایت می‌شود.

• یکی از معایب این روش این است که سرعت عملیات توسط فرایند کندتر تعیین می‌شود، چون فرایندها برای دسترسی به ناحیه بحرانی باید به صورت یک در میان عمل کنند.

• یکی از معایب این روش این است که اگر فرایندی در ناحیه بحرانی از کار بیفتد، فرایند دیگر تا ابد منتظر خواهد ماند.

تلاش دوم

در این روش از دو متغیر پرچم مشترک به نام‌های flag[0] و flag[1] با مقدار اولیه FALSE استفاده می‌شود که هر کدام متعلق به یک فرایند است. هر فرایندی که قصد ورود به ناحیه بحرانی خود را دارد، پرچم خود را TRUE می‌کند.

در این روش هر فرایند دارای کلید مجزا برای ورود به ناحیه بحرانی است تا اگر فرایندی قصد استفاده از ناحیه بحرانی را نداشت، فرایند دیگر بتواند به ناحیه بحرانی خود دسترسی داشته باشد. برنامه فرایندها به صورت زیر است:

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

P0(void)
{
while(TRUE) {
while(flag[1]);
flag[0] = TRUE;
critical-section( );
flag[0] = FALSE;
non-critical-section ( );
}
}
	P1(void)
	{
	while(TRUE) {
	while(flag[0]);
	flag[1] = TRUE;
	critical-section( );
	flag[1] = FALSE;
	non-critical-section ( );
	}
	}

بررسی شرط‌ها در تلاش دوم

۱- انحصار متقابل

این روش انحصار متقابل رعایت نمی‌شود. یعنی حتی از روش اول هم بدتر است.

سناریو: فرض کنید که P0، flag[1] را خوانده و آن را FALSE می‌بیند، اما قبل از TRUE کردن flag[0]، P1 اجرا شود و flag[0] را خوانده و آن را FALSE می‌بیند و برای ورود به ناحیه بحرانی flag[1] را TRUE کرده و وارد می‌شود. حال در این زمان P0 سوئیچ شده و flag[0] را TRUE کرده و این فرایند هم وارد ناحیه بحرانی می‌شود! بنابراین چون هر دو فرایند در یک زمان وارد ناحیه بحرانی خود شده‌اند، شرط انحصار متقابل برقرار نمی‌باشد.

۲- پیشرفت

این روش شرط پیشرفت را رعایت می‌کند. اگر P0 در ناحیه غیر بحرانی خود باشد، flag[0] را FALSE نگه می‌دارد تا P1 بتواند وارد ناحیه بحرانی خودش شود. اگر P0 برای مدت طولانی تصمیم به ورود به ناحیه بحرانی را نداشته باشد، P1 به دفعات می‌تواند وارد ناحیه بحرانی شود و سپس خارج شود. یعنی P0 جلوی پیشرفت P1 را نمی‌گیرد.

۳- انتظار محدود

این روش شرط انتظار محدود را رعایت نمی‌کند، چون امکان قحطی دارد. در این تلاش امکان ورود پی در پی یک فرایند به ناحیه بحرانی و عدم دستیابی فرایند دیگر به ناحیه بحرانی وجود دارد.

سناریو: فرض کنید P0 در ناحیه بحرانی است و به P1 سوئیچ می‌شود. چون flag[0] برابر TRUE است، P1 نمی‌تواند وارد ناحیه بحرانی شود. بعد از پایان کوانتوم، پردازنده به P0 داده شده و این فرایند سریعاً ناحیه بحرانی‌اش را اجرا کرده و سعی به ورود مجدد به ناحیه بحرانی را دارد و این اجازه به او داده می‌شود. P0 مجدداً flag[0] را TRUE کرده و اگر به P1 سوئیچ شود، باز هم نمی‌تواند اجازه ورود بگیرد. بنابراین تلاش برای دستیابی به ناحیه بحرانی تصادفی است و امکان قحطی وجود دارد.

• یکی از معایب این روش این است که اگر فرایندی در ناحیه بحرانی از کار بیفتد، فرایند دیگر تا ابد منتظر خواهد ماند.

تلاش سوم

در تلاش دوم هر فرایند ابتدا وضعیت فرایند مقابل را چک کرده و سپس flag خود را TRUE می‌کرد. بنابراین اگر هر دو به طور همزمان قصد ورود به ناحیه بحرانی را داشته باشند، flag یکدیگر را FALSE می‌بینند و با هم وارد می‌شوند. برای حل این مشکل دو سطر مسئله را عوض می‌کنیم.

boolean flag[2] = {FALSE, FALSE};

P0(void)
{
while(TRUE) {
flag[0] = TRUE ;
while(flag[1]);
critical-section( );
flag[0] = FALSE;
non-critical-section( );
}
}
	P1(void)
	{
	while(TRUE) {
	flag[1] = TRUE;
	while(flag[0]);
	critical-section( );
	flag[1] = FALSE;
	non-critical-section( );
	}
	}

بررسی شرط‌ها در تلاش سوم:

۱- انحصار متقابل

اگر یکی از فرایندها بتواند وارد ناحیه بحرانی شود، چون قبل از بررسی flag مقابل، flag خود را TRUE کرده، جلوی ورود فرایند مقابل را می‌گیرد. بنابراین شرط انحصار متقابل برقرار است.

صفحه پنجم (تصویر ۱۰۸): ادامه تلاش سوم

۲- پیشرفت

این روش شرط پیشرفت را رعایت می‌کند. (با همان استدلال تلاش دوم)

۳- انتظار محدود

در این روش شرط انتظار محدود رعایت نمی‌شود، چون امکان بن‌بست وجود دارد.

سناریو: فرض کنید که P0، flag[0] را TRUE کند ولی قبل از بررسی flag[1] در برنامه P0، به P1 سوئیچ شود و P1، flag[1] را TRUE کند. در این صورت هر دو فرایند تا ابد در حلقه انتظار گرفتار می‌شوند و بن‌بست رخ می‌دهد.

در این روش شرط انتظار محدود رعایت نمی‌شود، چون ممکن است یک فرایند به مدت نامعلوم و بدون حد بالای مشخص، گرفتار قسمت تاخیر ((delayشود و فرایند مقابل به دفعات وارد ناحیه بحرانی شود. بنابراین به دلیل امکان گرسنگی، شرط انتظار محدود رعایت نمی‌شود. همچنین به علت امکان وجود Livelock، نیز شرط انتظار محدود رعایت نمی‌شود.

تلاش چهارم (ادب و تعارف)

در تلاش قبلی هر فرایند می‌توانست روی حق خود برای ورود به بخش بحرانی‌اش پافشاری کند. در این روش هر فرایند متغیر flag خود را TRUE کرده تا خواست خود برای ورود به بخش بحرانی را نشان دهد، اما آماده است flag را تغییر دهد تا به فرایند دیگر احترام گذارد. یعنی فرایندی که قصد ورود به ناحیه بحرانی را دارد، اگر ببیند که فرایند مقابل هم می‌خواهد به ناحیه بحرانی وارد شود، flag خود را برای مدت کوتاهی FALSE کرده تا فرایند مقابل بتواند وارد شود. برنامه فرایندها به صورت زیر است:

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

```
P0(void)
{
while(TRUE)
{
flag[0] = TRUE;
while( flag[1] )
{
flag[0] = FALSE;
delay_for_a_short_time();
flag[0] = TRUE;
}
critical-section( );
flag[0] = FALSE;
non-critical-section( );
}
}
```

```
P1(void)
{
while(TRUE)
{
flag[1] = TRUE;
while( flag[0] )
{
flag[1] = FALSE;
delay_for_a_short_time();
flag[1] = TRUE;
}
critical-section( );
flag[1] = FALSE;
non-critical-section( );
}
}
```

بررسی شرط‌ها:

۱- انحصار متقابل:

راه حل گفته شده انحصار متقابل را تضمین می‌کند. (با استدلال تلاش سوم)

۲- پیشرفت:

این روش شرط پیشرفت را رعایت می‌کند. (با همان استدلال تلاش دوم و سوم)

مشکل Livelock

در تلاش چهارم مشکل بن‌بست وجود ندارد، اما مشکل جدیدی به نام Livelock وجود دارد. با دنبال کردن اجرای زیر، این مشکل را توضیح می‌دهیم:

۱) P0، flag[0] را TRUE کند.

۲) P1، flag[1] را TRUE کند.

۳) P0، flag[1] را بررسی کند.

۴) P1، flag[0] را بررسی کند.

۵) P0، flag[0] را FALSE کند.

۶) P1، flag[1] را FALSE کند.

هر دو فرایند در یک زمان به مدت کوتاه یکسان عقب‌نشینی می‌کنند. سپس با هم برمی‌گردند و مراحل بالا را تکرار می‌کنند. اگر این دنباله به طور نامحدود تکرار شود، ممکن است هیچ کدام از فرایندها نتوانند وارد ناحیه بحرانی شوند. البته این تکرار بن‌بست نمی‌باشد، چون با تغییر در سرعت نسبی فرایندها، این چرخه شکسته می‌شود.

تلاش پنجم (راه حل صحیح)

Decker بعد از چهار تلاش ناموفق، یک راه حل درست که شرایط انحصار متقابل، انتظار محدود و پیشرفت را با هم رعایت می‌کند، ارائه داد. در این روش متغیر نوبت را با متغیرهای پرچم ترکیب کرد. این الگوریتم در زیر آورده شده است:

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

int turn = 0;
```

P0(void)	P1(void)

{
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;
non-critical-section ( );
}
}
	{
	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;
	non-critical-section( );
	}
	}

هنگامی که P0 بخواهد وارد بخش بحرانی خود شود، در flag مربوط به خود مقدار TRUE می‌گذارد. سپس flag مربوط به P1 را بررسی می‌کند که دو حالت رخ می‌دهد:

الف - flag[1] برابر TRUE باشد:

اگر turn برابر یک باشد، P0 به P1 احترام گذاشته و با FALSE کردن پرچمش منتظر می‌ماند. در این هنگام P0 کاری انجام نمی‌دهد تا turn برابر صفر شود و سپس flag خودش را TRUE می‌کند.

ب - flag[1] برابر FALSE باشد:

P0 وارد بخش بحرانی شده و بعد از خروج از بخش بحرانی، در flag خود مقدار FALSE می‌گذارد تا بخش بحرانی را آزاد کند و در turn مقدار 1 را قرار می‌دهد تا حق پافشاری را به P1 واگذار کند.

الگوریتم Peterson

چندین سال بعد، Peterson راه حل ساده و زیبایی را برای حل مسئله انحصار متقابل ارائه کرد. این الگوریتم به سادگی برای {n} فرایند نیز قابل تعمیم است.

boolean flag[2] = {FALSE, FALSE};

turn = 0;

P0(void){
{
while (TRUE){
flag[0] = TRUE;
turn = 1;
while (turn == 1 && flag[1]
);
critical-section();
flag[0] = FALSE;
non-critical-section();
}
}
 	
	P1(void){
	{
	while (TRUE){
	flag[1] = TRUE;
	turn = 0;
	while (turn == 0 && flag[0]
	);
	critical-section();
	flag[1] = FALSE;
	non-critical-section();
	}
	}

در کتاب استالینگز، مقدار turn برعکس مقداردهی و تست می‌شود که تفاوتی با روش بالا ندارد.

به طور مثال برای P0، داریم

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

نحوه کار کردن این الگوریتم:

فرض کنید P0 وP1 به طور تقریبا همزمان(P1 کمی دیرتر)، قصد ورود به ناحیه بحرانی را دارند. هر دو فرایند، شماره خود را در turn ذخیره کرده ولی P1 که دیرتر شماره اش را ذخیره کرده، شماره اش در turn می ماند وturn برابر1 می شود. حال زمانی که P0 وP1 به دستور While می رسند، PO از حلقه عبور کرده و وارد ناحیه بحرانی می شود ولی P1 در حلقه می جِرخد (انتظار مشغول) و وارد ناحیه بحرانی نمی شود.

الگوریتم Peterson، شرایط انحصار متقابل، انتظار محدود و پیشرفت را با هم رعایت می کند. و مشکل بن بست وLivelock وقحطی ندارد. این الگوریتم ساده ترین و کوتاهترین راه حل نرم افزاری ست.

معایبی که در هر یک از تلاش های Decker و همچنین روش Peterson وجود دارند، عبارتنداز:

۱- اگریکی از فرایندها در داخل ناحیه بحرانی از کار بیفتد، فرایند دیگر تا ابد منتظر می ماند.

۲- مبتنی بر انتظار مشغول می باشند.

رویکردهای انحصار متقابل با حمایت سخت افزار

می توان به کمک راه حل هایی که از دستور العمل های ویژه ماشین استفاده می کنند، و نیاز به حمایت از طرف پردازنده دارند، مسئله انحصار متقابل راحل کرد. این راه حل ها عبارتنداز:

۱- دستور العمل از کار انداختن وقفه ها

۲- دستور العمل TSL

۳- دستور العمل SWAP

دستور العمل از کار انداختن وقفه ها

در این راه حل، هر فرایند باید به محض ورود به ناحیه بحرانی، تمام وقفه ها را از کار بیندازد و دقیقا قبل از خروج از ناحیه بحرانی، همه وقفه هارا مجددا فعال سازد. در این صورت پردازنده نمی تواند از فرایندی به فرایند دیگر سوئچ کند، چون وقفه ساعت غیر فعال است. بنابراین وقتی که فرایندی وقفه ها را غیر فعال می کند، می تواند بدون ترس از دخالت فرایندهای دیگر، به خواندن و نوشتن در حافظه مشترک بپردازد.

رویکرد از کار انداختن وقفه ها برای انحصار متقابل به صورت زیر است:

```
P(int i){
while(TRUE)
{
disable_interrupts();
critical_section();
enable_interrupts();
non_critical_section();
}
}
```

امکان دارد فرایندی وقفه ها را بعد از غیر فعال کردن، مجددا فعال نکند. (از معایب این روش)

در سیستم های چندپردازنده ای، غیر فعال کردن وقفه ها، فقط در پردازنده ای که این دستور العمل را جرا کرده تاثیر گذار است وپردازنده های دیگر می توانند به ناحیه بحرانی دسترسی داشته باشند.

دستور العمل TSL

بسیاری از کامپیوترها دارای دستورالعمل TSL(Test and Set Lock) می باشند. این دستور محتویات یک کلمه از حافظه (lock) را خوانده ودر ثبات قرار می دهد. سپس مقدار 1 رادر همان آدرس از حافظه ذخیره می کند. این عملیات غیر قابل تقسیم انجام می شوند و تا اینکه اجرای دستور العمل تمام نشود، هیچ فرایند و یا پردازنده دیگری نمی تواند به این کلمه از حافظه دسترسی پیدا کند. در زیر تابع ورود به ناحیه بحرانی (enter_region) با استفاده از دستوالعمل TSL ، به زبان اسمبلی آورده شده ست:

```
enter_region:
tsl  reg , lock
cmp  reg , #0
jne  enter_region
ret
```

توسط اولین دستور، مقدار قبلی lock در رجیستر ذخیره شده و سپس در lock مقدار 1 ذخیره می شود.

توسط دستور دوم مقدار قبلی lock با 0 مقایسه می شود. اکر 0 نبود، این عملیات تکرار شده (حلقه انتظار مشغول) و اگر 0 بود، زیربرنامه بازگشت کرده، در حالی که lock را 1 کرده است

تذکر: برای پاک کردن lock در هنکام خروج از ناحیه بحرانی، کافی است در ان 0 را ذخیره کرد:

```
move lock , #0
ret
```

راه حل داده شده، انحصار متقابل و پیشرفت را رعایت می کند ولی انتظار محدود به دلیل گرسنگی را رعایت نمی کند.

دستور swap

می توان از دستورالعملی به نام swap ، برای نوشتن یک روال ورود به ناحیه بحرانی استفاده کرد. این دستور می تواند در یک عمل واحد غیر قابل تقسیم، محتویات یک رجیستر پردازنده را با محتویات یک کلمه حافظه، جابجا کند.

```
enter_region:
move reg, #1
 swap reg, lock
 cmp reg, #0
 jne enter_region
ret
```

تذکر: برای پاک کردن lock در هنگام خروج از ناحیه بحرانی، کافی است در آن 0 را ذخیره کرد.

تذکر: xchg نام دیگر swap می باشد.

راه حل داده شده، انحصار متقابل و پیشرفت را رعایت می کند ولی انتظار محدود به دلیل گرسنگی را رعایت نمی کند..

سمافور

راه حل های نرم افزاری و سخت افزاری که بررسی کردیم ، دارای نقاط ضعف زیادی بودند. حال ابزاری به نام سمافور را معرفی می کنیم که دارای قدرت زیادی در برقراری انحصار متقابل می باشد و همچنین از عهده انواع مختلف مسایل همکام سازی بر می اید.

سمافور یک ساختار شامل فیلدهای زیر است:

۱- شمارنده صحیح (count)

از شمارنده برای شمارش تعداد wakeup هایی که می خواهند هدر بروند، استفاده می شود. با این کار این سیگنال ها برای استفاده های بعدی ذخیره می شوند.

2- صف (queue)

از صف برای نگهداری فرایندهای بلوکه شده بر روی سمافور استفاده می شود.

توسط Dijkstra ،دو تابع signal gwait مطرح شدند که به ترتیب تعمیم یافته sleep و wakeup هستند.

توابع wait وsignal به صورت زیر است:

```
void wait(semaphore s)
{
   s.count = s.count - 1;
   if (s.count < 0)
   {
      place this process in s.queue;
      block this process;
   }
}
```

```
void signal(semaphore s)
{
   s.count = s.count + 1;
   if (s.count <= 0)
   {
      remove a process from s.queue;
      place this process in ready queue;
   }
}
```

تابع wait ، یک واحد از شمارنده کم کرده و اگر منفی شود، فرایند مربوطه در صف، مسدود می شود.

تابع signal، یک واحد به شمارنده اضافه می کند، اگر مقدار شمارنده بیشتر از صفر نشود، یک فرایند از قبل مسدود شده در صف، آزاد می شود.

- در بعضی ازمتون از حرف Pبه جای wait واز حرف V به جای signal استفاده می شود.
- در بعضی از متون از down به جای wait واز up به جای signal استفاده می شود.

سمافور بر دو نوع عمومی و دودویی می باشد. در سمافور عمومی که آن را بررسی کردیم، شمارنده می تواند مثبت، صفر و یا منفی باشد. اما شمارنده در سمافور دودویی، فقط مقادیر 0 و1 را دریافت می کند.

توابع signalg wait برای سمافور باینری به صورت زیر است:

void wait(semaphore s)
{
if (s.count == 1)
s.count = 0;
else {
place this process in
s.queue;
block this process;
}
}
	void signal(semaphore s)
	{
	if (s.queue is empty)
	s.count = 1;
	else {
	remove a process from s.queue;
	place this process in ready queue;
	}
	}

- قدرت سمافور دودویی معادل با سمافور عمومی است.

در سمافورها صفی برای نگهداری فرایندهای بلوکه شده استفاده می شود. اگر خروج از این صف به ترتیب ورود باشد (FIFO)،به آن سمافور قوی می گویند و اگر این چنین نباشد، به آن سمافور ضعیف می گویند . در سمافور ضعیف، امکان گرسنگی وجود دارد. در این کتاب سمافورها از نوع قوی فرض می شوند.

انحصار متقابل با استفاده از سمافورها

می خواهیم انحصار متقابل را به کمک سمافورها برقرار کنیم. برای حل یک مسئله با سمافور، باید بدانیم

که:

۱- به چند سمافور نیاز است.

مقدار اولیه شمارنده هر سمافور چند باید باشد.

عمل signal و wait در کجای برنامه باید روی سمافور انجام شود.

مسئله انحصار متقابل برای دوفرایند p1 و p2 توسط برنامه زیر پیاده سازی شده است. در این برنامه عمل waitقبل از ورود به ناحیه بحرانی وعملsigna بعداز خروج از ناحیه بحرانی روی سمافور دودویی mutex با مقدار اولیه 1 انجام شده است:

```
semaphore  mutex =1;
void  p(int  i)
{
while(TRUE)
{
wait(mutex);
critical-section( );
signal(mutex);
non-critical-section( );
}
```

فرض کنیم P1 اول اجرا شود. با اجرای تابع wait، چون مقدار سمافور یک است،آن را صفر می کند و وارد ناحیه بحرانی می شود. در زمانی که P1 در ناحیه بحرانی است، اگر P2 سعی به ورود به ناحیه بحرانی داشته باشد، چون سمافور برابر 0 است، این فرایند بلوکه شده و در صف قرار می گیرد. بعد از خروج P1 از ناحیه بحرانی، تابع signal را اجرا کرده و چون صف خالی نیست، P2 که در صف قرار دارد را ازاد می کند و P2 می تواند وارد ناحیه بحرانی شود

- می توان با استفاده از سمافور عمومی، انحصار متقابل را برای بیش از دو فرایند نیز پیاده سازی کرد.

- سمافوری که ترتیب خروج در آن مشخص نباشد، سمافور ضعیف نامیده می شود. در این نوع سمافور، امکان گرسنگی وجود دارد.(در سمافور قوی، منطق صف FIFO است).

- پیاده سازی انحصار متقابل به کمک سمافور دارای مزایایی است از جمله: ارضای شرط های انحصار متقابل، پیشرفت و انتظار محدود. این روش عادلانه است،CPU را تلف نمی کند و مشکل اولویت معکوس ندارد.

همگام سازی با استفاده از سمافورها

سمافور دارای توانایی زیادی در حل مسائل همگام سازی دارد. دو فرایند را در نظر بگیرید که باید ابتدا دستور S1 در P1 و سپس دستور S2 در P2 اجرا شود. برای پیاده سازی این همگام سازی از سمافوری به نام S

با مقدار اولیه صفر به صورت زیر استفاده می کنیم:

P1	P2
. S1; Signal(s); .	. Wait(s); S2; .

مشخص است که تازمانی که S1 از P1 اجرا نشود، اجرای S2 ممکن نیست

تذکر: معمولا مقدار اولیه سمافور در همگام سازی برابر صفر و در انحصار متقابل برابر 1 است.

مثال

ایا ترتیب اجرای P1,P2,P3 ممکن است؟ (مقدار اولیه سمافورها S و Q برابر صفر است)

P1	P2	P3
. . Signal(s) ;	Wait(Q) ; . .	Wait(S) ; . Signal(Q)

پاسخ : خیر.

بعد از اجرای P1 ، فرایند P2 نمی تواند اجرا شود، چون در ابتدای P2 دستور wait(Q) قرار دارد که به علت یک بودن سمافور Q، باعث بلوکه شدن P2 می شود. ولی اگر بعد از P1 ، فرایند P3 اجرا شود، به علت وجود دستور signalI(Q) در انتهای آن، می توان بعد از P3 فرایند P2 را اجرا کرد. پس یک ترتیب اجرای ممکن عبارت است از : P1,P3,P2.

مثال

بافرض اینکه مقدار اولیه دو سمافور S و Q برابر صفر باشد، نحوه چاپ ABCDE را مشخص کنید.

P0	P1
Signal(Q) Wait(S) ; cout<<"C"; cout<<"D";	Wait(Q) ; cout<<"A"; Signal(s) ; cout<<"B"; cout<<"E";

پاسخ:

در زیر ترتیب اجرا با شماره مشخص شده است:

P0	P1
1 Signal(Q) 6 Wait(S) ; 7cout<<"C"; 8cout<<"D";	2 Wait(Q) ; 3 cout<<"A"; 5 Signal(s) ; 4cout<<"B"; 9 cout<<"E";

مثال

نحوه چاپ 1324 را مشخص کنید؟ (مقدار اولیه دو سمافور S و Q برابر صفر است)

P1	P2
cout<<"1"; Signal(s) ; Wait(Q) ; cout<<"2"; Signal(s) ;	Wait(S) ; cout<<"3"; Signal(Q) ; Wait(S) ; cout<<"4";

پاسخ: به علت استفاده از دستور wait(S) در ابتدای P2 ، و صفر بودن S ، اجرا نمی تواند با P2 شروع شود.

بنابراین ابتدا P1 اجرا شده و عدد 1 چاپ می شود. سپس توسط signal(S) مقدار سمافور S برابر یک می شود. به علت رسیدن به wait(Q) ، و صفر بودن Q ، نمی توان ادامه داد . با تعویض متن به p2 ، چون S برابر یک شده از Wait(S) رد شده و S صفر می شود. سپس 3 چاپ می شود. حال دستور signal(Q) مقدار

Q را برابر یک کرده وبا رسیدن به wait(S) ، چون S ، صفر است، نمی توان ادامه داد و به ادامه P1 رفته و چون Q برابر یک است از wait(Q) رد شده و Q برابر صفر شده و سپس عدد 2 چاپ می شود. در نهایت توسط signal(S) مقدار S برابر یک شده وبه P2 پرش کرده واز wait(S) عبور کرده و مقدار 4 چاپ می شود. در زیر ترتیب اجرا با شماره مشخص شده است:

P1	P2
1 cout<<"1"; 2 Signal(s) ; 6 Wait(Q) ; 7 cout<<"2"; 8 Signal(s) ;	3 Wait(S) ; 4 cout<<"3"; 5 Signal(Q) ; 9 Wait(S) ; 10 cout<<"4";

مثال نحوه اجرای فرایندهای P1 و P2 چگونه باشد، تا مقدار نهایی برابر a=10 , b=6 شود؟
(مقدار اولیه سمافور S برابر یک و مقدار اولیه متغیرهای a,b نیز برابر یک می باشند.)

P1	P2

a=a+2;
b=b+1;
signal(s);
b=b+1;
	a=3;
	wait(s);
	b=a+b;
	wait(s);
	a=a+b;

پاسخ :

ابتدا سه دستور اول P2 اجرا می شود. بعد از اجرای این سه دستور داریم: a=3,b=4,5=0

سپس سه دستور اول P1 اجرا شده و بعد از اجرا خواهیم داشت: a=5,b=5,5=1

مجددا به P2 برگشته و دو دستور بعدی آن اجرا شده و داریم: a=10,b=5, S=0

در نهایت به P1 رفته و بعد از اجرای دستور باقی مانده خواهیم داشت: a=10,b=6

مثال

با فرض اینکه مقدار اولیه سمافور S برابر 8 ، سمافور P برابر 3 وسمافور Q برابر 1 باشد، حداکثر چند فرایند پشت هر سمافور قرار می گیرد؟

.

wait(S);

wait(P);

wait(Q);

.

signal(Q);

signal(P);

signal(S);

.

پاسخ:

از n فرایند، حداکثر 8 فرایند می تواند از wait(S) عبور کند وبقیه (n-8) فرایند در صف سمافور S می خوابند. از 8 فرایندی که از سمافور S عبور کرده، حداکثر 3 فرایند می تواند از سمافور P عبور کند و 5 فرایند در صف Pمی خوابند. در نهایت از 3 فرایندی که از سمافور P عبور کرده، حداکثر 1 فرایند از

عبور کرده و 2 فرایند در صفQ می خوابند.

مسئله تولید کننده و مصرف کننده

یک تولیدکننده یا بیشتر، نوعی داده را تولید و آنها را در بافری به اندازه n قرار می دهند. یک مصرف کننده، این اقلام را یکی یکی از بافر برمی دارد. سیستم باید از همپوشانی اعمال بافر جلوگیری کند، یعنی در هر زمان مصرف کننده یا تولید کننده می تواند به بافر دسترسی داشته باشد.

این راه حل از سه سمافور استفاده می کند:

1- سمافور Mutex

سمافوری برای رعایت شرط انحصار متقابل است، تا تولید کننده و مصرف کننده به طور همزمان به بافر دسترسی نداشته باشند. (با مقدار اولیه 1)

2-سمافورfull

سمافوری برای شمارش تعداد خانه های پر بافر (با مقدار اولیه 0)

3- سمافور empty

سمافوری برای شمارش تعداد خانه های خالی بافر (با مقدار اولیه n)

void producer (void){
int item;
while(TRUE) {
item= produce( );
wait(empty);
wait(mutex);
insert(item);
signal(mutex);
signal(full); }
}
	void consumer(void){
	int item;
	while(TRUE) {
	wait(full);
	wait(mutex);
	item=remove( );
	signal(mutex);
	signal(empty);
	consume( ); }
	}

اگر در مسئله تولید کننده- مصرف کننده ، بافر نامحدود باشد، دیگر نیازی به سمافور empty نیست. در نتیجه دستور wait(empty) از تولید کننده و دستور signal(empty) از مصرف کننده حذف می‌شود.

مسئله غذا خوردن فیلسوف ها

پنج فیلسوف دور یک میز دایره ای نشسته اند. هر فیلسوف یک بشقاب ماکارونی دارد. بین هر جفت از بشقاب ها، یک چنگال قرار دارد.



هر فیلسوف برای خوردن از دو چنگال طرفین بشقاب استفاده می کند. زندگی هر فیلسوف از دو دوره متناوب خوردن وفکر کردن تشکیل شده است. زمانی که هر فیلسوف گرسنه می شود، سعی می کند دو چنگال سمت چپ و راست خود را بردارد. اگر موفق شد، برای مدتی غذا می خورد و سپس چنگال هارا زمین می گذارد وبه فکر کردن ادامه می دهد. مسأله تغذیه

فیلسوفان علاوه بر انحصار متقابل (که در یک زمان دو فیلسوف نمی توانند از یک چنگال استفاده کنند)، باید جوابگوی بن بست و گرسنگی نیز باشد.

راه حل: هرفیلسوف ابتدا چنگال چپ و سپس چنگال راست رابر می دارد. بعد از تغذیه یک فیلسوف، دو چنگالی که استفاده می کرد را روی میز گذاشته و دیکَران می توانند استفاده کنند.

```
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( ); //    ناحیه بحرانی
signal ( fork[(i+1) % 5] );
signal ( fork[i] );
signal ( room );
}

}

void main( ){
parbegin (p(0),p(1),p(2),p(3),p(4)));
}
```

سمافور room با مقدار اولیه 4، برای این است که اجازه ورود به بیش از چهار نفر داده نشود. اگر حداکثر چهار فیلسوف نشسته باشند، حداقل یک نفر به دو چنگال دسترسی خواهد داشت. اگر از room استفاده نمی شد، واجازه ورود همزمان هر پنج نفر، داده می شد، همه آنها

چنگالهای چپ خود را برداشته و دیگر چنگال اضافی نمی ماند که کسی بتواند چنگال راست خود را بردارد. بنابراین بن بست رخ می داد.

در کتاب تننباوم، راه حل زیر داده شده است. در این راه حل، از یک آرایه به نام state استفاده می کند که وضعیت جاری فیلسوف(فکر کردن(0) ، کرسنکَی(1) و خوردن(2)) را نگَهداری می کند. یک فیلسوف در صورتی که هیچ یک از فیلسوفان چپ و راستش، در حال خوردن نباشند، می تواند غذا بخورد. این راه حل بن بست ندارد.

```
#define LEFT  (i-1) % 5
#define RIGHT (i+1) % 5
typedef int semaphore;
semaphore mutex=1;
semaphore s[5];
int state[5];

void philosopher (int i){
   while(TRUE){
      think( );
      take_forks(i);
      eat( );
      put_forks(i);
   }
}

void take_forks(int i){
   wait(mutex);
   state[i]= 1;
   test(i);
   signal(mutex);
   wait(s[i]);
}

void put_forks(int i){
   wait(mutex);
   state[i]=0;
   test( LEFT );
   test( RIGHT );
   signal(mutex);
}

void test(int i){
   if( state[i]==1 && state[LEFT]!=2 && state[RIGHT]!=2 ) {
      state[i]=2;
      signal(s[i]);
   }
}
```

مسئله خوانندگان و نویسندگان

در این مسئله، ناحیه داده ای(مثل فایل) وجود دارد که بین تعدادی از فرایندها مشترک است. فرایند های خواننده می خواهند از این ناحیه بخوانند وفرایندهای نویسنده می خواهند در آن بنویسند

شرایط این مسئله:

ا- هر تعداد ازخوانندگان می توانند به صورت همزمان از فایل بخوانند.

۲- در هر زمان تنها یک فرایند ممکن است در این فایل بنویسد

۳- هنگامی که نویسنده ای در حال نوشتن است، هیچ خواننده ای نمی تواند فایل را بخواند.

برای مثال یک سیستم رزرواسیون هواپیمایی را در نظر بگیرید که تعداد زیادی فرایند در آن برای نوشتن و خواندن با یکدیگر رقابت می کنند. چند فرایند می توانند به طور همزمان پایگاه داده را بخوانند ولی اگر یک فرایند در حال به روز رسانی پایگاه داده باشد، فرایندهای دیگر حتی خوانندگتن، نباید به پایگاه داده دسترسی داشته باشند.

این مسئله دارای سه حالت است:

۱) خوانندگان اولیت دارند. (تا زمانی که خواننده ای وجود دارد، به خواننده اجازه ورود می دهیم)

۲) نویسندگان اولویت دارند.

۳) خوانندگان و نویسندگان بدون اولیت هستند

حالتی را بررسی می کنیم که خوانندگان اولویت دارند:

```
typedef int semaphore;
```

```
semaphore mutex=1;
semaphore w=1;
int rc=0;
void writer( )
}
while(TRUE)
}
wait(w);
writing( );
signal(w);
}
}
void reader( )
}
while(TRUE)
}
wait(mutex);
rc = rc+1;
if (rc == 1) wait(w);
signal(mutex);
reading( );
wait(mutex);
rc = rc-1;
if (rc == 0) signal(w);
signal(mutex);
}
}
```

روال نویسنده:

اولین نویسنده، باعمل Wait(w) ، اجازه دسترسی پیدا کرده (جون مقدار اولیه W برابر 1 است)، ولی با صفر شدن این سمافور، اجازه دسترسی نویسندگان و خوانندگان دیگر گرفته می شود. در این روال، تا هنگامی که نویسنده ای به فایل دسترسی دارد، هیچ نویسنده ای و خواننده دیگری نمی تواند به فایل دسترسی داشته باشد.

روال خواننده:

فرایند خواننده از سمافور W برای اعمال انحصار متقابل، از متغیر سراسری RC ، برای شمارش تعداد خوانندگان واز سمافور

mutex برای اطمینان از تغییر مناسب

دستور افزایش rc وکنترل آن در بین wait(mutex) و signal(mutex) قرار دارد، تا تغییر rc ، انحصاری شود. همچنین

بعد از پایان عمل خواندن، دستور کاهش rc و کنترل آن در بین Wait(mutex) و signal(mutex) قرار دارد تا تغییر rc

، انحصاری شود.

توسط if اول، به اولین خواننده اجازه ورود داده می شود،البته در صورتی که نویسنده ای فعال نباشد.

توسط if پایانی، بررسی می کنیم که اکر خواننده فعال دیکَری وجود نداشته باشد، در صورت اینکه نویسنده بلوکه شده ای داشته باشیم، به آن اجازه داده شود.

- در این روش شرط انحصار متقابل وشرط بیشرفت رعایت می شود ولی شرط انتظار محدود رعایت نمی شود.
- (چون اولویت با خوانندگان است و امکان کرسنگی نویسندکان وجود دارد.)