موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

سیستم عامل

(ویژه کنکور ارشد ۱۴۰۲)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

آبراهام سیلبرشاتز، ویلیام استالینگز و اندور اس تننباوم

ارسطو خلیلی‌فر

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

تقدیم به:

تمامی آنانی که برای پیشرفت و سعادت خود و بشریت

تلاش می کنند.

ارسطو خلیلی فر

khalilifar.ir

فهرست مطالب

تست‌های فصل اول: مفاهیم اولیه ۹
پاسخ تست‌های فصل اول: مفاهیم اولیه ۱۰

تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده ۱۴
پاسخ تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده ۱۸

تست‌های فصل سوم: مدیریت نخ ۴۰
پاسخ تست‌های فصل سوم: مدیریت نخ ۴۳

تست‌های فصل چهارم: مدیریت حافظه اصلی ۵۵
پاسخ تست‌های فصل چهارم: مدیریت حافظه اصلی ۶۰

تست‌های فصل پنجم: مدیریت حافظه مجازی ۱۱۱
پاسخ تست‌های فصل پنجم: مدیریت حافظه مجازی ۱۱۷

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند ۱۵۱
پاسخ تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند ۱۵۹

تست‌های فصل هفتم: مدیریت بن‌بست ۲۵۳
پاسخ تست‌های فصل هفتم: مدیریت بن‌بست ۲۵۷

تست‌های فصل هشتم: مدیریت دیسک ۲۷۴
پاسخ تست‌های فصل هشتم: مدیریت دیسک ۲۷۶

سؤالات کنکور کارشناسی ارشد سال ۱۳۹۷ ..... ۲۸۵
پاسخ سؤالات کنکور کارشناسی ارشد سال ۱۳۹۷ ..... ۲۸۸

سؤالات کنکور کارشناسی ارشد سال ۱۳۹۸ ..... ۳۲۳
پاسخ سؤالات کنکور کارشناسی ارشد سال ۱۳۹۸ ..... ۳۲۶

سؤالات کنکور کارشناسی ارشد سال ۱۳۹۹ ..... ۴۲۱
پاسخ سؤالات کنکور کارشناسی ارشد سال ۱۳۹۹ ..... ۴۲۶

سؤالات کنکور کارشناسی ارشد سال ۱۴۰۰ ..... ۵۱۷
پاسخ سؤالات کنکور کارشناسی ارشد سال ۱۴۰۰ ..... ۵۲۱

khalilifar.ir

تست‌های فصل اول: مفاهیم اولیه

۱- کدامیک از موارد زیر نادرست است؟ (مهندسی IT- دولتی۸۹)

۱) تنظیم زمان سیستم در مود کاربر انجام می‌شود.
۲) کامپیوتر در هنگام روشن شدن در مود کرنل قرار می‌گیرد.
۳) تغییر اولویت فرآیندها در مود کرنل انجام می‌شود.
۴) خواندن ساعت سیستم در مود کاربر انجام می‌شود.

۲- هدف اصلی از عملیات Dual-mode چیست؟ (مهندسی IT- دولتی۹۲)

۱) تشخیص خطاها
۲) کاهش توان مصرفی کامپیوتر
۳) توانایی سیستم عامل در کنترل پردازنده
۴) محافظت سیستم عامل از دیگر نرم‌افزارها

۳- در یک کامپیوتر، روتین سرویس‌دهی به وقفه ساعت (Interrupt Service Routine) به 2 میلی‌ثانیه زمان برای اجرا نیاز دارد. این زمان شامل زمان اجرا و زمان تعویض زمینه می‌باشد. این روتین سرویس‌دهی به وقفه ساعت در هر پالس ساعت اجرا شده و پالس ساعت دارای فرکانس 75 هرتز می‌باشد. چه درصدی از زمان پردازنده برای این روتین سرویس‌دهی به وقفه اختصاص می‌یابد؟ (مهندسی IT- دولتی ۹۶)

۱) 5 ۲) 7.5 ۳) 10 ۴) 15

---

khalilifar.ir

پاسخ تست‌های فصل اول: مفاهیم اولیه

۱- گزینه (۱) صحیح است.

در سطح پردازنده دو نوع کار اجرا می گردد، (۱) کارهای ممتاز (۲) کارهای غیرممتاز، کارهای ممتاز در مود هسته پردازنده و کارهای غیرممتاز در مود کاربر پردازنده اجازه‌ی اجرا دارند. بنابراین هیچگاه یک کار ممتاز اجازه اجرا در مود کاربر را نخواهد داشت.

توجه: کارهای سیستم عامل در مود هسته و کارهای کاربر (برنامه‌های کاربردی) در مود کاربر اجرا می‌شوند.

توجه: عملیات ورودی و خروجی، عمل تقسیم بر صفر، تغییر تایمر سیستم، تغییر ثبات‌های حفاظتی، ازکار انداختن وقفه، دسترسی به حافظه سیستم عامل، دسترسی بدون اجازه به حافظه فرآیندهای دیگر نمونه‌هایی از کارهای ممتاز محسوب می‌گردند. در یک بیان کلی‌تر، هرکاری که استفاده نادرست ازآن باعث مختل شدن بخشی از سیستم کامپیوتری یا کل سیستم کامپیوتری گردد، کار ممتاز به حساب می‌آید.

توجه: انجام کار ممتاز برای یک فرآیند کاربر، کار غیرمجاز است. و فقط انجام کارهای غیرممتاز است که برای یک فرآیند کاربر، مجاز محسوب می‌گردد.

هرگاه یک فرآیند کاربر قصد اجرای یک کار ممتاز را توسط پردازنده داشته باشد، پردازنده فوراً توسط یک وقفه، سیستم عامل را باخبر می‌کند و کنترل به سیستم عامل واگذار می‌گردد تا سیستم عامل در پاسخ به آن وقفه یک روال پاسخ به وقفه را اجرا کند.

توجه: دقت کنید که در هر لحظه فقط یک برنامه قادر به اجرا توسط پردازنده است، درواقع در لحظه‌ای که برنامه کاربر در حال اجرا است و تصمیم به انجام یک کار ممتاز می‌گیرد، سیستم عامل اصلاً در حال اجرا نیست که بخواهد جلوی این کار غیرمجاز فرآیند در حال اجرا را بگیرد، اما سیستم عامل این مسأله را قبلاً با پردازنده و سخت‌افزار هماهنگ کرده است، پردازنده و سخت‌افزار در غیاب سیستم عامل، به محض مشاهده یک کار غیرمجاز از فرآیند در حال اجرا، فوراً توسط صدور یک وقفه، سیستم عامل را وارد صحنه می‌کنند. در واقع به واسطه وقفه، پردازنده از فرآیند خاطی گرفته شده و در اختیار سیستم عامل قرار داده می‌شود، تا سیستم عامل بتواند به امور فرآیند خاطی رسیدگی کند.

یک بیت در رجیستر PSW (Program Status Word) مود اجرای پردازنده را نگهداری می‌کند. در ابتدای کار یعنی در هنگام راه‌اندازی کامپیوتر، این بیت برابر صفر است، بنابراین سیستم عامل در مود هسته آغاز به کار می‌کند. حال اگر در ادامه و بر حسب لزوم تصمیم گرفته شود که پردازنده در اختیار فرآیند کاربر قرار داده شود، قبل از آنکه پردازنده از سیستم عامل گرفته شود و به برنامه کاربر سپرده شود، بیت مود پردازنده از 0 به 1 تغییر می‌کند (پردازنده به مود کاربر سوئیچ می‌کند)

khalilifar.ir

و از آنجا که هر وقفه ناشی از کار غیرمجاز (انجام کار ممتاز در برنامه کاربر) پردازنده را در اختیار سیستم عامل قرار می‌دهد، بنابراین با وقوع هر وقفه (ناشی از کار غیرمجاز)، این بیت مجدداً صفر می‌گردد.

گزینه اول نادرست است. زیرا، تنظیم نادرست زمان سیستم منجر به مختل شدن سیستم کامپیوتری می‌گردد، بنابراین این کار یک کار ممتاز است و باید در مود هسته پردازنده اجرا گردد.

گزینه چهارم درست است. زیرا، خواندن ساعت سیستم منجر به مختل شدن سیستم کامپیوتری نمی‌گردد، بنابراین این کار، یک کار غیر ممتاز است و می‌تواند در مود کاربر پردازنده اجرا گردد.

۲- گزینه (۴) صحیح است.

در سطح پردازنده دو نوع کار اجرا می‌گردد، (۱) کارهای ممتاز (۲) کارهای غیرممتاز، کارهای ممتاز در مود هسته پردازنده و کارهای غیرممتاز در مود کاربر پردازنده اجازه‌ی اجرا دارند. بنابراین هیچگاه یک کار ممتاز اجازه اجرا در مود کاربر را نخواهد داشت.

توجه: کارهای سیستم عامل در مود هسته و کارهای کاربر (برنامه‌های کاربردی) در مود کاربر اجرا می‌شوند.

توجه: عملیات ورودی و خروجی، عمل تقسیم بر صفر، تغییر تایمر سیستم، تغییر ثبات‌های حفاظتی، از کار انداختن وقفه، دسترسی به حافظه سیستم عامل، دسترسی بدون اجازه به حافظه فرآیندهای دیگر نمونه‌هایی از کارهای ممتاز محسوب می‌گردند. در یک بیان کلی‌تر، هر کاری که استفاده نادرست از آن باعث مختل شدن بخشی از سیستم کامپیوتری یا کل سیستم کامپیوتری گردد، کار ممتاز به حساب می‌آید.

توجه: انجام کار ممتاز برای یک فرآیند کاربر، کار غیرمجاز است. و فقط انجام کار غیرممتاز است که برای یک فرآیند کاربر، مجاز محسوب می‌گردد.

هرگاه یک فرآیند کاربر قصد اجرای یک کار ممتاز را توسط پردازنده داشته باشد، پردازنده فوراً توسط یک وقفه، سیستم عامل را باخبر می‌کند و کنترل به سیستم عامل واگذار می‌گردد تا سیستم عامل در پاسخ به آن وقفه یک روال پاسخ به وقفه را اجرا کند.

توجه: دقت کنید که در هر لحظه فقط یک برنامه قادر به اجرا توسط پردازنده است، درواقع در لحظه‌ای که برنامه کاربر در حال اجرا است و تصمیم به انجام یک کار ممتاز می‌گیرد، سیستم عامل اصلاً در حال اجرا نیست که بخواهد جلوی این کار غیرمجاز فرآیند در حال اجرا را بگیرد، اما سیستم عامل این مسأله را قبلاً با پردازنده و سخت‌افزار هماهنگ کرده است، پردازنده و سخت‌افزار در غیاب سیستم عامل، به محض مشاهده یک کار غیرمجاز از فرآیند در حال اجرا، فوراً توسط صدور یک وقفه، سیستم عامل را وارد صحنه می‌کنند. در واقع به واسطه وقفه، پردازنده از فرآیند خاطی گرفته شده و در اختیار سیستم عامل قرار داده می‌شود، تا سیستم عامل بتواند به امور فرآیند خاطی رسیدگی کند.

khalilifar.ir

توجه: یکی از کاربردهای دو مود هسته و کاربر، محافظت سیستم عامل در برابر دسترسی های غیرمجاز است. در واقع فضای هسته از فضای کاربر جدا می‌گردد و سیستم عامل و حتی در سطح بالاتر خود، یعنی سیستم کامپیوتری در برابر برنامه‌های سطح کاربر محافظت می‌شود.

۳- گزینه (۴) صحیح است.

هر وقفه یک روتین یا روال پاسخ به وقفه دارد که به آن ISR که سرواژه عبارت Interrupt Service Routine و به معنی روتین سرویس‌دهی به وقفه است، گفته می‌شود. ISR ها توسط سازنده سیستم عامل نوشته می‌شود و بخشی از هسته سیستم عامل محسوب می‌شود. در واقع با وقوع یک وقفه، برنامه کاربر قطع و یک تکه برنامه از هسته سیستم عامل به نام ISR در پاسخ به وقفه اجرا می‌شود. به عبارت بهتر با توجه به نوع Interrupt ای که آمده، ISR مربوط به آن اجرا می‌شود. هر Interrupt برای خود یک ISR مختص به خود را دارد، یعنی به تعداد وقفه‌ها، ISR داریم. برای مثال وقتی دوستتان یا پست‌چی زنگ می‌زند، روال برخورد با آن‌ها متفاوت است، کسی که وقفه را می‌دهد باید مشخصات خود را نیز بدهد، وقتی که وقفه‌دهنده شناخته شد، روال نوع پاسخ به آن وقفه پیگیری می‌شود. برای اینکار جدول وقفه بررسی می‌شود تا مشخص شود کدام ISR باید فراخوانی شود. برای مثال در قدیم کوبه درب مخصوص زنان و مردان متفاوت بود و صدای هر کوبه عکس‌العمل مختص به خود را توسط اهالی خانه در پی داشت، سیستم عامل جهت تسریع در پاسخ دادن به وقفه‌ها، آدرس روال‌های سرویس‌دهنده به آن‌ها را در جدولی به نام جدول توصیف وقفه (Interrupt Descriptor Table) نگهداری می‌کند که به ازای هر وقفه یک درایه در این جدول وجود دارد که به آن بردار وقفه (Interrupt Vector) نیز گفته می‌شود.

شروع و کشف نوع وقفه سخت‌افزاری است اما پاسخ به وقفه نرم‌افزاری و توسط سیستم عامل و اجرای ISR مرتبط با نوع وقفه است.

مطابق فرض سؤال روتین سرویس‌دهی به وقفه ساعت در هر پالس ساعت اجرا شده و پالس ساعت دارای فرکانس 75 هرتز می‌باشد. در واقع ISR با فرکانس 75Hz اجرا می‌شود، یعنی 75 بار در ثانیه، همچنین مطابق فرض سؤال روتین سرویس‌دهی به وقفه ساعت (Interrupt Service Routine) به 2 میلی ثانیه زمان برای اجرا نیاز دارد که این زمان شامل زمان اجرا و زمان تعویض زمینه (متن) می‌باشد.

پس در مجموع زمان پردازش ISR در هر ثانیه به صورت زیر محاسبه می‌گردد:

زمان پردازش ISR در هر ثانیه $= 75 \times 2\,\text{ms} = 150\,\text{ms}$

یعنی در هر ثانیه 150ms برای ISR مصرف می‌شود.

خواسته سؤال این است که چه درصدی از زمان پردازنده برای این روتین سرویس‌دهی به وقفه اختصاص می‌یابد، که در تناسب زیر داریم:

khalilifar.ir

زمان	درصد	
$150\,ms$	x	
$1sec = 1000\,ms$	100	$\rightarrow \quad x = \frac{150 \times 100}{1000} = 15\%$

khalilifar.ir

تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۱- فرض کنید 5 فرآیند با مشخصات زیر به یک سیستم با زمان‌بند چرخشی (Round – Robin) با برش زمانی $q = 1$ وارد شوند: (e: execution time, r: release time)
با فرض این که همیشه بین فرآیندی که در لحظه t برش زمانی خود را به پایان می‌رساند و فرآیند ورودی در لحظه t اولویت با فرآیند قبلی موجود در سیستم است و در شرایط کاملاً یکسان بین دو فرآیند، اولویت با فرآیند با شماره کوچک‌تر است. میانگین زمان انتظار فرآیندهای فوق کدام است؟ (مهندسی IT - دولتی۸۹)

P_1 : r=0 , e=2
P_2 : r=0 , e=3
P_3 : r=1 , e=2
P_4 : r=1 , e=3
P_5 : r=2 , e=2

۱) 6.4 ۲) 5.8 ۳) 6.2 ۴) 5.6

۲- اگر در یک سیستم دو فرآیند P_1 و P_2 داشته باشیم که jobهای فرآیند P_1 به صورت دوره‌ای هر 5 ثانیه یکبار و jobهای فرآیند P_2 به صورت دوره‌ای هر 4 ثانیه یکبار به سیستم وارد شوند و زمان اجرای هر job از فرآیند P_1 برابر با 3 ثانیه و زمان اجرای هر job از فرآیند P_2 برابر با 1 ثانیه باشد، بهره‌وری (Utilization) و میانگین زمان پاسخ (Average Response Time) سیستم به ترتیب چه اعدادی خواهد بود؟
(الگوریتم زمان‌بندی RR با برش زمانی q=1 است و اگر در لحظه t یک job به سیستم وارد شود و در همین لحظه یک job دیگر پردازنده را ترک کرده و به صف آمادگی (Ready Queue) منتقل شود، اولویت با job قبلی موجود در سیستم است که تازه پردازنده را رها کرده است. زمان پاسخ، تأخیر بین ورود هر job و اولین زمان در اختیار گرفتن پردازنده توسط آن job است.)
(مهندسی کامپیوتر – دولتی۹۰)

۱) $\frac{1}{3}$, 85% ۲) $\frac{5}{9}$, 90% ۳) $\frac{5}{9}$, 90% ۴) $\frac{1}{3}$, 90%

۳- سه وظیفه T2, T1 (task) و T3 را در نظر بگیرید که تاکنون $n+1$ بار در سیکل آماده - اجرا - مسدود طی مسیر کرده‌اند. زمان اجرای واقعی سیکل $n+1$ام این وظایف به ترتیب 2، 4 و 6 میلی‌ثانیه و زمان برآورد شده برای اجرای $n+1$ام آنها نیز به ترتیب 4، 6 و 6 میلی‌ثانیه می‌باشد. زمان اجرای واقعی در سیکل $n+2$ام به ترتیب 4 و 3 و 6 است. پس از پایان سیکل $n+1$ام وظیفه‌های T1 و T2 بلافاصله آماده هستند و وظیفه T3 پس از 3 میلی‌ثانیه آماده می‌شود. با

khalilifar.ir

فرض $alpha = 0.5$ در صورتی که بخواهیم این وظایف را با استفاده از الگوریتم‌های **SPN** (یا **SJF**) و **FCFS**، **RR** زمان‌بندی کنیم، میانگین زمان کامل (**Turnaround**) این وظایف فقط در سیکل $n+2$ام چقدر خواهد بود؟ زمان **Context Switch** را در نظر نگیرید.

(مهندسی کامپیوتر – دولتی۹۱)

۱) FCFS 7, SPN 6.7
۲) FCFS 7, SPN 7
۳) SPN 6.7, RR 9
۴) SPN 7.3, RR 9

۴- چهار فرآیند P_1 و P_2 و P_3 و P_4 به ترتیب در زمان‌های **0** و **3** و **7** و **40** وارد می‌شوند و مشخصات آنها مطابق جدول زیر است. اطلاعات هر سطر، منبع مورد نیاز هر فرآیند و همچنین تعداد واحدهای زمانی مورد نیاز را تعیین می‌کنند. برای مثال **CPU 5** بدین معنی است که فرآیند **CPU** را به اندازه **5** واحد نیاز دارد. این سیستم، از زمان‌بندی **RR** استفاده می‌کند و $q = 5$ است. قبل از اینکه هر فرآیند پردازنده را در اختیار بگیرد **1** واحد زمانی سربار دارد. میانگین زمان کل (**Turnaround Time-TR**) و بهره‌وری پردازنده و بهره‌وری **IO** چقدر است؟ (مهندسی IT- دولتی۹۱)

P_1	P_2	P_3	P_4
CPU 5	CPU 2	CPU 8	CPU 9
IO 5	IO 22		IO 2
CPU 2	CPU 2		CPU 1

۱) $TR = 18.75\ ,\ CPU\_UTIL = 95.5\%\quad IO\_UTIL = 82.5\%$
۲) $TR = 20.75\ ,\ CPU\_UTIL = 95.5\%\quad IO\_UTIL = 72.5\%$
۳) $TR = 18.75\ ,\ CPU\_UTIL = 92.5\%\quad IO\_UTIL = 82.5\%$
۴) $TR = 20.75\ ,\ CPU\_UTIL = 92.5\%\quad IO\_UTIL = 72.5\%$

۵- اگر فرآیندهای P_2, P_1, P_0 با هم وارد سیستم شده و به ترتیب به **15**، **12**، **25** ثانیه زمان برای اجرا نیاز داشته باشند، در صورت استفاده از روش صف بازخورد چند سطحی (**Multilevel Feedback Queue**) با سه صف با کوانتوم‌های **4**، **8**، **16** ثانیه و زمان تعویض متن **1** ثانیه، میانگین زمان برگشت فرآیندها (**Turnaround Time**) چند ثانیه خواهد بود؟

(مهندسی کامپیوتر – دولتی۹۲)

۱) 40
۲) 39.33
۳) 39.66
۴) 39

۶- کدام یک از انواع برنامه‌ریزها وظیفه فعال‌سازی و تعلیق فرآیندها را به عهده دارد؟

(مهندسی IT- دولتی۹۲)

۱) Prioritized Scheduler
۲) Long-Term Scheduler
۳) Medium-Term Scheduler
۴) Short-Term Scheduler

---

khalilifar.ir

۷- جدول زیر زمان ورود و زمان پردازش پردازه‌ها را نشان می‌دهد. برای آنکه دو روش برنامه‌ریزی First Come First Served (FCFS) و Highest Response Ratio Next (HRRN) زمان‌بندی‌های یکسانی را تولید نمایند، چه رابطه‌ای باید بین t_B و t_C برقرار باشد؟ برای سادگی از زمان تعویض زمینه بین پردازه‌ها صرف‌نظر کنید. (مهندسی کامپیوتر – دولتی۹۳)

فرآیند	زمان ورود	زمان پردازش
A	0	$t_A = 3$
B	1	t_B
C	2	t_C

۱) $t_B < t_C$ ۲) $t_C < t_B$ ۳) $t_C < 2t_B$ ۴) $t_B < 2t_C$

۸- اگر هر فرآیند در سیستم عامل بتواند در یکی از حالت‌های ایجاد، مسدود، اجرا، پایان‌یافته و آماده اجرا باشد، کدام یک از تغییر حالت‌های مستقیم زیر مجاز نیست؟ (مهندسی IT- دولتی۹۳)

۱) آماده اجرا به اجرا ۲) آماده اجرا به مسدود
۳) اجرا به آماده اجرا ۴) مسدود به آماده اجرا

۹- تحت چه شرایطی زمان‌بند Round-Robin دقیقاً مانند زمان‌بند FCFS عمل می‌نماید؟

(مهندسی IT – دولتی ۹۴)

۱) همه پردازه‌ها مانند هم باشند و ترتیب ورود پردازه‌ها در دو زمان‌بند یکسان باشد.
۲) اولویت همه پردازه‌ها برابر باشد و ترتیب ورود پردازه‌ها در دو زمان‌بند یکسان باشد.
۳) زمان اجرای همه پردازه‌ها برابر باشد و ترتیب ورود پردازه‌ها در دو زمان‌بند یکسان باشد.
۴) زمان اجرای پردازه‌ها کوچکتر از زمان کوانتوم باشد و ترتیب ورود پردازه‌ها در دو زمان‌بند یکسان باشد.

۱۰- پنج پردازه همزمان وارد سیستم می‌شوند که زمان اجرای آنها به صورت زیر است. پردازه‌ها به ترتیب شماره آنها اجرا می‌شوند. اگر زمان‌بندی پردازه‌ها بر اساس الگوریتم Round Robin و Quantum=2 صورت بگیرد، زمان انتظار صف پردازه 2 و 4 چقدر است؟ (مهندسی IT – دولتی ۹۴)

P1:2
P2:6
P3:1
P4:4
P5:3

۱) P2=10 و P4=10 ۲) P2=8 و P4=10
۳) P2=10 و P4=9 ۴) P2=10 و P4=8

---

khalilifar.ir

۱۱- یک سیستم تک‌پردازنده‌ای با صف بازخورد چند سطحی (Multi-level Feedback Queue) را در نظر بگیرید. به سطح اول تکه زمانی معادل 8 میکروثانیه و به سطح دوم، تکه زمانی معادل 16 میکروثانیه داده شده است و سطح سوم، به ترتیب ورود زمان‌بندی شده است (FCFS). فرض کنید 6 کار همگی در زمان صفر وارد سطح اول سیستم شده و زمان اجرای آنها به ترتیب برابر با 4، 7، 12، 20، 25 و 30 میکروثانیه باشد. در این صورت میانگین زمان بازگشت (ATT: Average Turnaround Time) و میانگین زمان انتظار (AWT: Average Waiting Time) کارهای فوق در سیستم به ترتیب از راست به چپ کدام است؟ (مهندسی کامپیوتر- دولتی ۹۵)

۱) 41.8، 55.5 ۲) 51.8، 45.5 ۳) 51.8، 35.5 ۴) 41.8، 25.5

۱۲- در جدول زیر اطلاعات مربوط به پنج پردازه (process) که وارد سیستم می‌شوند داده شده است. اگر سیستم از روش زمان‌بندی نوبتی (RR) با تکه زمانی (کوآنتوم) 40 ثانیه و سربار 10 ثانیه برای تعویض متن فرآیندها (context switch) استفاده کند، میانگین زمان انتظار پردازه‌ها چقدر است؟ (مهندسی IT – دولتی ۹۵)

نام برنامه	A	B	C	D	E
زمان ورود (s)	10	20	250	260	270
مدت زمان لازم برای اجرا (s)	100	50	30	30	40

۱) 48
۲) 56
۳) 106
۴) 108

۱۳- با توجه به جدول، متوسط زمان برگشت (Turnaround Time) و زمان انتظار (Waiting Time) پردازه‌های زیر را به ازای الگوریتم Preemptive Shortest Remaining Job First چه عددی است؟ (مهندسی کامپیوتر – دولتی ۹۶)

پردازه	زمان ورود به سیستم	زمان موردنیاز برای اجرا
P_1	1	10
P_2	3	8
P_3	7	6
P_4	11	3
P_5	15	7

۱) 14 و 6.2 ۲) 16 و 6.2 ۳) 14 و 7.2 ۴) 16 و 7.2

---

khalilifar.ir

پاسخ تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۱- گزینه (۴) صحیح است.

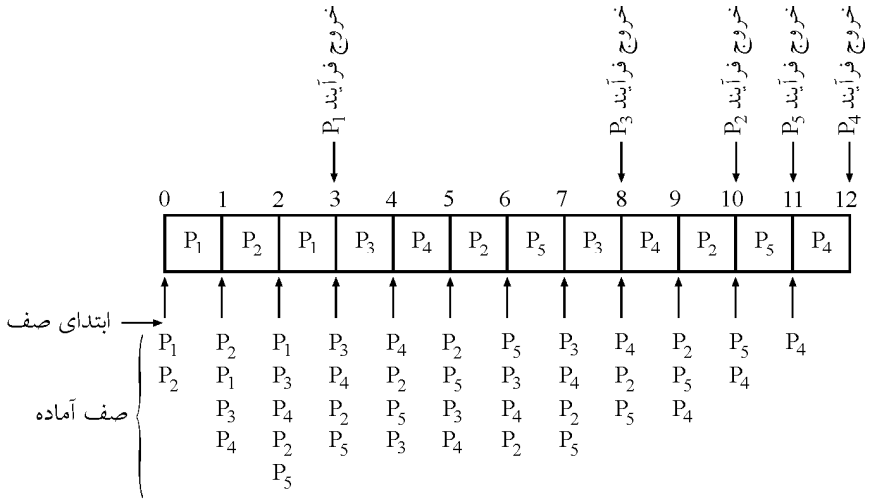
با توجه به مفروضات مطرح شده درصورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	0	2		
P_2	0	3		
P_3	1	2		
P_4	1	3		
P_5	2	2		

توجه: به فرض مطرح شده در صورت سؤال توجه فراوان داشته باشید.

فرض مطرح شده: همیشه بین فرآیندی که در لحظه t، برش زمانی خود را به پایان می رساند و فرآیند ورودی در لحظه t، اولویت با فرآیند قبلی موجود در سیستم است و در شرایط کاملاً یکسان بین دو فرآیند، اولویت با فرآیند با شماره کوچکتر است.

با توجه به برش زمانی q=1 و فرض مطرح شده در صورت سؤال جهت اولویت‌بندی صف الگوریتم RR، نمودار گانت زیر را داریم:



زمان ورود فرآیند- زمان خروج فرآیند= زمان بازگشت فرآیند

P_1 زمان بازگشت = 3-0=3

khalilifar.ir

P_2 زمان بازگشت = 10-0=10

P_3 زمان بازگشت = 8-1=7

P_4 زمان بازگشت = 12-1=11

P_5 زمان بازگشت = 11-2=9

میانگین زمان بازگشت $= \frac{3+10+7+11+9}{5} = \frac{40}{5} = 8$

زمان اجرای فرآیند – زمان بازگشت فرآیند= زمان انتظار فرآیند

P_1 زمان انتظار = 3-2=1

P_2 زمان انتظار = 10-3=7

P_3 زمان انتظار = 7-2=5

P_4 زمان انتظار = 11-3=8

P_5 زمان انتظار = 9-2=7

میانگین زمان انتظار $= \frac{1+7+5+8+7}{5} = \frac{28}{5} = 5.6$

میانگین زمان اجرا $= \frac{2+3+2+3+2}{5} = = \frac{12}{5} = 2.4$

میانگین زمان انتظار + میانگین زمان اجرا= میانگین زمان بازگشت
8=2.4+5.6

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	0	2	1	3
P_2	0	3	7	10
P_3	1	2	5	7
P_4	1	3	8	11
P_5	2	2	7	9

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا
8 5.6 2.4

۲- گزینه (۱) صحیح است.

مفهومی که از فرآیند و job در این تست مطرح شده غلط است. فرآیند دارای job نیست بلکه می‌تواند دارای تعدادی نخ (thread) باشد. طراح محترم، طبق کدام مرجع شما job را بخشی از فرآیند می‌دانید؟ اگر درس سیستم عامل مدنظر شماست، در این درس هریک مدت یکبار زمان‌بند

khalilifar.ir

بلند مدت یک job را از هارد انتخاب می‌کند و در RAM قرار می‌دهد تا درجه چند برنامگی کنترل گردد و این job موجود در RAM، فرآیند نام می‌گیرد.
طراح محترم آیا شما زمان پاسخ را سلیقه‌ای تعریف می‌کنید؟ مگر می‌شود تعاریف اساسی و بنیادی را سلیقه‌ای تعریف نمود؟! در کدام مرجع زمان پاسخ این چنین که شما بیان کرده‌اید، تعریف شده است؟!
جهت اطلاع: تعریف درست زمان پاسخ: از لحظه‌ی ورود فرآیند تا نمایش اولین خروجی.
تعریف درست زمان گردش کار (بازگشت): از لحظه‌ی ورود فرآیند تا نمایش آخرین خروجی (اتمام کار). هیچ مرجعی نگفته است از لحظه‌ی ورود کار تا لحظه‌ی اولین دریافت cpu!
آیا بهتر نبود به جای نقض، دستکاری و خرابکاری تعاریف بنیادی بر اساس مفاهیم اصلی سوال طرح می‌شد؟ حداقل بهتر بود، لحظه‌ی ورود اولین کارها را مشخص می‌کردید.
به هر حال...
اگر فرض کنیم P_1 و P_2 در ابتدا با هم وارد می‌شوند. می‌توان jobهای دو فرآیند و زمان ورود آنها را با توجه به فرضیات سوال به صورت زیر در نظر گرفت:

jobهای فرآیند P_2	زمان ورود	زمان اجرا
J_{21}	0	1
J_{22}	4	1
J_{23}	8	1
J_{24}	12	1
J_{25}	16	1

jobهای فرآیند P_1	زمان ورود	زمان اجرا
J_{11}	0	3
J_{12}	5	3
J_{13}	10	3
J_{14}	15	3

در صورت سوال بیان شده است که اولویت با job قبلی موجود در سیستم است که تازه پردازنده را رها کرده است.
از آنجایی که کارهای مربوط به فرآیندهای P_1 و P_2 به ترتیب به صورت دوره‌ای در هر 5 ثانیه و 4 ثانیه یکبار شروع می‌شود، بنابراین به صورت دوره‌ای هر 20 ثانیه یکبار دو فرآیند P_1 و P_2 همزمان با هم وارد سیستم شده و روند اجرای آنها از سر گرفته می‌شود، پس جهت محاسبه‌ی بهره‌وری پردازنده و میانگین زمان پاسخ کافی است که اجرای کارهای فرآیند P_1 و P_2 را در یک دوره‌ی زمانی از صفر تا 20 ثانیه مورد بررسی قرار دهیم.
برای اینکه ببینیم اولین یا کوچکترین عددی که هم بر 4 و هم بر 5 بخش‌پذیر است، به عبارت دیگر برای محاسبه‌ی کوچکترین مضرب مشترک دو عدد 4 و 5 باید (ک.م.م) را محاسبه نمود:

$$A \cup B = \frac{A \times B}{A \cap B} \qquad \text{ک.م.م} = \frac{4 \times 5}{\text{ب.م.م}}$$

بنابراین داریم:

khalilifar.ir

	1	4	
5	4	(1)	0
4	4		

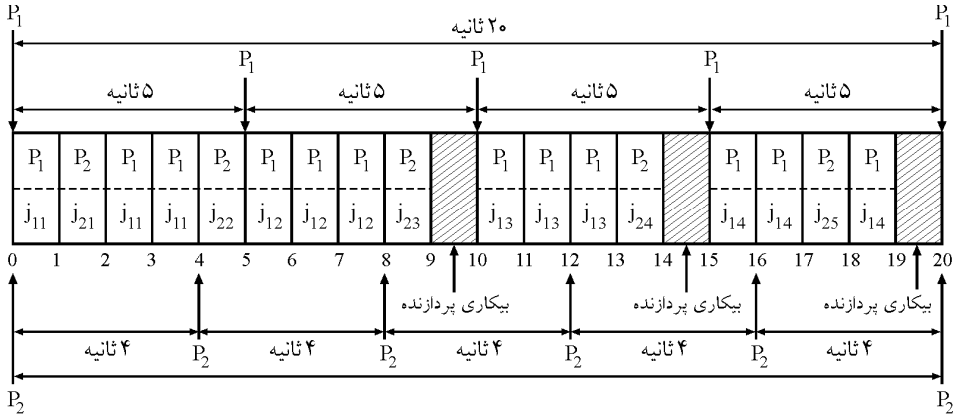
ب.م.م

توجه: ب.م.م، بزرگترین مقسوم علیه مشترک می‌باشد.

$$\text{ک.م.م} = \frac{4\times5}{1} = 20$$

توجه: ب.م.م دو عدد متوالی (پشت سر هم) همواره برابر یک است.

توجه: ک.م.م دو عدد متوالی (پشت سر هم) همواره برابر حاصل ضرب آن دو عدد است.



از آنجا که پردازنده از 20 ثانیه، 17 ثانیه را مشغول اجرا بوده است و 3 ثانیه را مشغول اجـرا نبـوده است، پس بهره‌وری آن معادل 85 درصد است.

$$\text{بهره‌وری پردازنده} = \frac{\text{زمان مفید پردازنده}}{\text{زمان مصرفی پردازنده}} = \frac{17}{20} = 0.85\times100\% = 85\%$$

و یا با توجه به دوره‌ی تناوب و زمان اجرای مطرح شـده بـرای هـر کـار، بهـره‌وری پردازنـده بـه صورت مجموع بهره‌وری کارها و به شرح زیر قابل محاسبه است: (در سیسـتم‌هـای متنـاوب ایـن فرمول کاربرد دارد.)

$$U\,cpu = \sum_{i=1}^{m}\frac{Ci}{Pi} = \frac{C1}{P1} + \frac{C2}{P2} = \frac{3}{5} + \frac{1}{4} = 0.6 + 0.25 = 0.85\times100\% = 85\%$$

C_i : زمان اجرای کار

P_i : دوره تناوب

khalilifar.ir

در دوره‌ی زمانی صفر تا 20 ثانیه، کارهای فرآیند P1، 4 بار و کارهای فرآیند P2، 5 بـار، یعنـی در مجموع 9 کار وارد سیستم شده است.

تعریف زمان پاسخ از نگاه طراح: تأخیر بین ورود هر کار و اولین زمان در اختیار گرفتن پردازنده توسط آن کار.

زمان ورود کار- زمان دریافت پردازنده = زمان پاسخ کار

برای فرآیند P_1 داریم:

j_{11} زمان پاسخ = 0- 0= 0

j_{12} زمان پاسخ = 5 - 5= 0

j_{13} زمان پاسخ = 10 - 10= 0

j_{14} زمان پاسخ = 15 - 15 = 0

برای فرآیند P_2 داریم:

j_{21} زمان پاسخ = 1 - 0= 1

j_{22} زمان پاسخ = 4 - 4 = 0

j_{23} زمان پاسخ = 8 - 8 = 0

j_{24} زمان پاسخ = 13 - 12 = 1

j_{25} زمان پاسخ = 17 - 16 = 1

$$\text{میانگین زمان پاسخ} = \frac{(0+0+0+0)+(1+0+0+1+1)}{9} = \frac{3}{9} = \frac{1}{3}$$

۳- گزینه (۲) صحیح است.

جهت زمان‌بندی فرآیندها در الگوریتم SPN(SJF) مـی‌بایسـت قبـل از اجـرای فرآینـدها، از زمـان اجرای هر یک اطلاع داشته باشیم، در غیر این صورت با استفاده از فرمول تخمینی زیـر (الگـوریتم سالمندی (Aging))، از این الگوریتم می‌توانیم بهره ببریم:

زمان اجرای واقعی پدیده‌ی لحظه‌ای + تخمین قدیم = تخمین جدید

$$S_{n+2} = \alpha S_{n+1} + (1-\alpha)T_{n+1}$$

S_{n+2}: زمان اجرای تخمین نوبت n+2

S_{n+1}: زمان اجرای تخمین نوبت n+1

T_{n+1}: زمان اجرای واقعی نوبت n+1

T_{n+2}: زمان اجرای واقعی نوبت n+2

$\alpha(0<\alpha<1)$: نشان می‌دهد که برای پدیده‌ی لحظه‌ای که الان اتفاق افتاده است، برای T_{n+1} چه وزنـی را نسبت به داده‌های قبلی قائل هستیم.

توجه: هر چه α بیشتر باشد، اهمیت پدیده‌ی لحظه‌ای کمتر می‌شود.

khalilifar.ir

مثال: از شما می‌پرسند معدل کل شما در 6 ترم گذشته چند می‌شود؟

معدل این ترم 16 و میانگین 5 ترم گذشته 15.5

معدل پدیده‌ی لحظه‌ای + تخمین قدیم = تخمین جدید

$$\text{معدل کل} = \frac{5}{6}\times 15.5 + \frac{1}{6}\times 16$$

چون اهمیت ترم ششم، $\frac{1}{6}$ است، بنابراین به این ترم نسبت به 5 ترم گذشته وزن می‌دهیم.

یعنی به پدیده‌ی قدیمی وزن $\frac{5}{6}$ و به پدیدی جدید وزن $\frac{1}{6}$ را می‌دهیم.

بنابراین به کمک فرمول فوق و مقدار $\alpha = \frac{1}{2}$، زمان برآورده شده برای مرحله‌ی n+2 ام را برای هر یک از وظایف به دست می‌آوریم:

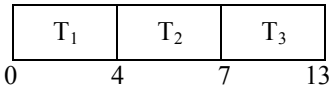
$$S_{n+2} = \alpha S_{n+1} + (1-\alpha)T_{n+1}$$

$$S_{n+2} = \frac{1}{2}S_{n+1} + \frac{1}{2}T_{n+1}$$

وظایف	زمان ورود	S_{n+1}	T_{n+1}	S_{n+2}	T_{n+2}
T_1	0	4	2	$\frac{1}{2}\times 4 + \frac{1}{2}\times 2 = 3$	4
T_2	0	6	4	$\frac{1}{2}\times 6 + \frac{1}{2}\times 4 = 5$	3
T_3	3	6	6	$\frac{1}{2}\times 6 + \frac{1}{2}\times 6 = 6$	6

از ستون S_{n+2} در جدول فوق به این نتیجه می‌رسیم که وظیفه T_1 که دارای زمان اجرای کوچکتر از زمان اجرای وظیفه‌ی T_2 می‌باشد، پس در زمان‌بندی اولویت می‌گیرد و اول وظیفه T_1 باید اجرا گردد، سپس T_2 و در آخر وظیفه‌ی T_3 اجرا شود.

SJF

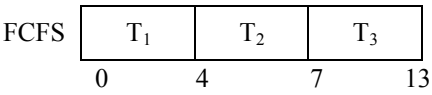


توجه: در نمودار زمانی گانت فوق، مقادیر زمان واقعی T_{n+2} باید در نظر گرفته شود.

$$\text{میانگین زمان کامل (زمان گردش کار)} = \frac{(4-0)+(7-0)+(13-3)}{3} = \frac{21}{3} = 7$$

در پیاده‌سازی الگوریتم RR و FCFS، نیازی به دانستن و یا محاسبه‌ی زمان اجرای فرآیندها قبل از اجرا نمی‌باشد. زیرا FCFS بر اساس خروج به ترتیب ورود می‌باشد و RR، همانند FCFS می‌باشد که می‌چرخد. بنابراین داریم:

khalilifar.ir

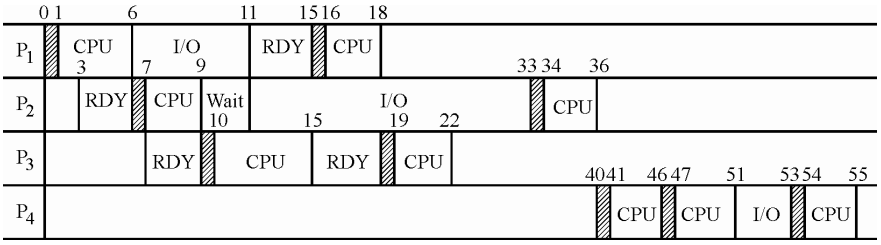


$$\text{میانگین زمان کامل (زمان گردش کار)} = \frac{(4-0)+(7-0)+(13-3)}{3} = \frac{21}{3} = 7$$

توجه: در صورت مسأله برش زمانی ذکر نشده است. بنابراین امکان محاسبه‌ی میانگین زمان کامل به کمک الگوریتم RR میسر نمی‌باشد.

۴- گزینه ( ) صحیح است.

تخصیص منابع به هر فرآیند را مطابق نمودار ذیل انجام می‌دهیم:



توجه: منظور از RDY در نمودار فوق در صف آماده (Ready) می‌باشد.

$$\text{میانگین زمان کامل} = \frac{1}{4}\left[(18-0)+(36-3)+(22-7)+(55-40)\right] = 20.25 \quad \text{واحد زمانی}$$

حاصل جمع زمان‌های اجرای فرآیندها:

$$\underbrace{(CPU5+CPU2)}_{P_1}+\underbrace{(CPU2+CPU2)}_{P_2}+\underbrace{(CPU8)}_{P_3}+\underbrace{(CPU9+CPU1)}_{P_4}=29$$

$$\text{حاصل جمع زمان‌های ورودی و خروجی فرآیندها} = \underbrace{(IO5)}_{P_1}+\underbrace{(IO22)}_{P_2}+\underbrace{(IO2)}_{P_4}=29$$

$$\text{بهره‌وری ورودی و خروجی} = \frac{\text{حاصل جمع زمان‌های ورودی و خروجی فرآیندها}}{\text{زمان کل}} \times 100$$

$$= \frac{29}{53}\times 100 = 54.71\%$$

$$\text{بهره‌وری پردازنده} = \frac{\text{حاصل جمع زمان‌های اجرای فرآیندها}}{\text{زمان کل}} \times 100 = \frac{29}{55}\times 100 = 52.72\%$$

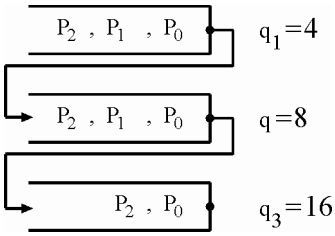
توجه: سازمان سنجش آموزش کشور در کلید اولیه خود ابتدا گزینه چهارم را به عنوان پاسخ اعلام نمود، سپس در کلید نهایی نظر خود را عوض کرد و کلاً تست را حذف نمود، که عمل درستی را انجام داده است که البته عمل درست‌تر آن است که سؤال از ابتدا، درست طرح گردد.

khalilifar.ir

۵- گزینه ( ) صحیح است.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

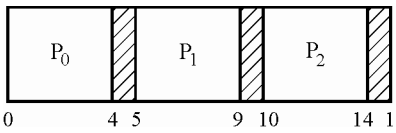
فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_0	0	15		
P_1	0	12		
P_2	0	25		



ابتدا فرآیندها در صف اول با تکه زمانی (کوانتوم زمانی) معادل 4 ثانیه قرار می‌گیرند. و اگر در مدت 4 ثانیه تمام نشوند به صف دوم با تکه زمانی (کوانتوم زمانی) معادل 8 ثانیه منتقل می‌گردند. و اگر باز هم در مدت 8 ثانیه تمام نشوند به صف سوم با تکه زمانی (کوانتوم زمانی) معادل 16 ثانیه منتقل می‌گردند، تا بالاخره تمام شوند.

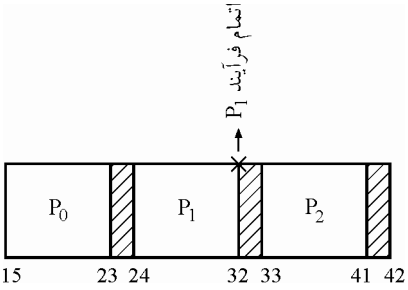
نمودار گانت این فرآیندها در هر صف به صورت زیر است:

صف اول: TS=4s



در صف اول، از فرآیندهای P_0، P_1، P_2 به ترتیب، 11، 8 و 21 ثانیه باقی می‌ماند. بنابراین این فرآیندها به صف دوم با تکه زمانی معادل 8 ثانیه منتقل می‌شوند.

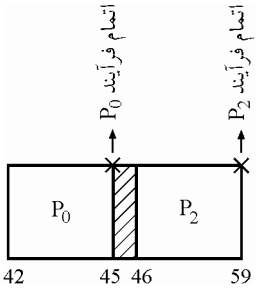
صف دوم: TS=8s



khalilifar.ir

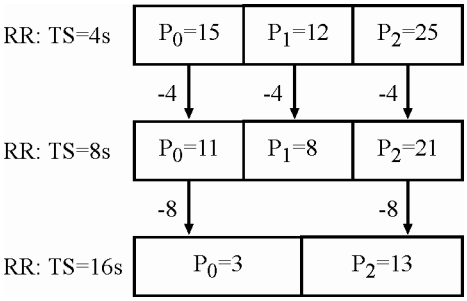
در صف دوم، فرآیند P_1 در لحظه 32 خاتمه می‌یابد و از فرآینـدهای P_0 و P_2 بـه ترتیـب، 3 و 13 ثانیه باقی می‌ماند. بنابراین این فرآیندها به صف سوم با تکه زمانی معادل 16 ثانیه منتقل می‌شـوند، تا بالاخره تمام شوند.

صف سوم: TS=16s

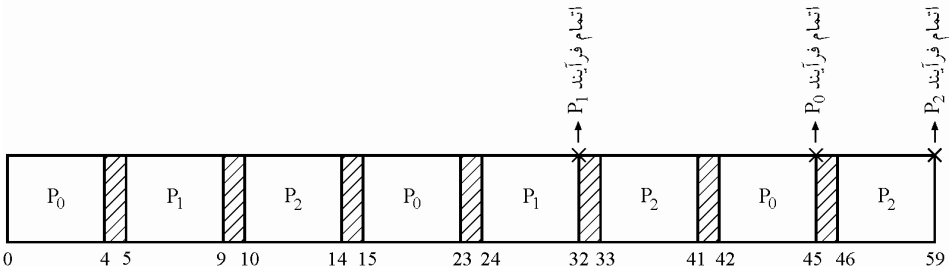


در صف سوم، فرآیندهای P_0 و P_2 در لحظات 45 و 59 خاتمه می‌یابند.

مدل‌سازی صف‌های فوق به صورت زیر است:



نمودار گانت کلی صف‌های فوق به صورت زیر است:



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_0 زمان بازگشت $= 45 - 0 = 45$

P_1 زمان بازگشت $= 32 - 0 = 32$

khalilifar.ir

P_2 زمان بازگشت $= 59 - 0 = 59$

میانگین زمان بازگشت $= \frac{45+32+59}{3} = \frac{136}{3} = 45.33$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_0 زمان انتظار $= 45 - 15 = 30$

P_1 زمان انتظار $= 32 - 12 = 20$

P_2 زمان انتظار $= 59 - 25 = 34$

میانگین زمان انتظار $= \frac{30+20+34}{3} = \frac{84}{3} = 28$

میانگین زمان اجرا $= \frac{15+12+25}{3} = \frac{52}{3} = 17.33$

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

$45.33 = 17.33 + 28$

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_0	0	15	30	45
P_1	0	12	20	32
P_2	0	25	34	59

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا
45.33 = 28 + 17.33

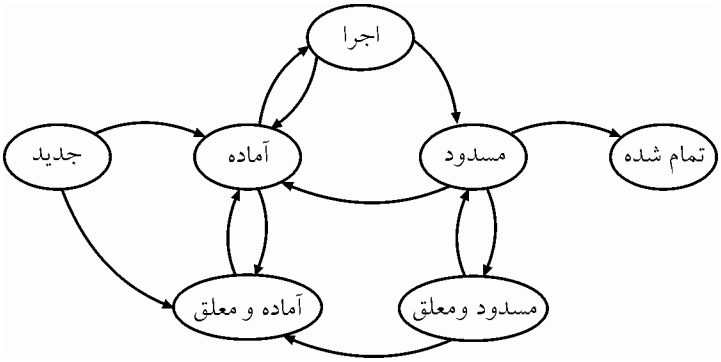
توجه: سازمان سنجش آموزش کشور در کلید اولیه خود ابتدا گزینه دوم را به عنوان پاسخ اعلام نمود، سپس در کلید نهایی نظر خود را عوض کرد و کلاً تست را حذف نمود، که عمل درستی را انجام داده است که البته عمل درست‌تر آن است که سؤال از ابتدا، درست طرح گردد.

۶- گزینه (۳) صحیح است.

بیشتر سیستم عامل‌ها حالت‌های فرآیند را فقط محدود به پنج حالت جدید، آماده، اجرا، مسدود و پایان یافته نمی‌کنند، بلکه یک موقعیت جدید با عنوان معلق (suspend) نیز برای فرآیند در نظر می‌گیرند. در حالت معلق فرآیند از حافظه به روی دیسک بیرون رانده شده است. به عنوان مثال فرض کنید همه‌ی فرآیندهای موجود در سیستم به حالت مسدود نقل مکان کنند (و همگی مثلاً

khalilifar.ir

منتظر اتمام ورودی و خروجی مختص به خود هستند)، بنابراین هیچ فرآیندی در صف آماده قرار ندارد. در این حالت سیستم عامل جهت بیکار نماندن cpu، می‌تواند یک فرآیند را که در حالت مسدود است، از حافظه به دیسک منتقل کرده و یک فرآیند جدید را وارد صف آماده کند. به شکل زیر دقت کنید:



زمان‌بند میان مدت (Middle Term Scheduler) بنا به دلایلی ممکن است فرآیندهایی را از روی حافظه، به دیسک و بالعکس منتقل کند. در واقع وظیفه‌ی این زمان‌بند جابه‌جا کردن فرآیندها بین حافظه و دیسک می‌باشد. به این کار مبادله (Swapping) گفته می‌شود.

بنابراین زمان‌بند میان مدت (سطح میانی) که همان زمان‌بند حافظه یا swapper است، وظیفه suspend کردن (تعلیق) و دوباره فعال‌سازی (آماده کردن) فرآیندها را برعهده دارد.

۷- گزینه (۴) صحیح است.

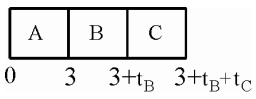
فرآیند	زمان ورود	زمان پردازش
A	0	$t_A = 3$
B	1	t_B
C	2	t_C

در الگوریتم FCFS

در این روش، فرآیندها به همان ترتیب ورود به سیستم، در یک صف قرار گرفته و از ابتدای صف به ترتیب، پردازنده را در اختیار می‌گیرند.

توجه: FCFS یک الگوریتم **انحصاری** است.

با توجه به مفروضات مسأله، نمودار گانت زیر را داریم:



khalilifar.ir

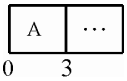
در الگوریتم HRRN

برای تعیین اولویت یک فرآیند در الگوریتم HRRN از فرمول زیر استفاده می‌شود:

$$\text{اولویت} = \frac{\text{زمان انتظار + زمان اجرا}}{\text{زمان اجرا}} = \frac{\text{زمان انتظار}}{\text{زمان اجرا}} + 1$$

نکته: HRRN یک الگوریتم **انحصاری** است.

در این الگوریتم در زمان صفر فقط فرآیند A قرار دارد که به شکل انحصاری اجرا می‌گردد.



اما در زمان 3 فرآیندهای B و C در صف آماده قرار دارند. انتظار هر یک از فرآیندهای B و C تا لحظه 3 به صورت زیر است:

$$\text{انتظار B} = 3 - 1 = 2$$

$$\text{انتظار C} = 3 - 2 = 1$$

در ادامه براساس رابطه اولویت داریم:

$$\text{اولویت (B)} = \frac{2}{t_B} + 1$$

$$\text{اولویت (C)} = \frac{1}{t_C} + 1$$

حال اگر مطابق فرض مسأله قرار باشد، زمان‌بندی‌های دو الگوریتم FCFS و HRRN یکسان باشد. پس در الگوریتم HRRN هم باید اول فرآیند A اجرا گردد که بدیهی است و اجرا می‌گردد، اما بعد از اجرای فرآیند A، در لحظه 3، دو فرآیند B و C در الگوریتم HRRN شانس اجرا دارند، ولی از آنجا که باید این الگوریتم مانند FCFS باشد، پس باید اول فرآیند B اجرا گردد و بعد، فرآیند C و این زمانی رخ می‌دهد که اولویت فرآیند B از اولویت فرآیند C بیشتر باشد، یعنی داشته باشیم:

$$\text{اولویت (B)} > \text{اولویت (C)}$$

$$\frac{2}{t_B} + 1 > \frac{1}{t_C} + 1 \rightarrow$$

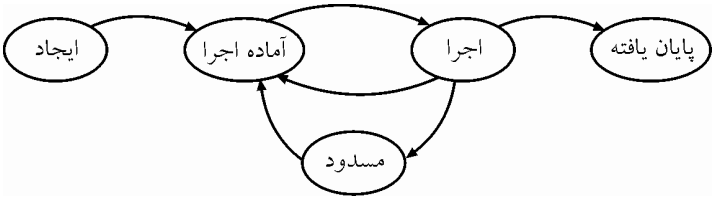
$$\frac{2}{t_B} > \frac{1}{t_C} \rightarrow t_B < 2t_C$$

تست جالبی بود.

۸- گزینه (۲) صحیح است.

تغییر حالات ممکن برای یک فرآیند مطابق حالت‌های ایجاد، مسدود، اجرا، پایان یافته و آماده اجرا مطابق شکل زیر می‌باشد:

khalilifar.ir



بنابراین واضح است که تغییر حالتی از آماده اجرا به مسدود وجود ندارد، زیرا ابتدا باید به حالت اجرا برویم تا بتوانیم درخواستی بدهیم و بعد مسدود شویم.

۹- گزینه (۴) صحیح است.

الگوریتم FCFS (First Come First Served)

این الگوریتم ساده‌ترین الگوریتم زمان‌بندی پردازنده است. در این روش فرآیندها با همان ترتیب ورود به سیستم، در یک صف قرار گرفته و از ابتدای صف به ترتیب، پردازنده را در اختیار می‌گیرند.

توجه: این الگوریتم، **FIFO** (First In First Out) نیز نامیده می‌شود.

توجه: FCFS، یک الگوریتم **انحصاری** (Non Preemtive) است.

توجه: الگوریتم FIFO، مشکل **قحطی‌زدگی** ندارد.

الگوریتم RR (Round Robin)

الگوریتم RR (نوبت چرخشی) یکی از پرکابردترین الگوریتم‌ها در سیستم‌های اشتراک زمانی است. درباره نحوه عملکرد آن می‌توان گفت این الگوریتم نسخه غیرانحصاری (Preemptive) الگوریتم FCFS می‌باشد. در این الگوریتم، زمان پردازنده را به برش‌های زمانی کوتاه (Time Slice) تقسیم می‌کنیم. همانند الگوریتم FCFS، فرآیندهایی که به سیستم تحویل داده می‌شوند به انتهای یک صف وارد می‌شوند. سپس پردازنده از ابتدای صف شروع کرده و به هر فرآیند حداکثر به اندازه یک برش زمانی سرویس می‌دهد.

در واقع پس از آنکه برش زمانی یک فرآیند به پایان رسید، پردازنده آن فرآیند را رها کرده و به سراغ فرآیند بعدی موجود در صف می‌رود. این عمل آنقدر تکرار می‌شود تا پردازنده به انتهای صف فرآیندهای آماده برسد. به عبارت دیگر فرآیندها در یک صف دایره‌ای شکل سازماندهی می‌شوند و پردازنده به صورت دوار در این صف حرکت کرده و به هر فرآیند فقط به اندازه حداکثر یک برش زمانی سرویس می‌دهد.

توجه: به برش زمانی، کوانتوم زمانی (Quantum Time) نیز می‌گویند.

توجه: الگوریتم RR مشکل **قحطی‌زدگی** ندارد.

حال اگر ترتیب ورود پردازه‌ها (فرآیند) یکسان در نظر گرفته شود و زمان اجرای پردازه‌ها کوچکتر از زمان کوانتوم باشد، آنگاه این شرایط فراهم می‌گردد تا زمان‌بند RR دقیقاً مانند زمان‌بند FCFS

khalilifar.ir

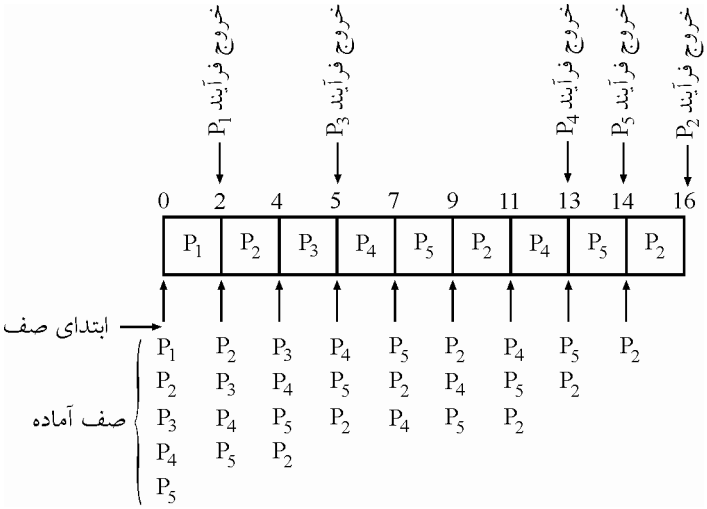
عمل نماید. چون در چنین شرایطی در زمان‌بند RR هر فرآیند قبل از پایان کوانتوم خاتمه می‌یابد و این یعنی FIFO.

۱۰- گزینه (۳) صحیح است.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	0	2		
P_2	0	6		
P_3	0	1		
P_4	0	4		
P_5	0	3		

با توجه به برش زمانی $quantum = 2$، نمودار گانت زیر را داریم:



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 2 - 0 = 2$

P_2 زمان بازگشت $= 16 - 0 = 16$

P_3 زمان بازگشت $= 5 - 0 = 5$

P_4 زمان بازگشت $= 13 - 0 = 13$

P_5 زمان بازگشت $= 14 - 0 = 14$

khalilifar.ir

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 2 - 2 = 0$

P_2 زمان انتظار $= 16 - 6 = 10$

P_3 زمان انتظار $= 5 - 1 = 4$

P_4 زمان انتظار $= 13 - 4 = 9$

P_5 زمان انتظار $= 14 - 3 = 11$

بنابراین مطابق خواسته سوال، زمان انتظار پردازه 2 و 4 به ترتیب برابر $P_2 = 10$ و $P_4 = 9$ می‌باشد. بنابراین گزینه سوم پاسخ سؤال است.

۱۱- گزینه (۳) صحیح است.

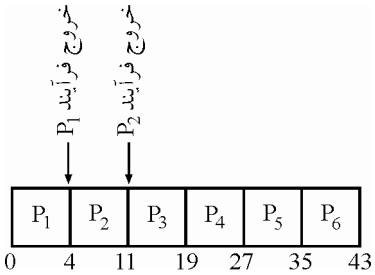
با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	0	4		
P_2	0	7		
P_3	0	12		
P_4	0	20		
P_5	0	25		
P_6	0	30		

ابتدا فرآیندها در صف اول با تکه زمانی معادل 8 میکروثانیه قرار می‌گیرند. و اگر در مدت 8 میکروثانیه تمام نشوند به صف دوم با تکه زمانی معادل 16 میکروثانیه منتقل می‌گردند. و اگر باز هم در مدت 16 میکروثانیه تمام نشوند به صف سوم با الگوریتم FCFS منتقل می‌گردند، تا بالاخره تمام شوند.

نمودار گانت این فرآیندها در هر صف به صورت زیر است:

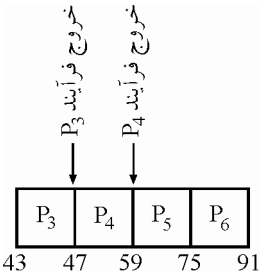
صف اول: $TS = 8\mu s$



khalilifar.ir

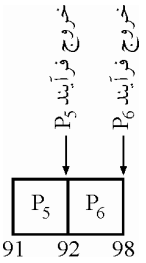
در صف اول، فرآیندهای P_1 و P_2 در لحظات 4 و 11 خاتمه می‌یابند و از فرآیندهای P_3 تا P_6 به ترتیب، 4، 12، 17 و 22 میکروثانیه باقی می‌ماند. بنابراین این فرآیندها به صف دوم با تکه زمانی معادل 16 میکروثانیه منتقل می‌شوند.

صف دوم: $TS = 16\mu s$



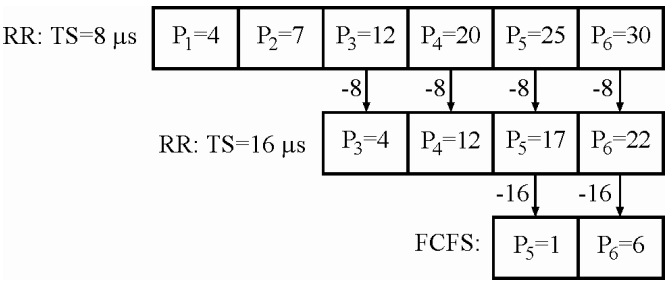
در صف دوم، فرآیندهای P_3 و P_4 در لحظات 47 و 59 خاتمه می‌یابند و از فرآیندهای P_5 و P_6 به ترتیب، 1 و 6 میکروثانیه باقی می‌ماند. بنابراین این فرآیندها به صف سوم با الگوریتم FCFS منتقل می‌شوند.

صف سوم: FCFS



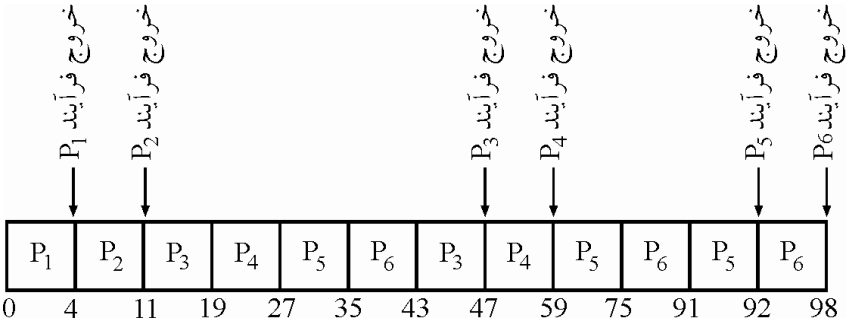
در صف سوم، فرآیندهای P_5 و P_6 در لحظات 92 و 98 خاتمه می‌یابند.

مدل‌سازی صف‌های فوق به صورت زیر است:



khalilifar.ir

نمودار گانت کلی صف‌های فوق به صورت زیر است:



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 4 - 0 = 4$

P_2 زمان بازگشت $= 11 - 0 = 11$

P_3 زمان بازگشت $= 47 - 0 = 47$

P_4 زمان بازگشت $= 59 - 0 = 59$

P_5 زمان بازگشت $= 92 - 0 = 92$

P_6 زمان بازگشت $= 98 - 0 = 98$

میانگین زمان بازگشت $= \text{ATT} = \frac{4+11+47+59+92+98}{6} = \frac{311}{6} = 51.83$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 4 - 4 = 0$

P_2 زمان انتظار $= 11 - 7 = 4$

P_3 زمان انتظار $= 47 - 12 = 35$

P_4 زمان انتظار $= 59 - 20 = 39$

P_5 زمان انتظار $= 92 - 25 = 67$

P_6 زمان انتظار $= 98 - 30 = 68$

میانگین زمان انتظار $= \text{AWT} = \frac{0+4+35+39+67+68}{6} = \frac{213}{6} = 35.5$

میانگین زمان اجرا $= \text{AST} = \frac{4+7+12+20+25+30}{6} = \frac{98}{6} = 16.33$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

khalilifar.ir

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

51.83 = 16.33 + 35.5

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد، که فقط در گزینه سوم و چهارم این مورد رعایت شده است. بنابراین گزینه اول و دوم نادرست است.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است، بنابراین از این نگاه نیز گزینه اول نادرست است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	0	4	0	4
P_2	0	7	4	11
P_3	0	12	35	47
P_4	0	20	39	59
P_5	0	25	67	92
P_6	0	30	68	98

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا
51.83 35.5 16.33

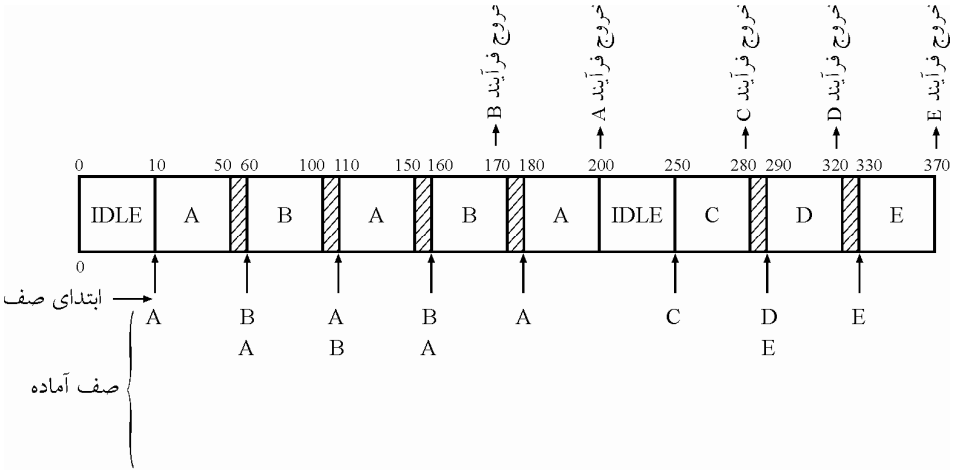
۱۲- گزینه (۲) صحیح است.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
A	10	100		
B	20	50		
C	250	30		
D	260	30		
E	270	40		

با توجه به برش زمانی quantum = 40 و زمان سربار تعویض متن فرآیندها context switch = 10 نمودار گانت زیر را داریم:

khalilifar.ir



زمان ورود فرآیند- زمان خروج فرآیند= زمان بازگشت فرآیند

A زمان بازگشت = 200–10=190

B زمان بازگشت =170–20=150

C زمان بازگشت =280–250=30

D زمان بازگشت =320–260=60

E زمان بازگشت =370–270=100

$$\text{میانگین زمان بازگشت} = \frac{190+150+30+60+100}{5} = \frac{530}{5} = 106$$

زمان اجرای فرآیند - زمان بازگشت فرآیند= زمان انتظار فرآیند

A زمان انتظار =190–100=90

B زمان انتظار =150–50=100

C زمان انتظار =30–30=0

D زمان انتظار =60–30=30

E زمان انتظار =100–40=60

$$\text{میانگین زمان انتظار} = \frac{90+100+0+30+60}{5} = \frac{280}{5} = 56$$

$$\text{میانگین زمان اجرا} = \frac{100+50+30+30+40}{5} = \frac{250}{5} = 50$$

میانگین زمان انتظار + میانگین زمان اجرا= میانگین زمان بازگشت

106 = 50+56

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

khalilifar.ir

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
A	10	100	90	190
B	20	50	100	150
C	250	30	0	30
D	260	30	30	60
E	270	40	60	100

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا
106 56 50

۱۳- گزینه (۳) صحیح است.

الگوریتم SJF (Shortest Job First)

در این روش ابتدا کاری برای اجرا انتخاب می‌شود که از همه کوتاهتر باشد (زمان اجرای کمتری داشته باشد).

توجه: این الگوریتم، **SPN** (Shortest Process Next) و **SPT** (Shortest Process Time) نیز نامیده می‌شود.

توجه: SJF یک الگوریتم **انحصاری** (Non Preemptive) است.

توجه: یک نقص عمده الگوریتمSJF این است که ممکن است باعث قحطی‌زدگی فرآیندهای طولانی شود. به این ترتیب که اگر همواره تعدادی فرآیند کوچک وارد سیستم شوند، اجرای فرآیندهای بزرگ به طور متناوب به تعویق می‌افتد. این روال حتی می‌تواند تا بینهایت ادامه یابد و هیچگاه نوبت به فرآیندهای بزرگ نرسد!!!!!

توجه: در این روش اگر دو فرآیند مدت زمان اجرای برابر داشته باشند، بر اساس FCFS زمان‌بندی می‌شوند.

توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسخ و میانگین زمان گردش کار (بازگشت) فرآیندهاست.

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآیندها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجرای فرآیندها را فقط حدس زده و به طور تقریبی بدست آورد.

الگوریتم SRT (Shortest Remaining Time)

این الگوریتم نسخه **غیرانحصاری** (Preemptive) الگوریتم SJF است. در این الگوریتم اگر حین اجرای یک فرآیند، فرآیندی وارد شود که زمان اجرای کوتاه‌تری داشته باشد، پردازنده را در اختیار می‌گیرد.

khalilifar.ir

توجه: این الگوریتم، **SRPT** (Shortest Remaining Processing Time)،
SRTF (Shortest Remaining Time First) و
SRTN (Shortest Remaining Time Next) نیز نامیده می‌شود.

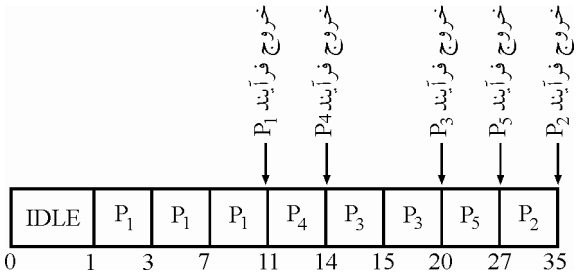
توجه: اگر لحظه ورود همه فرآیندها یکی باشد، الگوریتم SRT مشابه SJF عمل می‌کند.

توجه: در الگوریتم **SRT** نیز همانند الگوریتم SJF، احتمال وقوع **قحطی‌زدگی برای کارهای بزرگ** وجود دارد.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	1	10		
P_2	3	8		
P_3	7	6		
P_4	11	3		
P_5	15	7		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 11 - 1 = 10$

P_2 زمان بازگشت $= 35 - 3 = 32$

P_3 زمان بازگشت $= 20 - 7 = 13$

P_4 زمان بازگشت $= 14 - 11 = 3$

P_5 زمان بازگشت $= 27 - 15 = 12$

میانگین زمان بازگشت $= \text{ATT} = \frac{10+32+13+3+12}{5} = \frac{70}{5} = 14$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 10 - 10 = 0$

khalilifar.ir

P_2 زمان انتظار $= 32 - 8 = 24$

P_3 زمان انتظار $= 13 - 6 = 7$

P_4 زمان انتظار $= 3 - 3 = 0$

P_5 زمان انتظار $= 12 - 7 = 5$

میانگین زمان انتظار $= AWT = \frac{0+24+7+0+5}{5} = \frac{36}{5} = 7.2$

میانگین زمان اجرا $= AST = \frac{10+8+6+3+7}{5} = \frac{34}{5} = 6.8$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

14= 6.8 + 7.2

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد، که فقط در گزینه سوم این مورد رعایت شده است. بنابراین گزینه‌های اول، دوم و چهارم پاسخ سؤال نیستند.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است، بنابراین از این نگاه همه گزینه‌ها درست هستند، که در اینجا این نگاه کارآمد نیست.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	1	10	0	10
P_2	3	8	24	32
P_3	7	6	7	13
P_4	11	3	0	3
P_5	15	7	5	12

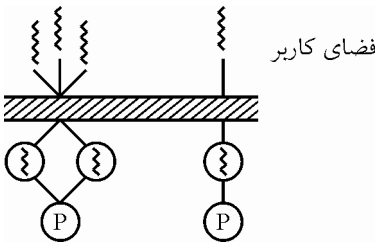
میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

14 7.2 6.8

khalilifar.ir

تست‌های فصل سوم: مدیریت نخ

۱- در یک سیستم کامپیوتری نحوه استفاده از نخ (Thread) در لایه کاربر و در لایه کرنل به صورت مقابل نشان داده شده است. کدام عبارت صحیح است؟ (مهندسی کامپیوتر – دولتی۸۹)



فرآیند Ⓟ

نخ در لایه کاربر ≀

نخ در لایه کرنل ⦚

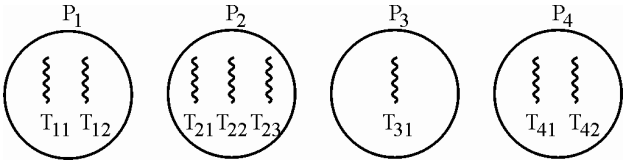
۱) فراخوانی‌های سیستمی از نوع مسدود (Blocking) با تأمین همروندی حمایت می‌شوند و برای فراخوانی‌های سیستمی از نوع غیرمسدود (Non- Blocking) درجه همروندی پایین‌تر است.

۲) فراخوانی‌های سیستمی از نوع مسدود (Blocking) بدون تأمین همروندی اجرا می‌شوند و فراخوانی‌های سیستمی از نوع غیرمسدود (Non- Blocking) همروندی را تأمین می‌کنند.

۳) فراخوانی‌های سیستمی از نوع مسدود (Blocking)، با تأمین همروندی حمایت می‌شوند و برای فراخوانی‌های سیستمی از نوع غیرمسدود (Non- Blocking) درجه همروندی بالاتر است.

۴) فراخوانی‌های سیستمی از نوع مسدود (Blocking) بدون تأمین همروندی اجرا می‌شوند و فراخوانی‌های سیستمی از نوع غیرمسدود (Non- Blocking) نیز با مشکل همروندی مواجه هستند.

۲- سیستمی شامل 4 فرآیند است که داخل هر فرآیند می‌تواند بیش از یک نخ (Thread) اجرایی وجود داشته باشد. در لحظه صفر وضعیت این 4 فرآیند و تعداد نخ‌های اجرایی آنها در شکل و جدول زیر مشخص شده است؟



زمان لازم برای اجرای نخ‌ها

فرآیند	P_1		P_2			P_3	P_4	
نخ	T_{11}	T_{12}	T_{21}	T_{22}	T_{23}	T_{31}	T_{41}	T_{42}
زمان اجرا (ms)	12	9	7	8	8	9	7	8

khalilifar.ir

سهم زمانی هر فرآیند 10ms است و از روش نوبت گردشی (RR) استفاده می‌شود. همچنین داخل هر فرآیند از روش FIFO برای تعویض نخ‌ها استفاده می‌شود و تا زمان اجرایی یک نخ تمام نشده نوبت به نخ بعدی نمی‌رسد. برای تعویض فرآیند 1ms و برای تعویض نخ در داخل فرآیند 0.5ms زمان لازم است. زمان پایان نخ‌های T_{12} و T_{22} چقدر است؟ (مهندسی IT- دولتی۸۹)

۱) 73ms و 59.5ms
۲) 70ms و 45.5ms
۳) 45.5ms و 70.5ms
۴) 18ms و 29ms

۳- در این سوال تفاوت‌های مابین تغییر متن (context switch) در فرآیندها و تغییر متن در نخ‌های یک فرآیند بررسی می‌شوند. کدام گزینه صحیح است؟ (مهندسی کامپیوتر – دولتی۹۰)

۱) تغییر متن در فرآیندها موجب تغییر ثبات‌های اجرایی برنامه می‌شود و سیاست‌های حفاظتی را تغییر می‌دهد.
تغییر متن در نخ‌ها موجب تغییر اشاره‌گر پشته (SP) و پاک شدن TLB می‌شود.

۲) تغییر متن در فرآیندها موجب تغییر اشاره‌گر پشته (SP) و پاک شدن TLB می‌شود.
تغییر متن در نخ‌ها موجب تخصیص سهم زمانی تازه می‌شود و برنامه شمار (PC) تغییر می‌یابد.

۳) تغییر متن در فرآیندها موجب تغییر ثبات‌های اجرایی برنامه می‌شود و با TLB کاری ندارد.
تغییر متن در نخ‌ها موجب تغییر اشاره‌گر پشته (SP) و پاک شدن TLB می‌شود.

۴) تغییر متن در فرآیندها موجب تغییر ثبات‌های اجرایی برنامه و تغییر برنامه شماره (PC) می‌شود.
تغییر متن در نخ‌ها، ثبات‌ها و جداول مدیریت حافظه را تغییر نمی‌دهد و اشاره‌گر پشته (SP) را تغییر می‌دهد.

۴- در رابطه با مدیریت نخ (thread) کدام یک از جملات زیر صحیح است؟ (مهندسی IT- دولتی۹۱)
(توجه: LWP مخفف Light Weight Process است و محیط اجرای نخ می‌باشد)

۱) تغییر متن (Context switch) ما بین نخ‌های شامل: (1) ذخیره ثبات‌های پردازنده مربوط به نخ بیرون رونده و بار کردن ثبات‌های پردازنده مربوط به نخ داخل شونده و (2) ذخیره لیست فایل‌های باز شده توسط نخ است.

۲) یک نخ عادی در طول حیات خود ممکن است در LWP های متفاوتی بخش‌هایی از اجرای خود را بگذارند.

۳) به ازای هر نخ سیستم عامل یک LWP ایجاد می‌کند و نخ تا پایان حیات خود به آن LWP منتسب است، زمان‌بندی نخ می‌تواند توسط سیستم عامل یا کاربر انجام پذیرد.

۴) نخ مستقیماً زیر نظر سیستم عامل اجرا می‌شود و مدیریت آن نمی‌تواند در سطح کاربر باشد.

khalilifar.ir

۵- کدام گزینه درباره مدل‌های چند نخی درست نیست؟ (مهندسی کامپیوتر – دولتی۹۲)

۱) مدل‌های یک به یک و چند به چند توانایی بکارگیری بهتر از پردازنده‌ها / هسته‌ها را دارند.
۲) مدل چند به یک نسبت به مدل یک به یک از کارایی کمتری برخودار است.
۳) در مدل‌های یک به یک و چند به یک امکان همزمانی کامل بین نخ‌ها وجود دارد.
۴) مدل یک به یک نسبت به مدل چند به یک از همزمانی بیشتری برخوردار است.

۶- اگر در یک پردازنده دو هسته‌ای از زمان‌های انتظار نخ‌ها (Threads) برای عملیات حافظه (Memory Stall) جهت سیکل محاسباتی سایر نخ‌ها استفاده شود. بهترین ترکیب تخصیص چهار نخ زیر به هسته‌ها برای کاهش زمان تأخیر چگونه خواهد بود؟ (مهندسی IT- دولتی۹۲)

T_1	C 4	M 3	C 7	M 3
T_2	C 5	M 2	C 6	M 2
T_3	C 2	M 6	C 4	M 6
T_4	C 2	M 2	C 2	M 2

توضیح: سیکل محاسباتی هر نخ با C و سیکل حافظه با M نشان داده شده و مدت زمان هر سیکل به ثانیه، زیر آن نشان داده شده است. پردازنده، اجرای سیکل C را تا زمانی ادامه می‌دهد که به سیکل M برسد و پس از آن سیکل C نخ دیگری را اجرا می‌نماید.

۱) T_1 و T_2 روی هسته اول، بقیه روی هسته دوم
۲) T_1 و T_3 روی هسته اول، بقیه روی هسته دوم
۳) T_1 و T_4 روی هسته اول، بقیه روی هسته دوم
۴) T_2 و T_4 روی هسته اول، بقیه روی هسته دوم

۷- برای یک نخ (Thread) کدام‌یک از تغییر وضعیت‌ها، امکان‌پذیر است؟ (مهندسی IT – دولتی ۹۶)

۱) رفتن از حالت آماده به خاتمه
۲) رفتن از حالت آماده به انتظار
۳) رفتن از حالت آماده اجرا به انتظار
۴) رفتن از حالت آماده انتظار به اجرا

---

khalilifar.ir

پاسخ تست‌های فصل سوم: مدیریت نخ

۱- گزینه (۳) صحیح است.

به طور کلی مدیریت و زمان‌بندی نخ‌ها به سه روش زیر انجام می‌گیرد:

۱- روش سطح کاربر یا مدل چند به یک (many to one)

۲- روش سطح هسته یا مدل یک به یک (one to one)

۳- روش ترکیبی (سطح کاربر و هسته) یا مدل چند به چند (many to many)

با توجه به شکل مطرح شده در صورت سؤال، واضح است که در این سیستم کامپیوتری مدیریت و زمان‌بندی نخ‌ها به روش ترکیبی انجام می‌گردد.

در روش ترکیبی به دلیل آنکه نخ‌های کاندید فرآیندهای سطح کاربر، به هسته سیستم عامل معرفی می‌گردند و هسته سیستم عامل از وجود نخ‌های کاندید فرآیندهای سطح کاربر آگاه است، اگر یک نخ کاندید موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدود کننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم خانواده با آن نخ مسدود شده، همچنان می‌توانند از پردازنده بهره ببرند. این راه کار، منجر به امکان هم‌روندی (در سیستم‌های تک پردازنده‌ای) و امکان توازی (در سیستم‌های چند پردازنده‌ای) میان فرآیندهای مختلف کاربر و یا میان نخ‌های داخل یک فرآیند کاربر می‌شود. البته اگر نخی مسدود نگردد، درجه هم‌روندی و توازی بالاتر هم خواهد رفت، زیرا در این صورت همه نخ‌ها بطور هم‌روند یا موازی در حال حرکت هستند. مانند یک تیم فوتبال ۱۱ نفره که اگر بازیکنی مرتکب خطا گردد، هم‌روندی یا توازی همچنان برقرار است، چون داور تک تک بازیکنان را می‌شناسد و فقط بازیکن خاطی را اخراج، جریمه و مسدود می‌کند و بقیه بازیکنان تیم به بازی خود ادامه می‌دهند، اما اگر هیچ یک از بازیکنان تیم مرتکب خطا نگردند، واضح است که هم‌روندی و توازی بالاتر هم خواهد بود.

توجه: همانند نخ‌های سطح کاربر، هم‌روندی و توازی برای نخ‌های سطح هسته نیز، برقرار است.

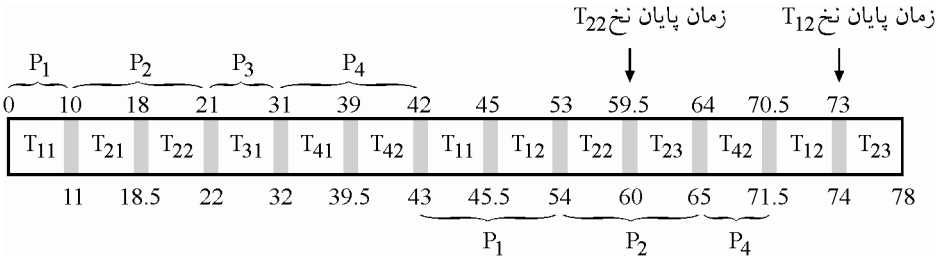
۲- گزینه (۱) صحیح است.

با توجه به جدول مطرح شده در صورت سؤال داریم:

فرآیند	نخ	زمان اجرای نخ
P_1	T_{11}	12
	T_{12}	9
P_2	T_{21}	7
	T_{22}	8
	T_{23}	8
P_3	T_{31}	9
P_4	T_{41}	7
	T_{42}	8

khalilifar.ir

با توجه به اعمال الگوریتم RR با Q=10 در سطح فرآیندها و به کارگیری FCFS در سطح نخ‌ها، نمودار زمانی گانت را ترسیم می‌کنیم.



زمان پایان نخ T_{12} برابر 73 و زمان پایان نخ T_{22} برابر 59.5 می باشد.

۳- گزینه (۴) صحیح است.

یک فرآیند در واقع شامل اجزایی مانند رجیسترهای پردازنده (از جمله شمارنده برنامه (PC)، یک پشته در حافظه (جهت نگهداری داده‌های موقتی، متغیرهای محلی، پارامترهای توابع و زیر روال‌ها) و بخش داده‌ای (Data Segment) (شامل داده‌ها و متغیرهای سراسری) است.

جهت نگهداری اطلاعات مربوط به فرآیندها نزد سیستم عامل از PCB استفاده می‌گردد، محتوای PCB برای هر فرآیند به صورت زیر است:

- شناسه فرآیند(Process ID)
- شناسه کاربر فرآیند یا حتی شناسه گروهی که کاربر فرآیند عضو آن است. (GID و UID)
- اولویت فرآیند
- رجیسترهای پردازنده (شامل PC، PSW ومابقی رجیسترها)
- حالت و وضعیت فرآیند (مانند جدید، آماده، در حال اجرا، منتظر و معلق)
- اطلاعات راجع به منابع در اختیار فرآیند (فایل‌ها، حافظه، پردازنده)
- اطلاعاتی راجع به میزان مصرف منابع توسط فرآیند (مانند میزان پردازنده مصرفی)
- اطلاعات مربوط به زمان‌بندی
- اشاره‌گرهایی به قسمت کد، داده و پشته (SP, DS, CS)
- اطلاعات مربوط به مدیریت حافظه (جداول صفحه، قطعه و مسائل حفاظتی)

توجه: در واقع PCB مانند یک مخزن است که اطلاعاتی که از یک فرآیند به فرآیند دیگر متغیر است، در آن ذخیره می‌شود.

پرونده حیاتی یک فرآیند در طول حیاتش در یک ساختمان داده به نام PCB نگهداری می‌گردد، هر فرآیند قبل از تعویض متن، پرونده حیاتی خود را در PCB قرار می‌دهد تا هنگام بازگشت مجدد، برای ادامه کار، این پرونده حیاتی بتواند مورد استفاده پردازنده و خود فرآیند قرار گیرد. زیرا پردازنده قادر به نگهداری اطلاعات مربوط به تمام فرآیندها نیست، چون در هر لحظه فقط

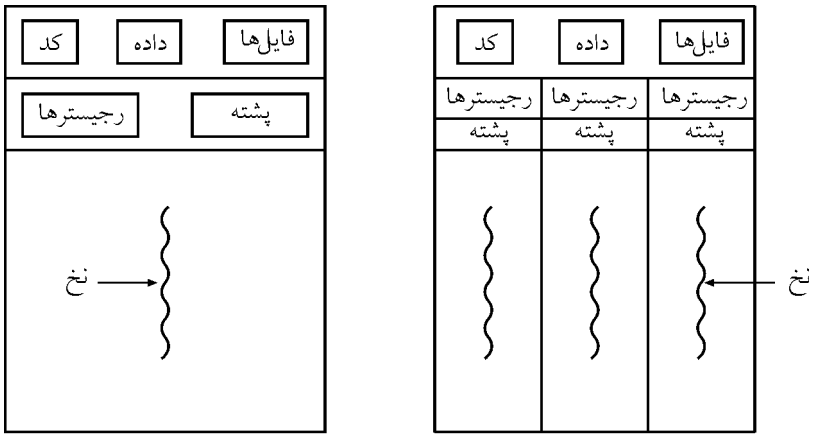
khalilifar.ir

امکانات و فضای کار لازم برای اجرای یک فرآیند را دارد. بنابراین هر فرآیند بطور مستقل از سایر فرآیندها در طول حیات خود باید پرونده حیاتی خود را درون PCB نگهداری کند.
پرونده حیاتی فرآیند، مانند پرونده پزشکی یک انسان در طول حیاتش می‌باشد. تا ادامه درمان انسان، در طول حیاتش امکان پذیر باشد. زیرا پزشکان قادر به نگهداری وضعیت درمانی همه بیماران در داخل ذهن خود نیستند.
توجه: نخ‌های همتا که در یک فرآیند قرار دارند، از کد، داده و منابع مشترک استفاده می‌کنند، اما هر نخ، شمارنده برنامه (PC)، مجموعه رجیستر و فضای پشته جداگانه‌ای در اختیار دارد. در واقع هر نخ، TBC مجزایی دارد.
گزینه اول: تغییر متن در فرآیندها منجر به تغییر ثبات‌ها، سیاست‌های حفاظتی و تغییر اشاره‌گر پشته (SP) می‌گردد. اما تغییر متن نخ‌ها منجر به پاک شدن TLB نمی‌گردد، توجه کنید که TLB برای ترجمه سریع صفحات یک فرآیند مورد استفاده قرار می‌گیرد. بنابراین گزینه اول نادرست است.
گزینه دوم: تغییر متن در فرآیند منجر به تغییر اشاره گر پشته، تغییر برنامه شمار (PC) و پاک شدن TLB می‌گردد، جدول TLB براساس فرآیند در حال اجرا پر می‌شود. تغییر متن در نخ‌ها همیشه باعث تخصیص سهم زمانی تازه نمی‌شود. چون سهم زمانی موقعی به نخ تعلق می‌گیرد که الگوریتم زمان‌بندی نخ‌ها RR باشد و اگر FIFO باشد سهم زمانی تازه معنا ندارد، سهمی در کار نیست. بنابراین گزینه دوم نادرست است.
گزینه سوم: تغییر متن در فرآیندها منجر به تغییر ثبات‌ها و پاک شدن TLB می‌گردد. جدول TLB براساس فرآیند در حال اجرا پر می‌شود. همچنین تغییر متن در نخ‌ها منجر به تغییر اشاره‌گر پشته (SP) می‌گردد، اما با TLB کاری ندارد. بنابراین گزینه سوم نادرست است.
گزینه چهارم: تغییر متن در فرآیندها منجر به تغییر ثبات‌های اجرایی برنامه و تغییر برنامه شمار (PC) می‌گردد. تغییر متن در نخ‌ها منجر به تغییر ثبات‌ها و جداول مربوط به مدیریت حافظه نمی‌گردد. اما منجر به تغییر اشاره‌گر (SP) می‌گردد. بنابراین گزینه چهارم درست است.

۴- گزینه (۲) صحیح است.

گزینه اول نادرست است. زیرا عبارت دوم نادرست، چون لیست فایل‌های باز شده، برای کل یک فرآیند ذخیره می‌شود. لذا در عمل تعویض متن بین نخ‌ها، لیست فایل‌های باز شده، توسط نخ ذخیره نمی‌شود. در واقع فایل‌های بازشده، برای فرآیند است و به شکل مشترک نخ‌های آن فرآیند، از آن استفاده می‌کنند. عبارت اول گزینه اول درست است، زیرا هر نخ ثبات‌های مخصوص خود را دارد که در پی عمل تعویض متن باید ثبات‌های خود را برای استفاده در آینده ذخیره کند. اما نخ وارد شونده، باید ثبات‌های خود را برای ادامه کار، بار کند.
به شکل زیر توجه کنید:

khalilifar.ir



فرآیند تک نخی فرآیند چند نخی

گزینه دوم درست است زیرا:

بطور کلی مدیریت و زمان‌بندی نخ‌ها به سه روش زیر انجام می‌گیرد:

۱- روش سطح کاربر یا مدل چند به یک (many to one)

۲- روش سطح هسته یا مدل یک به یک (one to one)

۳- روش ترکیبی (سطح کاربر و هسته) یا مدل چند به چند (many to many)

در روش ترکیبی موارد زیر برقرار است:

توجه: برای معرفی نخ‌های کاندید فرآیندهای سطح کاربر به زمان‌بند چند نخی در سطح هسته سیستم عامل، علاوه بر جایگاه‌های مخصوص نخ‌های فرآیندهای سیستم عامل در زمان‌بند چند نخی در سطح هسته سیستم عامل، تعدادی جایگاه، ویژه نخ‌های سطح کاربر نیز در زمان‌بند چند نخی در سطح هسته سیستم عامل در نظر گرفته شده است، به این جایگاه ویژه که محیط اجرای نخ نیز نامیده می‌شود، LWP گفته می‌شود. در واقع هر نخ کاندید انتخاب شده توسط زمان‌بند چند نخی در سطح کاربر، پس از معرفی به زمان‌بند چند نخی در سطح هسته، در یکی از جایگاه‌های ویژه که همان LWP است، جهت زمان‌بندی توسط زمان‌بند چندنخی در سطح هسته، قرار می‌گیرد.

توجه: LWP سرواژه عبارت Light Weight Process و به معنی فرآیند سبک وزن است.

توجه: هنگامی که یک نخ به زمان‌بند چند نخی در سطح هسته معرفی می‌گردد و در ادامه در یک LWP جهت زمان‌بندی توسط زمان‌بند چند نخی در سطح هسته قرار می‌گیرد، در طول حیات خود ممکن است در LWPهای متفاوتی بخش‌هایی از اجرای خود را طی کند، مثلاً یک نخ در صورت رسیدن به عملیات ورودی و خروجی، جایگاه خود یعنی LWP را واگذار می‌کند و در اجرای بعدی پس از پایان عملیات ورودی و خروجی ممکن است به یک LWP دیگر منتسب شود. توجه کنید که LWP محیط اجرای نخ می‌باشد. مانند بچه‌ای که داخل یکی از کابین‌های یک

khalilifar.ir

چرخ و فلک است، اما پس از مدتی به دلیل مسائل ورودی و خروجی، مجبور به ترک کابین چرخ و فلک می‌شود، اما بعدها پس از حل مسائل ورودی و خروجی ممکن است در سوار شدن مجدد در چرخ و فلک کابین دیگری را ملاقات کند.
گزینه سوم نادرست است، زیرا عبارت اول این گزینه براساس مطالب فوق نادرست است. اما عبارت دوم این گزینه درست است، زیرا زمان‌بندی نخ مطابق مطالب فوق می‌تواند توسط سیستم عامل یا کاربر و یا حتی هر دو توام با هم و به شکل ترکیبی انجام گردد.
گزینه چهارم نادرست است، مطابق آنچه برای گزینه سوم شرح داده شد.

۵- گزینه (۳) صحیح است.
به طور کلی مدیریت و زمان‌بندی نخ‌ها به سه روش زیر انجام می‌گردد:

۱- روش سطح کاربر یا مدل چند به یک (many to one)
در این روش فقط زمان‌بند پردازنده و زمان‌بند چند نخی در سطح کاربر وجود دارد و زمان‌بند چند نخی در سطح هسته در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل فقط فرآیندها را می‌شناسد و هیچ اطلاعاتی از نخ‌ها ندارد.
توجه: به دلیل آنکه نخ‌های یک فرآیند کاربر، تماماً در فضای کاربر مدیریت می‌شوند، اگر نخ موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل نه تنها آن نخ، بلکه کل فرآیند کاربر را که شامل تمام نخ‌های دیگر می‌باشد، مسدود می‌کند، زیرا هسته سیستم عامل خبری از نخ‌های داخل فرآیند کاربر ندارد، در واقع در این روش هسته سیستم عامل اصلاً زمان‌بند چند نخی ندارد که بخواهد، چند نخی را زمان‌بندی کند!
این راه‌کار، منجر به عدم امکان هم‌روندی (در سیستم‌های تک پردازنده‌ای) نخ‌های داخل یک فرآیند کاربر در حالت انسداد یک نخ داخل یک فرآیند کاربر می‌گردد. البته اگر نخی داخل یک فرآیند کاربرد مسدود نگردد، امکان هم‌روندی میان نخ‌های داخل فرآیند کاربر برقرار است.
توجه: در روش سطح کاربر، یک فرآیند چند نخی کاربر نمی‌تواند از امتیازات چند پردازنده‌ای بهره ببرد. زیرا در روش سطح کاربر، هسته سیستم عامل در هر لحظه فقط یک پردازنده را در اختیار نخ‌های یک فرآیند کاربر قرار می‌دهد. حتی اگر چند پردازنده موجود باشد.
بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند کاربر وجود ندارد.

۲- روش سطح هسته یا مدل یک به یک (one to one)
در این روش فقط زمان‌بند پردازنده و زمان‌بند چند نخی در سطح هسته وجود دارد و زمان‌بند چند نخی در سطح کاربر در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل نه تنها فرآیندها را می‌شناسد بلکه از وجود نخ‌های داخل یک فرآیند نیز در صورت وجود نخ آگاه است.

khalilifar.ir

توجه: به دلیل آنکه نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل)، تماماً در فضای هسته سیستم عامل مدیریت می‌شوند، و هسته سیستم عامل از وجود نخ‌های یک فرآیند آگاه است، اگر نخ موجود در یک فرآیند، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم خانواده با آن نخ مسدود شده، همچنان می‌توانند از پردازنده بهره ببرند.

توجه: این راه‌کار، منجر به امکان هم‌روندی (در سیستم‌های تک پردازنده‌ای) و امکان توازی (در سیستم‌های چند پردازنده‌ای) میان فرآیندهای مختلف و یا میان نخ‌های داخل یک فرآیند می‌شود. البته اگر نخی مسدود نگردد، درجه همروندی و توازی بالاتر هم خواهد رفت، زیرا در این صورت همه نخ‌ها به طور همروند یا موازی در حال حرکت هستند.

توجه: در روش سطح هسته، یک فرآیند چند نخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چند پردازنده‌ای بهره ببرد. در روش سطح هسته، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد.

۳- روش ترکیبی (سطح کاربر و هسته) یا مدل چند به چند (many to many)

این روش از اجتماع دو روش سطح کاربر و هسته ابداع گردیده است. در این روش علاوه بر زمان‌بند پردازنده و زمان‌بند چند نخی در سطح هسته سیستم عامل، زمان‌بند چند نخی نیز در سطح کاربر برای زمان‌بندی و اولویت‌دهی نخ‌های فرآیند کاربر وجود دارد. در واقع زمان‌بندی نخ‌های فرآیندهای سیستم عامل، توسط هسته سیستم عامل و زمان‌بندی نخ‌های فرآیندهای کاربر، توسط برنامه کاربر انجام می‌گردد، که این امر منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد.

توجه: به دلیل آنکه نخ‌های کاندید فرآیندهای سطح کاربر، به هسته سیستم عامل معرفی می‌گردند و هسته سیستم عامل از وجود نخ‌های کاندید فرآیندهای سطح کاربر آگاه است، اگر یک نخ کاندید موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم خانواده با آن نخ مسدود شده، همچنان می‌توانند از پردازنده بهره ببرند.

توجه: این راه‌کار، منجر به امکان همروندی (در سیستم‌های تک پردازنده‌ای) و امکان توازی (در سیستم‌های چند پردازنده‌ای) میان فرآیندهای مختلف کاربر و یا میان نخ‌های داخل یک فرآیند کاربر می‌شود. البته اگر نخی مسدود نگردد، درجه همروندی و توازی بالاتر هم خواهد رفت، زیرا در این صورت همه نخ‌ها به طور همروند یا موازی در حال حرکت هستند.

توجه: در روش ترکیبی، یک فرآیند چند نخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چند پردازنده‌ای بهره ببرد. در روش ترکیبی، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. بنابراین در این روش نیز

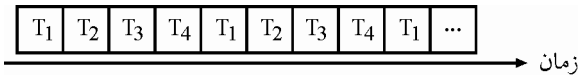
khalilifar.ir

امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد. مطابق مطالب بیان شده، گزینه‌های اول، دوم و چهارم درست هستند و گزینه سوم نادرست است. البته در گزینه سوم این عبارت که در مدل‌های یک به یک امکان همزمانی کامل بین نخ‌ها وجود دارد درست و این عبارت که در مدل‌های چند به یک امکان همزمانی کامل بین نخ‌ها وجود دارد نادرست است، زیرا در مدل چند به یک از آن جا که هسته سیستم عامل، نخ‌های کاربر را نمی‌شناسد، در صورت انسداد یک نخ کاربر، سیستم عامل کل فرآیند را مسدود می‌کند، بنابراین همزمانی و همروندی کامل در مدل چند به یک بین نخ‌های کاربر همواره برقرار نیست.

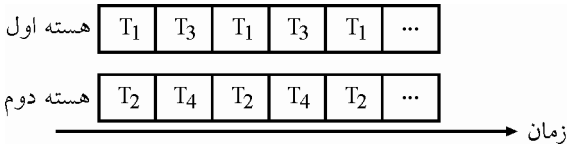
۶- گزینه (۱) صحیح است.

روند اخیر در طراحی سیستم‌ها، به گونه‌ای است که چندین هسته محاسباتی را در یک تراشه قرار می‌دهند. به گونه‌ای که هر هسته به صورت یک پردازنده مجزا برای سیستم عامل تلقی می‌شود. برنامه‌نویسی چند رشته نخی، مکانیزمی را برای استفاده مؤثرتر از هسته‌های سیستم فراهم نموده و توازی را افزایش می‌دهد. یک فرآیند را با چهار رشته نخ در نظر بگیرید. در یک سیستم با یک هسته پردازش، سیستم در یک زمان تنها یک رشته نخ را اجرا می‌کند. در یک سیستم چند رشته نخی، با چند هسته توازی به این معنی است که رشته نخ‌ها می‌توانند به طور موازی اجرا شوند و سیستم می‌تواند در هر هسته یک رشته نخ را قرار دهد.

شکل زیر اجرای همروند یک فرآیند چند نخی را در یک سیستم تک هسته‌ای نمایش می‌دهد:



شکل زیر اجرای موازی یک فرآیند چند نخی را در یک سیستم دو هسته‌ای نمایش می‌دهد:



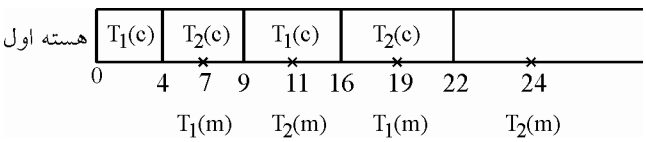
گزینه دوم: T_1 و T_3 روی هسته اول و T_2 و T_4 روی هسته دوم

گزینه چهارم: T_2 و T_4 روی هسته اول و T_1 و T_3 روی هسته دوم

بنابراین گزینه دوم با گزینه چهارم فرقی ندارد!!!

حال به بررسی نخ‌ها می‌پردازیم:

اگر T_1 و T_2 روی هسته اول و T_3 و T_4 روی هسته دوم باشند:

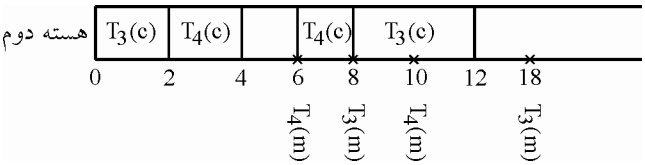


khalilifar.ir

بنابراین زمان خروج نخ‌های T_1 و T_2 به صورت زیر است:

T_1 خروج = 19 (16 بعد از حافظه ثانیه 3)

T_2 خروج = 24 (22 بعد از حافظه ثانیه 2)



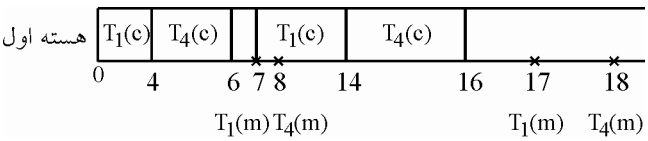
بنابراین زمان خروج نخ‌های T_3 و T_4 به صورت زیر است:

T_3 خروج = 18 (12 بعد از حافظه ثانیه 6)

T_4 خروج = 10 (8 بعد از حافظه ثانیه 2)

$$\text{متوسط زمان تأخیر} = \frac{T_1+T_2+T_3+T_4}{4} = \frac{19+24+18+10}{4} = \frac{71}{4} = 17.75$$

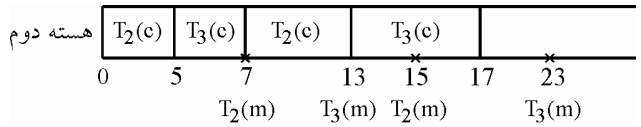
اگر T_1 و T_4 روی هسته اول و T_2 و T_3 روی هسته دوم باشند:



بنابراین زمان خروج نخ‌های T_1 و T_4 به صورت زیر است:

T_1 خروج = 17 (14 بعد از حافظه پس ثانیه 3)

T_4 خروج = 18 (16 بعد از حافظه پس ثانیه 2)



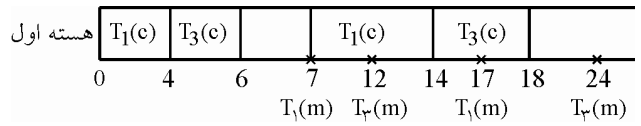
بنابراین زمان خروج نخ‌های T_2 و T_3 به صورت زیر است:

T_2 خروج = 15 (13 بعد از حافظه پس ثانیه 2)

T_3 خروج = 23 (17 بعد از حافظه پس ثانیه 6)

$$\text{متوسط زمان تأخیر} = \frac{T_1+T_2+T_3+T_4}{4} = \frac{17+15+23+18}{4} = \frac{73}{4} = 18.25$$

اگر T_1 و T_3 روی هسته اول و T_2 و T_4 روی هسته دوم باشند:

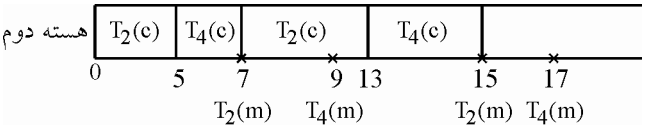


khalilifar.ir

بنابراین زمان خروج نخ‌های T_1 و T_3 به صورت زیر است:

T_1 خروج = 17 (14 بعد از حافظه 3 ثانیه)

T_3 خروج = 24 (18 بعد از حافظه 6 ثانیه)



بنابراین زمان خروج نخ‌های T_2 و T_4 به صورت زیر است:

T_2 خروج = 15 (13 بعد از حافظه 2 ثانیه)

T_4 خروج = 17 (15 بعد از حافظه 2 ثانیه)

$$\text{متوسط زمان تأخیر} = \frac{T_1+T_2+T_3+T_4}{4} = \frac{17+15+24+17}{4} = \frac{73}{4} = 18.25$$

۷- گزینه ( ) صحیح است.

نخ‌ها هم مانند فرآیندها می‌توانند حالت‌های مختلفی را تجربه کنند مانند آماده، در حال اجرا یا منتظر (مسدود). در واقع پردازنده می‌تواند بین نخ‌ها به اشتراک گذاشته شود. حالت آماده اجرا همان معنی حالت آماده را می‌دهد بنابراین گزینه دوم و سوم یکسان هستند، سازمان سنجش آموزش کشور در کلید اولیه خود ابتدا گزینه سوم را به عنوان پاسخ اعلام کرده بود، سپس در کلید نهایی نظر خود را عوض کرد و گزینه سوم را با تأثیر مثبت اعلام کرد.

در روش سطح کاربر یا مدل چند به یک (Many to One) فقط زمان‌بند پردازنده و زمان‌بند چندنخی در سطح کاربر وجود دارد و زمان‌بند چند نخی در سطح هسته در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل فقط فرآیندها را می‌شناسد و هیچ اطلاعاتی از نخ‌ها ندارد. در واقع اولویت‌بندی نخ‌ها، مدیریت نخ‌ها و زمان‌بند چندنخی در سطح کاربر و توسط یک بسته نرم‌افزاری انجام می‌گردد. بدین معنی که نخ‌ها را برنامه‌نویس مشخص می‌کند و مدیریت آن‌ها را نیز بر عهده می‌گیرد. بنابراین زمان‌بند پردازنده، براساس الگوریتم مشخصی مثلاً نوبت چرخشی پردازنده را در اختیار یکی از فرآیندهای آماده قرار می‌دهد. سپس زمان‌بند چند نخی در سطح کاربر، متناسب با کاربردی که در آن فرآیند به کار گرفته می‌شود، الگوریتم زمان‌بند را انتخاب کرده و تصمیم می‌گیرد که پردازنده در اختیار کدام‌یک از نخ‌های آماده در فرآیند موردنظر قرار گیرد و تا زمانی که پردازنده در تملک فرآیند باشد و یا تا قبل از پایان برش زمانی مربوط به فرآیند، نخ‌های یک فرآیند از پردازنده بهره‌مند می‌شوند و به محض مسدود شدن یک نخ، و یا پایان برش زمانی یک فرآیند، یا اتمام فرآیند، پردازنده به فرآیند بعدی تعلق می‌گیرد.

khalilifar.ir

توجه: در این روش نخ ماهیت منطقی دارد و از دید کاربر فقط وجود دارد، در واقع از نظر سیستم عامل ماهیت فیزیکی ندارد، بنابراین نخ کاربر در این روش همانند یک تابع در فرآیند می‌باشد که از رجیستر و پشته مختص به خود نیز بهره‌مند نمی‌باشد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، بین پدران خانواده‌ها تقسیم گردد و این پدران خانواده‌ها باشند که تصمیم بگیرند به هر عضو خانواده چه مقدار نقدینگی تعلق بگیرد. در این روش فقط پدران شماره حساب مختص به خود را دارند. اما اگر یکی از اعضای خانواده خطایی انجام دهد و محکوم گردد، آنگاه تمام اعضای خانواده برای مدتی از خدمات دولت محکوم می‌گردند، زیرا در این مدل، دولت فقط پدران خانواده را می‌شناسد و از اعضای خانواده اطلاعی ندارد. بنابراین حساب پدر خانواده برای مدتی مسدود می‌گردد.

توجه: عمل تعویض متن مابین نخ‌های یک فرآیند کاربر، کاملاً در سطح کاربر و با سربار بسیار ناچیز (در حد فراخوانی نخ بعدی) و بدون تغییر حالت پردازنده به مد هسته پردازنده، توسط زمان‌بند چندنخی در سطح کاربر (برنامه‌های کاربر) انجام می‌گردد. اما عمل تعویض متن مابین فرآیندها (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل و در مد هسته پردازنده، توسط زمان‌بند پردازنده برای انتخاب یک فرآیند جدید بر اساس یک الگوریتم خاص انجام می‌گردد.

توجه: به دلیل آنکه نخ‌های یک فرآیند کاربر، تماماً در فضای کاربر مدیریت می‌شوند، اگر نخ موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل نه تنها آن نخ، بلکه کل فرآیند کاربر را که شامل تمام نخ‌های دیگر می‌باشد، مسدود می‌کند، زیرا هسته سیستم عامل خبری از نخ‌های داخل فرآیند کاربر ندارد. در واقع در این روش هسته سیستم عامل نخ‌ها را همانند توابع داخل یک فرآیند می‌بیند، یعنی هسته سیستم عامل، یک فرآیند چند نخی سطح کاربر را، مانند یک فرآیند تک نخی اما دارای چند تابع مختلف می‌بیند! بنابراین در اینجا انتقال از حالت آماده نخ‌های همتای دیگر به حالت مسدود (منتظر) به دلیل اجرای فراخوان سیستمی مسدود کننده توسط یک نخ همتای دیگر را داریم، بنابراین گزینه دوم و سوم هم می‌تواند در این حالت درست باشد ولی نه در حالت کلی.

توجه: این راه‌کار، منجر به عدم امکان هم‌روندی (در سیستم‌های تک‌پردازنده‌ای) نخ‌های داخل یک فرآیند کاربر در حالت انسداد یک نخ داخل یک فرآیند کاربر می‌گردد.

البته اگر نخی داخل یک فرآیند کاربر مسدود نگردد، امکان هم‌روندی میان نخ‌های داخل فرآیند کاربر برقرار است. مانند یک تیم فوتبال ۱۱ نفره که اگر بازیکنی مرتکب خطا گردد، از آن‌جا که داور فقط نام تیم را می‌شناسد و نه تک‌تک بازیکنان تیم را، آن‌گاه کل تیم را جریمه، اخراج و

khalilifar.ir

مسدود می‌کند. اما اگر هیچ‌یک از بازیکنان تیم مرتکب خطایی نگردد، واضح است که هم‌روندی برقرار است.

توجه: فرض کنید یک کیک داریم که آن را به چهار قسمت مساوی تقسیم کرده‌ایم، همچنین فرض کنید خوردن هر بخش کیک یک ساعت زمان بخواهد، اگر یک نفر بخواهد تمام این کیک را بخورد، پس ۴ ساعت طول می‌کشد اگر ۴ نفر بخواهند تمام این کیک را بخورند و به هر نفر یک بخش کیک داده شود، آنگاه خوردن تمام کیک به‌طور موازی ۱ ساعت طول خواهد کشید. در روش سطح کاربر، یک فرآیند چند نخی کاربر نمی‌تواند از امتیازات چند پردازنده‌ای بهره ببرد، زیرا در روش سطح کاربر، هسته سیستم عامل در هر لحظه فقط یک پردازنده را در اختیار نخ‌های یک فرآیند کاربر قرار می‌دهد. حتی اگر چند پردازنده موجود باشد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر نیست. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند کاربر وجود ندارد.

توجه: نرم‌افزارهای POSIX P-threads و Mach C-thread به عنوان یک بسته نرم‌افزاری، می‌توانند جهت مدیریت نخ‌ها و زمان‌بند چند نخی در سطح کاربر مورد استفاده قرار گیرند.

توجه: در این روش تخصیص منابع و زمان‌بندی پردازنده، بر روی فرآیندها انجام می‌شود. همچنین زمان‌بندی نخ‌های سطح کاربر، بر عهده زمان‌بند چندنخی سطح کاربر خواهد بود.

توجه: سیستم عامل سولاریس، مدل چند به یک را پیاده سازی می‌کند.

حذف نخی (Thread Cancellation)

حذف نخی به معنی **خاتمه‌دادن** یک نخ قبل از تکمیل‌شدن آن است. برای‌مثال اگر چند نخ بطور همروند درحال جستجو در یک پایگاه داده باشند، در صورتی که یکی از نخ‌ها نتیجه را برگرداند، **احتمالاً** ادامه کار مابقی نخ‌ها کنسل می‌شود. حالت دیگر وقتی است که یک کاربر، دکمه‌ای را در یک مرورگر وب می‌فشارد که صفحه وب را از ادامه بارگذاری باز می‌دارد، اغلب یک صفحه وب با استفاده از چندین نخ بارگذاری می‌شود (هر تصویر توسط یک نخ مجزا بارگذاری می‌شود). وقتی که یک کاربر، دکمه stop را در مرورگر می‌فشارد، تمامی نخ‌های بارگذاری صفحه کنسل می‌شوند. نخی که باید کنسل شود اغلب **نخ هدف** (Target Thread) معرفی می‌شود. حذف (کنسل‌کردن) نخ هدف ممکن است در دو سناریوی مختلف رخ دهد:

۱- حذف ناهمگام (Asynchronous Cancellation)

یک نخ بلافاصله **نخ هدف** را خاتمه دهد، حذف نخ در این حالت به شکل نا ایمن انجام می‌گردد، و بدون بررسی اشتراکات نخ حذف شده با نخ‌های دیگر.

بنابراین در این حالت، انتقال از حالت آماده نخ هدف به حالت خاتمه وجود دارد، پس گزینه اول هم می‌تواند درست باشد ولی نه در حالت کلی.

khalilifar.ir

۲- حذف همراه تعویق (Asynchronous Cancellation)

نخ هدف به طور متناوب چک می‌شود که آیا می‌تواند خاتمه پیدا کند یا نه، و به آن اجازه داده می‌شود که بنا به درخواستش در صورت امکان به شکل امن، خاتمه یابد. حذف نخ در این حالت به شکل ایمن انجام می‌گردد، عموم سیستم‌عامل‌ها حذف همراه تعویق را پشتیبانی می‌کنند. به طور مثال سیستم عامل لینوکس حذف همراه تعویق را پشتیبانی می‌کند.

بنابراین در این حالت، انتقال از حالت اجرای نخ هدف به خاتمه وجود دارد، ولی نه در حالت کلی.

khalilifar.ir

تست‌های فصل چهارم: مدیریت حافظه اصلی

۱- آدرس منطقی 0001010010111010 را در نظر بگیرید. با مدیریت صفحه‌بندی ۲۵۶ صفحه‌ای برای یک حافظه با 256 قاب (frame) و استفاده از جدول صفحه‌ای که در آن هر شماره قاب $\frac{1}{4}$ شماره صفحه باشد، کدام گزینه در مورد مدیریت این حافظه و آدرس فیزیکی متناظر با آدرس منطقی فوق صحیح است؟ (مهندسی کامپیوتر – دولتی۸۹)

۱) اگرچه این روش نگاشت صفحه مشکل دارد ولی آدرس فیزیکی متناظر 0000101010111010 است.

۲) اگرچه این روش نگاشت صفحه مشکل دارد ولی آدرس فیزیکی متناظر 0000010110111010 است.

۳) این روش نگاشت صفحه بدون مشکل کار می‌کند و آدرس فیزیکی متناظر 0000010110111010 است.

۴) این روش نگاشت صفحه بدون مشکل کار می‌کند و آدرس فیزیکی متناظر 0000101010111010 است.

۲- مدیریت حافظه در یک سیستم فرضی به صورت قطعه‌بندی صفحه‌بندی شده (paged segmentation) است. و اندازه هر صفحه 4 کیلو بایت است. هر درایه (entry) جدول قطعه دارای 3 بایت و به صورت زیر است: (مهندسی IT- دولتی۸۹)

بایت سوم	بایت دوم بایت اول
LIMIT	PTBA

و هر درایه جدول صفحه یک بایتی است و نشان‌دهنده شماره قاب (frame) است. در PCB یک فرآیند برای آدرس پایه جدول قطعه (STBA) مقدار 0AFEH دیده می‌شود. اگر در این فرآیند آدرس منطقی [3456H , 02H] تولید شود، آدرس فیزیکی نظیر چه خواهد بود. بخش اول آدرس منطقی شماره قطعه است. حرف H به معنی Hex است. محتویات حافظه به صورت زیر است:

0B00H	08H	0B08H	08H
0B01H	09H	0B09H	09H
0B02H	03H	0B0AH	00H
0B03H	0AH	0B0BH	0BH
0B04H	0BH	0B0CH	0AH
0B05H	09H	0B0DH	0CH
0B06H	05H	0B0EH	04H
0B07H	0BH	0B0FH	05H

۱) 08345H

۲) 0A456H

۳) 0B560H

۴) 08456H

khalilifar.ir

۳- حافظه اصلی با وضعیت نشان داده شده در شکل را در نظر بگیرید. اگر مدیریت حافظه اصلی براساس اختصاص‌دهی پویا باشد و اختصاص‌دهی فضای خالی به فرآیندها براساس Next-fit انجام گیرد و فرآیندهای P_0, P_1, ..., P_5 جهت اجرا شدن، مطابق با اطلاعات جدول زیر وارد سیستم شوند، با فرض اینکه از بین فضاهای پر شماره 1 تا 4، فقط فضای پر 3 در لحظه t+20 آزاد گردد (دیگر فضاهای پر تا اتمام اجرای فرآیندهای فوق آزاد نمی‌گردند)، متوسط زمان بازگشت (turn around) و متوسط زمان انتظار فرآیندهای فوق در روش FCFS به ترتیب چقدر است؟ (مهندسی کامپیوتر – دولتی۹۰)

زمان سرویس	حافظه مورد نیاز (KB)	زمان ورود	فرآیند
30	25	t	P_0
40	20	t+1	P_1
20	40	t+2	P_2
45	25	t+3	P_3
35	10	t+4	P_4
15	35	t+5	P_5

فضای پر 1 (30KB)
فضای خالی 1 (20KB)
فضای پر 2 (10KB)
فضای خالی 2 (40KB)
فضای پر 3 (20KB)
فضای خالی 3 (30KB)
فضای پر 4 (40KB)
فضای خالی 4 (30KB)

۱) 110.83 و 80 ۲) 109.5 و 85.83 ۳) 109.5 و 78.67 ۴) 116.67 و 85.83

۴- فرض کنید سیستمی از فضای آدرس‌دهی مجازی $2^{40}B$ پشتیبانی می‌کند. در این سیستم اندازه حافظه فیزیکی قابل دسترسی $2^{32}B$ و طول هر قاب (Frame) حافظه در این سیستم $2^{10}B$ می‌باشد. این سیستم از روش صفحه‌بندی (Paging) برای مدیریت حافظه استفاده کرده است. با فرض اینکه هر مدخل از جدول صفحه به 10bit به عنوان بیت‌های کنترلی (بیت حضور - غیاب و ...) نیاز داشته باشد، در این صورت برای اینکه هر جدول صفحه جزئی دقیقاً در یک قاب قرار گیرد (الزامی برای پیوسته قرار گرفتن هر جدول صفحه در حافظه اصلی نباشد) باید حداقل از جدول صفحه چند سطحی استفاده شود؟ (مهندسی IT- دولتی۹۰)

۱) سه ۲) دو ۳) چهار ۴) پنج

۵- کدام گزینه در مورد جدول صفحه وارونه درست نیست؟ (مهندسی IT- دولتی۹۳)

۱) به ازاء هر صفحه فیزیکی حافظه در جدول صفحه وارونه تنها یک خانه دارد.

۲) در جدول صفحه وارونه هر پردازه یک جدول صفحه دیگر برای خود نگهداری می‌نماید.

۳) در جدول صفحه وارونه، تنها کل اطلاعات فضای آدرس منطقی برای یک پردازه در دسترس است.

۴) هر خانه از جدول صفحه وارونه دست کم دارای آدرس منطقی صفحه و شماره پردازه استفاده‌کننده از این صفحه است.

khalilifar.ir

۶- کدامیک از عبارات زیر صحیح است؟ (مهندسی کامپیوتر-دولتی۹۴)

۱) اندازه آدرس‌های مجازی و فیزیکی با هم برابر است.

۲) dispatcher اولویت ریسه‌ها (thread) را تعیین و مقدار دهی می‌نماید.

۳) اگر یک ریسه (thread)، CPU-Bound باشد باید اولویت آن برای دسترسی به I/O بالاتر از ریسه‌های I/O Bound باشد.

۴) اگر در یک سیستم مدیریت حافظه garbage collection استفاده شود این سیستم دچار fragmentation نمی‌شود.

۷- فرض کنید اندازه هر صفحه 1KB است و هر مدخل صفحه 4 بایت فضا می‌گیرد. برای اینکه بتوانیم یک آدرس 34 بیتی را نگاشت کنیم به طوری که هر جدول صفحه تنها در یک صفحه ذخیره شود. چند سطح جدول صفحه نیاز است؟ فرض کنید که هر جدول به اندازه یک صفحه است. (مهندسی IT – دولتی ۹۴)

۱) 4 ۲) 3 ۳) 2 ۴) 1

۸- در یک سیستم حافظه مجازی از نوع قطعه – صفحه‌ای، بخشی از جدول TLB به صورت زیر است. اگر تعداد کلمات هر صفحه 4096 باشد، حجم حافظه مجازی چند برابر حافظه اصلی است؟ (مهندسی کامپیوتر– دولتی ۹۵)

← 4 →	← 12 →	← 12 →
قطعه	صفحه	بلوک
1	2FF	012
5	02A	2A5

۱) 16
۲) 8
۳) 4
۴) 2

۹- سیستمی علاوه بر ذخیره جدول صفحه در حافظه اصلی، از جدول TLB نیز با نرخ miss برابر 20% استفاده می‌کند. اگر خواندن از حافظه اصلی 100ms زمان بردارد و درصد کارایی سیستم در صورت استفاده نکردن از جدول TLB برابر با 80% باشد، خواندن از TLB چند نانوثانیه زمان لازم دارد؟ (مهندسی کامپیوتر– دولتی ۹۵)

۱) 20 ۲) 40 ۳) 50 ۴) 60

۱۰- در یک ساختار حافظه، سیستم صفحه‌بندی سه سطحی از آدرس مجازی مطابق فرمت زیر استفاده می‌شود. اگر نرخ برخورد هر سطح 90% باشد، احتمال برخورد تبدیل یک آدرس مجازی به فیزیکی چقدر است؟ فرض شود که احتمال برخورد هر سطح مستقل از احتمال برخورد در سطح دیگر است. (مهندسی IT – دولتی ۹۵)

khalilifar.ir

P3	P2	P1	D

۱) 0.001 ۲) 0.1 ۳) 0.729 ۴) 0.9

۱۱- فرض کنید که اندازه فضای آدرس منطقی، 8 صفحه 1024 بایتی و اندازه حافظه فیزیکی 32 قاب است. طول آدرس منطقی و آدرس فیزیکی (به ترتیب از راست به چپ) هر کدام چند بیت می‌باشد؟ (مهندسی IT – دولتی ۹۵)

۱) 11,9 ۲) 13,11 ۳) 15,13 ۴) 13,15

۱۲- کدام گزینه درباره نرخ برخورد (hit ratio) در TLB درست نیست؟ (مهندسی IT – دولتی ۹۵)

۱) اگر تعداد خانه‌های TLB بیشتر شود نرخ برخورد بیشتر می‌شود.

۲) اگر اندازه صفحه‌ها بزرگتر شود نرخ برخورد بیشتر می‌شود.

۳) برای صفحه‌های با اندازه ثابت، اگر طول آدرس فیزیکی بیشتر شود نرخ برخورد تغییر نمی‌کند.

۴) برای صفحه‌های با اندازه ثابت، با افزایش اندازه قطعه (segment)، نرخ برخورد بیشتر می‌شود.

۱۳- یک کامپیوتری با 8 گیگابایت حافظه را در نظر بگیرید که اندازه هر صفحه 8 کیلوبایت و هر خانه از جدول 4 بایت باشد. در صورتیکه این کامپیوتر از جدول چند سطحی استفاده نماید که هر جدول صفحه در یک صفحه ذخیره شود و بخواهیم آدرس مجازی 46 بیتی را به آدرس فیزیکی تبدیل نماییم. برای خواندن یک کلمه 32 بیتی به چند دسترسی به حافظه نیاز است؟

(مهندسی کامپیوتر – دولتی ۹۶)

۱) 4 ۲) 3 ۳) 2 ۴) 1

۱۴- در یک سیستم صفحه‌بندی که دارای 34 بیت آدرس است 23 بیت اول برای شماره صفحه و 11 بیت بعدی برای آدرس‌دهی درون صفحه است. در یک سیستم با صفحه‌بندی وارونه (Inverted Page Table) با 128 مگابایت حافظه، جدول صفحه دارای چند خانه است؟

(مهندسی IT – دولتی ۹۶)

۱) 2^{16} ۲) 2^{17} ۳) 2^{23} ۴) 2^{34}

۱۵- سه سیستم مدیریت حافظه S_1، S_2 و S_3 را در نظر بگیرید. S_1 دارای یک TLB با زمان پاسخ 120ns است که نرخ برخورد آن 60% می‌باشد. S_2 دارای یک TLB بزرگ‌تر ولی کندتر می‌باشد که زمان پاسخ آن 150ns و نرخ برخورد آن 80% می‌باشد. S_3 دارای TLB با زمان پاسخ 120ns و نرخ برخورد 70% است. دو سیستم S_1 و S_2 همزمان با ارسال درخواست به TLB، آن را به حافظهٔ اصلی نیز ارسال می‌کنند تا در صورت عدم یافتن آدرس در TLB، زمان پاسخ کاهش یابد. سیستم S_3 پس از دریافت پاسخ از TLB و عدم یافتن آدرس، درخواست را به حافظهٔ اصلی

khalilifar.ir

ارسال می‌کند. کمترین و بیشترین میانگین زمان پاسخ این سه سیستم به ترتیب از راست به چپ چند نانوثانیه است؟ **(مهندسی IT – دولتی ۹۶)**

۱) 312-300 ۲) 312-264 ۳) 300-312 ۴) 300-264

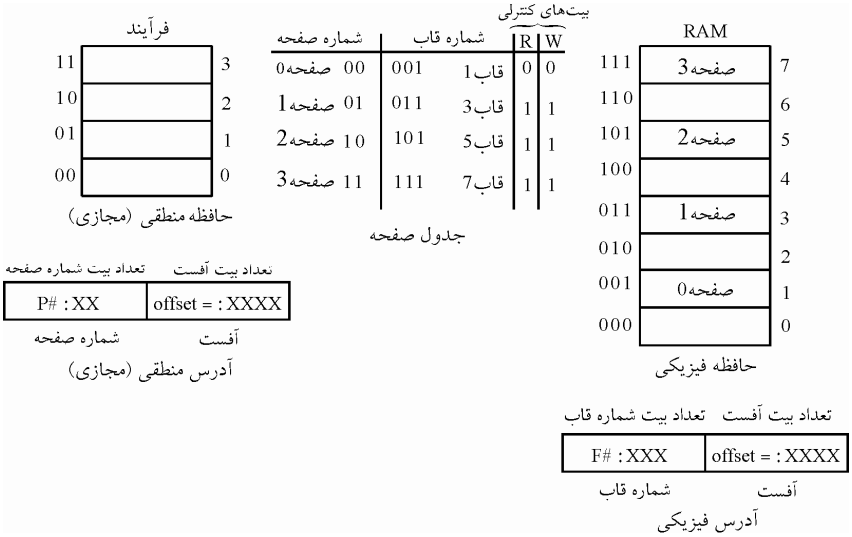
khalilifar.ir

پاسخ تست‌های فصل چهارم: مدیریت حافظه اصلی

۱- گزینه (۲) صحیح است.

بطور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه بندی به صورت زیر است:

توجه: فرض کنید اندازه هر صفحه 16 بایت، اندازه هر فرآیند 64 بایت و اندازه حافظه فیزیکی برابر 128 بایت باشد.



$$\text{تعداد صفحات فرآیند} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{64}{16} = 4$$

$$\text{تعداد قاب‌های حافظه‌ی فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{128}{16} = 8$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^4 = 2bit$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^8 = 3bit$$

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{16} = 4bit$$

عرض جدول صفحه × تعدادصفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد. دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطح جدول صفحه می‌باشد.

khalilifar.ir

در جدول فوق 2 بیت مربوط به بیت‌های کنترلی و 3 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

مثال: عرض جدول صفحه فوق برابر $3bit + 2bit = 5bit$ می‌باشد، بنابراین داریم:

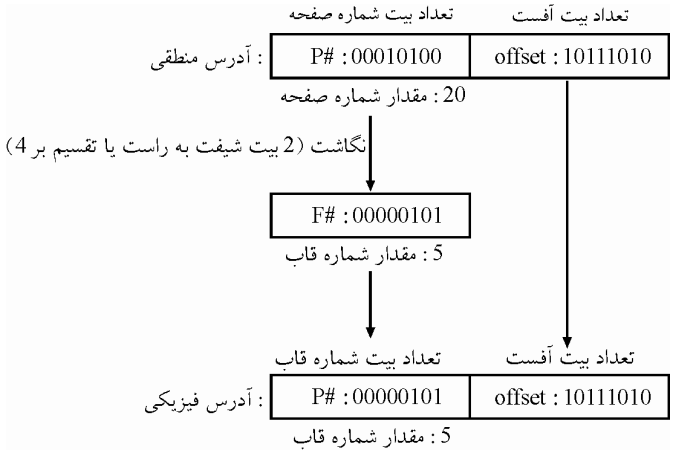
اندازه جدول صفحه $= 4 \times 5bit = 20bit$

حال برگردید به مسئله مطرح شده در صورت سوال .....

توجه: در صورت سوال مطرح شده، برای تبدیل شماره صفحه به شماره قاب یک الگوی ثابت و از قبل تعیین شده، مشخص گردیده است که این فرض نادرست می‌باشد. تعیین شماره قاب در حافظه فیزیکی برای یک صفحه از یک فرآیند براساس قاب‌های آزاد در حافظه فیزیکی و توسط سیستم عامل انجام می‌گردد و نه براساس یک الگوی از قبل مشخص شده و نگاشت ثابت، زیرا یک صفحه از فرآیند همواره به یک قاب خاص از حافظه فیزیکی نگاشت نمی‌شود.

از آنجا که فرآیند موجود در این سیستم شامل 256 صفحه است، بنابراین 8 بیت سمت چپ $(b = \log_2^{256} = 8bit)$ از آدرس منطقی 16 بیتی مربوط به شماره صفحه و 8 بیت سمت راست باقیمانده مربوط به آفست است. همچنین از آنجا که حافظه فیزیکی موجود در این سیستم نیز شامل 256 قاب است، بنابراین 8 بیت سمت چپ $(b = \log_2^{256} = 8bit)$ از آدرس فیزیکی نیز مربوط به تعداد بیت‌های شماره قاب می‌باشد. با توجه به اینکه در ترجمه آدرس منطقی به فیزیکی در سیستم‌های صفحه‌بندی شده، تعداد بیت‌های آفست ثابت مانده و فقط شماره صفحه با شماره قاب جای‌گزین می‌گردد، بنابراین آدرس فیزیکی نیز دارای طولی برابر 16 بیت خواهد بود.

باتوجه به فرض مطرح شده در صورت سؤال که شماره قاب $\frac{1}{4}$ شماره صفحه است (که فرض نادرستی است) اما برای این کار، کافی است شماره صفحه را بر 4 تقسیم کنیم، یا مقدار دودویی شماره صفحه را دوبار به سمت راست شیفت دهیم تا مقدار شماره قاب بدست بیاید و سپس 8 بیت سمت راست آدرس منطقی را به عنوان آفست به آن بیافزاییم:



khalilifar.ir

ولی این روش نگاشت نادرست است، چرا که همیشه مجموع صفحاتی که در 2 بیت سمت راست شماره صفحه و به شکل 00، 01، 10، 11 متفاوت ولی در 6 بیت باقی‌مانده یکسان هستند به دلیل انتقال 2 بیت به سمت راست، همگی یک شماره قاب واحد خواهند داشت. یعنی به هر قاب موجود در حافظه فیزیکی، 4 صفحه از آدرس منطقی نسبت داده می‌شود. مثلاً صفحات شماره 0، 1، 2 و 3 همگی به شماره قاب 0 نگاشت می‌شوند و نمی‌توانند همزمان با هم درون حافظه فیزیکی قرار داده شوند.

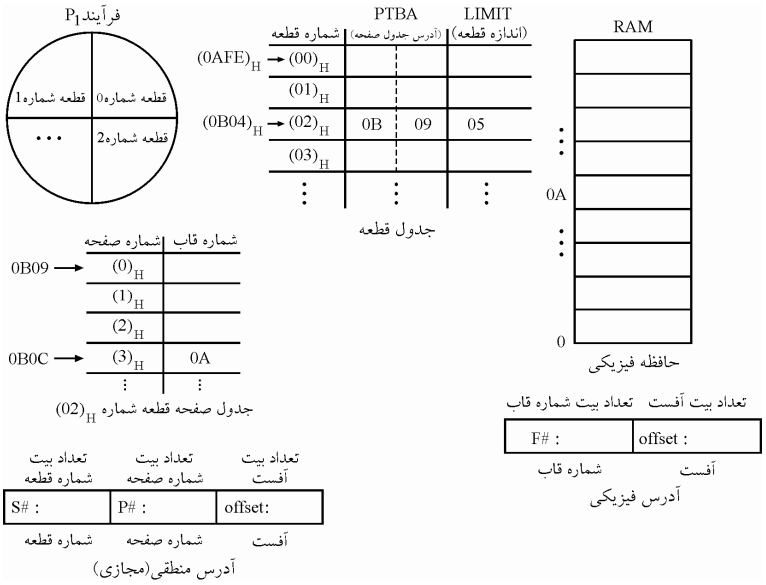
شماره قاب صفر: 0000 0000 ← نگاشت (2 بیت شیفت به راست)

- 00000000 : شماره صفحه 0
- 00000001 : شماره صفحه 1
- 00000010 : شماره صفحه 2
- 00000011 : شماره صفحه 3

۲- گزینه (۲) صحیح است.

در راه‌حل قطعه‌بندی همراه با صفحه‌بندی، قطعات تولید شده توسط برنامه‌نویس یا کامپایلر، توسط سیستم عامل صفحه‌بندی می‌شوند. در این حالت به ازای هر فرآیند یک جدول قطعه و به ازای هر قطعه، یک جدول صفحه ایجاد می‌شود که تعداد درایه‌های جدول قطعه، بستگی به اندازه فرآیند و تعداد قطعات حاصل و تعداد درایه‌های جدول صفحه، بستگی به اندازه قطعه و تعداد صفحات حاصل دارد.

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه‌حل قطعه‌بندی همراه با صفحه‌بندی، به صورت زیر است:



khalilifar.ir

مطابق صورت سوال، آدرس منطقی به صورت [02H,3456H] بیان شده است. در ابتدا از این آدرس منطقی مطرح شده می‌توان متوجه شد که از میان قطعات موجود در یک فرآیند، قطعه شماره $(02)_H$ مورد درخواست است. از آنجا که هر قطعه صفحه‌بندی شده است، پس بخش دوم آدرس منطقی، یعنی (3456H) از #P و offset تشکیل شده‌است، با توجه به اینکه صفحات 4 کیلوبایتی ($4KB = 2^2 \times 2^{10} = 2^{12}$) هستند، پس 12 بیت کم ارزش آن (سه رقم سمت راست هگزا دسیمال) که برابر $(456)_H$ است، بیانگر آفست و $(3)_H$ نشان‌دهنده‌ی شماره صفحه مورد درخواست خواهد بود.

$$\text{تعداد بیت آفست} = b = \log_{2^{\text{اندازه صفحه یا اندازه قاب}}} = \log_2^{2^{12}} = 12bit$$

توجه: 12 بیت در مبنای باینری برابر 3 رقم در مبنای هگزا دسیمال است. هر رقم هگزا دسیمال برابر با 4 رقم باینری است.

توجه: دقت کنید که همه‌ی اعداد به جای دسیمال، به شکل هگزا دسیمال بیان شده است.

با توجه به مطالب بیان شده، آدرس منطقی به صورت زیر خواهد بود:

	تعداد بیت شماره قطعه	تعداد بیت شماره صفحه	تعداد بیت آفست
آدرس منطقی	S# : $(02)_H$	P# : $(3)_H$	offset: $(456)_H$
	شماره قطعه	شماره صفحه	آفست

توجه: از آنجا که تعداد قطعات فرآیند مشخص نشده است، بنابراین، تعداد بیت شماره قطعه قابل تشخیص نیست.

توجه: از آنجا که تعداد صفحات یک قطعه مشخص نشده است، بنابراین، تعداد بیت شماره صفحه قابل تشخیص نیست.

توجه: در این سوال، داشتن اطلاعات مربوط به دو مورد فوق ضروری نیست.

چون در آدرس منطقی داده شده، قطعه شماره $(02)_H$ مورد درخواست است، پس باید از ابتدای جدول قطعه سطر به سطر حرکت کنیم تا به قطعه شماره $(02)_H$ برسیم. مطابق مفروضات مسأله، هر سطر جدول قطعه 3 بایت در نظر گرفته شده است، در صورت سوال، آدرس ابتدای جدول قطعه به صورت $(0AFE)_H$ بیان شده است، مطابق جدول قطعه واضح است که برای کشف آدرس سطر حاوی قطعه شماره‌ی $(02)_H$ باید، از دو سطر جدول قطعه گذر شود، یعنی $2 \times 3 = 6$ بایت از ابتدای جدول قطعه به جلو برویم، بنابراین آدرس سطر حاوی قطعه مورد نظر برابر است با:

$$\begin{array}{r} 0AFE \\ +\quad 6 \\ \hline (0B04)_H \end{array}$$

به آدرس به دست آمده رفته و 3 بایت را خارج می‌کنیم که باتوجه به الگوی داده شده در صورت سؤال، داریم:

khalilifar.ir

بایت اول	بایت دوم	بایت سوم
PTBA		LIMIT
0B	09	05

آدرس جدول صفحه — اندازه قطعه

بنابراین آدرس شروع جدول صفحه برای قطعه شماره (02) برابر $(0B09)_H$ می‌باشد. چون در آدرس منطقی داده شده، صفحه شماره $(3)_H$ مورد درخواست است، پس باید از ابتدای جدول صفحه سطر به سطر حرکت کنیم تا به صفحه شماره $(3)_H$ برسیم، مطابق مفروضات مسأله، هر سطر جدول صفحه 1 بایت در نظر گرفته شده است. مطابق جدول صفحه واضح است که برای کشف آدرس سطر حاوی صفحه شماره $(3)_H$ باید، از سه سطر جدول صفحه گذر شود، یعنی $3 \times 1 = 3$ بایت از ابتدای جدول صفحه به جلو برویم، بنابراین آدرس سطر حاوی صفحه مورد نظر برابر است با:

$$\begin{array}{r} 0B09 \\ +\quad 3 \\ \hline (0B0C)_H \end{array}$$

حال با کشف آدرس فوق، به سطر مورد نظر در جدول می‌رویم، محتوای آدرس $(0B0C)_H$ برابر مقدار $(0A)_H$ می‌باشد، که برابر شماره قاب مورد نظر برای صفحه مورد درخواست است. بنابراین آدرس فیزیکی به صورت زیر خواهد بود:

F#: $(0A)_H$	offset: $(456)_H$

شماره قاب — آفست

آدرس فیزیکی

آدرس فیزیکی $= (0A456)_H$

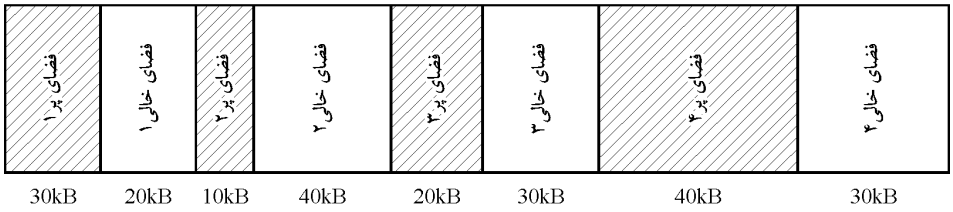
۳- گزینه (۴) صحیح است.

یک فرآیند پس از ورود به حافظه و قرارگیری در صف آماده پردازنده، بر اساس الگوریتم زمان‌بندی پردازنده (زمان‌بندی کوتاه مدت) می‌تواند پردازنده را دریافت کند و اجرا گردد. در این سوال الگوریتم قرارگیری فرآیندها در حافظه Next-Fit در نظر گرفته شده است.

الگوریتم Next-Fit : این روش شبیه به روش First-Fit است، با این تفاوت که جستجو برای یافتن اولین محل مناسب، همواره از ابتدای حافظه آغاز نمی‌شود، بلکه جستجو از محل آخرین تخصیص به بعد شروع می‌شود. این الگوریتم از ابتدای حافظه، شروع به کار می‌کند.

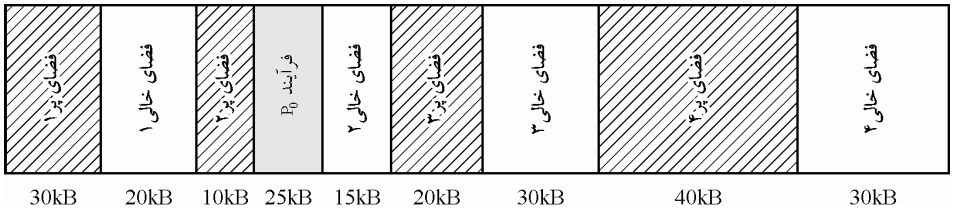
محتویات حافظه قبل از زمان t به صورت زیر است:

khalilifar.ir

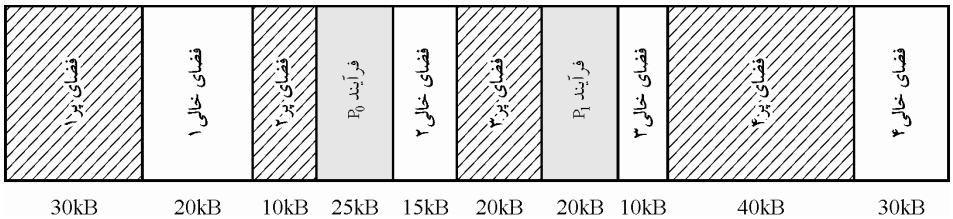


توجه: مطابق وعده سؤال، فقط فضای پُر 3 در لحظه t+20 آزاد می‌گردد و مابقی فضاهای پُر تا اتمام اجرای فرآیندهای مطرح شده، آزاد نمی‌گردد.

در لحظه t، فرآیند P_0، علاقه‌مندی خود را برای ورود به حافظه اعلام می‌کند، میزان حافظه مورد نیاز این فرآیند 25KB است، پس از بررسی واحد مدیریت حافظه توسط الگوریتم Next-Fit مشخص می‌شود این فضا به صورت همجوار موجود است. بنابراین با ورود این فرآیند به حافظه و قرارگیری در صف آماده پردازنده موافقت می‌گردد. بنابراین شکل حافظه به صورت زیر خواهد بود:



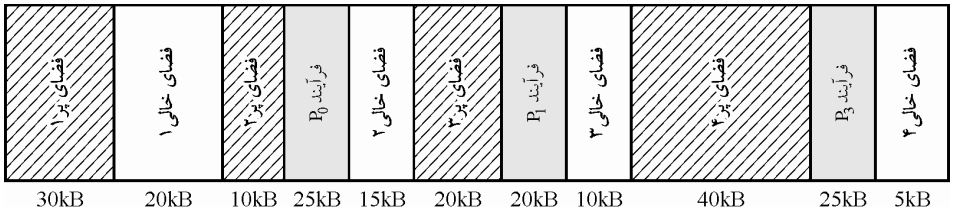
در لحظه t+1، فرآیند P_1، علاقه‌مندی خود را برای ورود به حافظه اعلام می‌کند، میزان حافظه‌ی مورد نیاز این فرآیند 20KB است، پس از بررسی واحد مدیریت حافظه توسط الگوریتم Next-Fit مشخص می‌شود این فضا به صورت همجوار موجود است. بنابراین با ورود این فرآیند به حافظه و قرارگیری در صف آماده پردازنده موافقت می‌گردد. فضای خالی پس از آخرین تخصیص، فضای خالی 2 است، اما کافی نیست، پس جلوتر می‌رویم تا به فضای خالی 3 برسیم که کافی است، بنابراین شکل حافظه به صورت زیر خواهد بود:



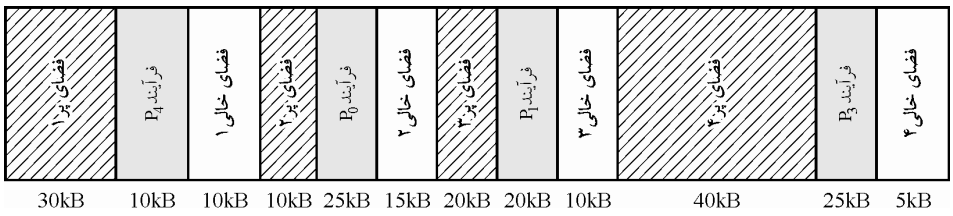
در لحظه t+2، فرآیند P_2، علاقه‌مندی خود را برای ورود به حافظه اعلام می‌کند، میزان حافظه‌ی مورد نیاز این فرآیند 40KB است، پس از بررسی واحد مدیریت حافظه توسط الگوریتم Next-Fit

khalilifar.ir

مشخص می‌شود این فضا به صورت همجوار موجود نیست. بنابراین با ورود این فرآیند به حافظه و قرارگیری در صف آماده پردازنده موافقت نمی‌گردد. بنابراین فرآیند P_2، باید منتظر بماند تا حافظه‌ی مورد نیازش آزاد گردد. اولین آزادسازی حافظه در لحظه t+20 به میزان 20KB می‌باشد که مربوط به فضای پُر 3 می‌باشد، بنابراین فرآیند P_2 فعلاً تا آن لحظه باید صبر کند. در لحظه t+3 فرآیند P_3، علاقه‌مندی خود را برای ورود به حافظه اعلام می‌کند، میزان حافظه مورد نیاز این فرآیند 25KB است، پس از بررسی واحد مدیریت حافظه توسط الگوریتم Next-Fit مشخص می‌شود این فضا به صورت همجوار موجود است. بنابراین با ورود این فرآیند به حافظه و قرارگیری در صف آماده پردازنده موافقت می‌گردد. فضای خالی پس از آخرین تخصیص، فضای خالی 3 است، اما کافی نیست، پس جلوتر می‌رویم تا به فضای خالی 4 برسیم که کافی است، بنابراین شکل حافظه به صورت زیر خواهد بود:



در لحظه t+4، فرآیند P_4، علاقه‌مندی خود را برای ورود به حافظه اعلام می‌کند، میزان حافظه مورد نیاز این فرآیند 10KB است، پس از بررسی واحد مدیریت حافظه توسط الگوریتم Next-Fit مشخص می‌شود این فضا به صورت همجوار موجود است. بنابراین با ورود این فرآیند به حافظه و قرارگیری در صف آماده پردازنده موافقت می‌گردد. فضای خالی پس از آخرین تخصیص، فضای خالی 4 است، اما کافی نیست، از آنجا که الگوریتم به انتهای حافظه رسیده است و امکان جلوتر رفتن وجود ندارد، بنابراین الگوریتم دور زده و مجدداً از ابتدای حافظه شروع به کار می‌کند. اولین فضای خالی در ابتدای حافظه فضای خالی 1 می‌باشد که کافی است. بنابراین شکل حافظه به صورت زیر خواهد بود:



در لحظه t+5، فرآیند P_5، علاقه‌مندی خود را برای ورود به حافظه اعلام می‌کند، میزان حافظه مورد نیاز این فرآیند 35KB است، پس از بررسی واحد مدیریت حافظه توسط الگوریتم Next-Fit مشخص می‌شود این فضا به صورت همجوار موجود نیست. بنابراین با ورود این فرآیند

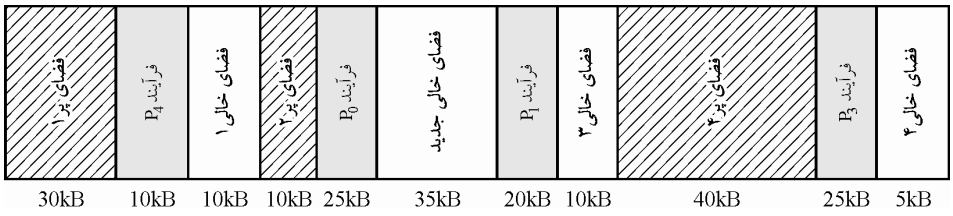
khalilifar.ir

به حافظه و قرارگیری در صف آماده پردازنده موافقت نمی‌گردد. بنابراین فرآیند P_5 نیز در کنار فرآیند P_2، باید منتظر بماند تا حافظه مورد نیازش آزاد گردد. اولین آزادسازی حافظه در لحظه t+20 به میزان 20KB می‌باشد که مربوط به فضای پُر 3 می‌باشد، بنابراین فرآیند P_5 فعلاً تا آن لحظه باید صبر کند.

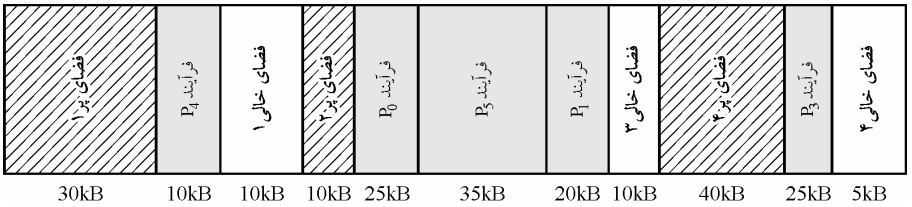
توجه: در حال حاضر فرآیندهای P_0، P_1، P_3 و P_4 در حافظه و در صف آماده پردازنده قرار دارند و مطابق الگوریتم FCFS، به نوبت اجرا خواهند شد.

20 ثانیه بعد...

هم اکنون در لحظه t+20 قرار داریم، در این لحظه 20 ثانیه از اجرای فرآیند P_0 گذشته است و همچنین بالاخره فضای پُر3 مطابق وعده سؤال آزاد می‌گردد. این فضای آزاد شده 20KB، با فضای خالی 2 به میزان 15KB ادغام شده و یک فضای 35KB همجوار را ایجاد می‌کند. بنابراین از بین فرآیندهای منتظر ورود به حافظه یعنی فرآیندهای P_2 و P_5، فرصت برای ورود فرآیند P_5 به حافظه و قرارگیری در صف آماده فراهم می‌گردد. و می‌رود در انتهای صف آماده پردازنده و بعد از فرآیندهای P_0، P_1، P_3 و P_4 قرار می‌گیرد. بنابراین شکل حافظه به صورت زیر خواهد بود:



در لحظه t+20 فرآیندهای P_2 و P_5 هر دو علاقه‌مندی خود را به ترتیب برای ورود به حافظه اعلام می‌کنند، میزان حافظه مورد نیاز این فرآیندها به ترتیب 40KB و 35KB است، پس از بررسی واحد مدیریت حافظه توسط الگوریتم Next-Fit مشخص می‌شود این فضا به صورت همجوار فقط برای فرآیند P_5 موجود است. بنابراین با ورود فرآیند P_5 به حافظه و قرارگیری در صف آماده پردازنده موافقت و با ورود فرآیند P_2 مخالفت می‌گردد. فضای خالی پس از آخرین تخصیص فضای خالی 1 است، که کافی نیست. پس جلوتر می رویم تا به فضای خالی همجوار جدید، حاصل از ادغام دو فضای خالی 2 و 3، به اندازه 35KB برسیم که کافی است. بنابراین شکل حافظه به صورت زیر خواهد بود:



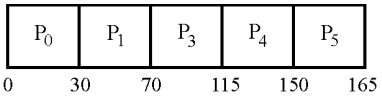
khalilifar.ir

در صورت خروج فرآیندهای P_0، P_1، P_3 و P_4 همچنان فضای آزاد کافی برای ورود فرآیند P_2 به حافظه فراهم نمی‌گردد و فقط با خروج فرآیند P_5 است که فضای آزاد کافی برای ورود فرآیند P_2 فراهم می‌گردد.

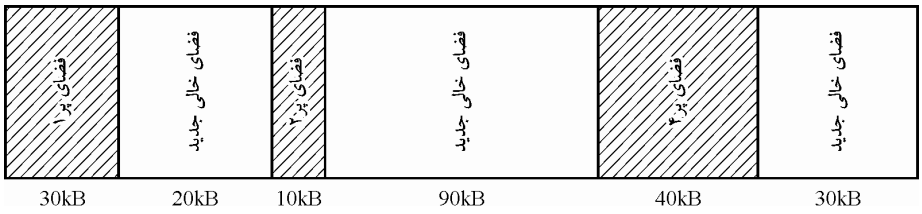
توجه: در حال حاضر فرآیندهای P_0، P_1، P_3، P_4 و P_5 در حافظه و در صف آماده پردازنده قرار دارند و مطابق الگوریتم FCFS، به نوبت اجرا خواهند شد.

165 ثانیه بعد...

نمودار گانت اولیه فرآیندها، براساس زمان اجرای فرآیندها و الگوریتم FCFS به صورت زیر است:



هم اکنون در لحظه t+165 قرار داریم، در این لحظه فرآیندهای P_0، P_1، P_3، P_4 و P_5 به پایان کار خود رسیده‌اند، بنابراین فرصت برای ورود فرآیند P_2 به حافظه و قرارگیری در صف آماده پردازنده فراهم می‌گردد و می‌رود به صورت تنها در ابتدای صف آماده قرار می‌گیرد، زیرا مابقی فرآیندها به دلیل اتمام کار خود از حافظه و صف آماده پردازنده خارج شده‌اند. بنابراین شکل حافظه به صورت زیر خواهد بود:



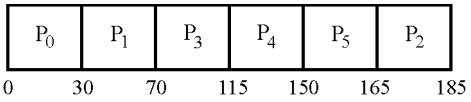
تا قبل از زمان t+165 هرفضایی که به واسطه اتمام فرآیندها آزاد می‌گردد، کل فضای حافظه برای بررسی فضای کافی برای ورود فرآیند P_2 توسط الگوریتم Next-Fit و از آخرین تخصیص مورد بررسی قرار می‌گیرد، اما هیچگاه این الگوریتم موفق به کشف این فضای همجوار کافی برای ورود فرآیند P_2 نمی‌گردد. اما درست در لحظه t+165 اتفاق دیگری می‌افتد.....

در لحظه t+165، فرآیند P_2 پس از آن همه ناکامی و تلاش برای ورود به حافظه و قرارگیری در صف آماده، امیدوارانه باز هم شناس خود را امتحان می‌کند و علاقه‌مندی خود را برای ورود به حافظه باز هم به سیستم اعلام می‌کند، میزان حافظه مورد نیاز این فرآیند 40KB است، پس از بررسی مجدد واحد مدیریت حافظه توسط الگوریتم Next-Fit به صورت غیرمنتظره‌ای مشخص می‌شود که این فضا به صورت همجوار موجود است. بنابراین با ورود این فرآیند به حافظه و قرارگیری در صف آماده پردازنده موافقت می‌گردد و سرانجام فرآیند P_2 در تلاش برای ورود به حافظه و قرارگیری در صف آماده پردازنده، موفق می‌شود.....

khalilifar.ir



نمودار گانت نهایی فرآیندها، براساس زمان اجرای فرآینـدها و الگـوریتم FCFS بـه صـورت زیـر است:



توجه: برای سادگی زمان t را برابر صفر در نظر بگیرید، بنابراین داریم:

فرآیند	زمان ورود	حافظه مورد نیاز	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_0	0	25	30	0	30
P_1	1	20	40	29	69
P_2	2	40	20	163	183
P_3	3	25	45	67	112
P_4	4	10	35	111	146
P_5	5	35	15	145	160

زمان ورود فرآیند- زمان خروج فرآیند= زمان بازگشت فرآیند

P_0 زمان بازگشت = 30–0=30

P_1 زمان بازگشت =70–1=69

P_2 زمان بازگشت =185–2=183

P_3 زمان بازگشت =115–3=112

P_4 زمان بازگشت =150–4=146

P_5 زمان بازگشت =165–5=160

$$\text{میانگین زمان بازگشت} = \frac{30+69+183+112+146+160}{6} = \frac{700}{6} = 116.67$$

زمان اجرای فرآیند - زمان بازگشت فرآیند= زمان انتظار فرآیند

P_0 زمان انتظار =30–30=0

P1 زمان انتظار =69–40=29

P_2 زمان انتظار =183–20=163

khalilifar.ir

P_3 زمان انتظار =112–45=67

P_4 زمان انتظار =146–35=111

P_5 زمان انتظار =160–15=145

$$\text{میانگین زمان انتظار} = \frac{0+29+163+67+111+145}{6} = \frac{515}{6} = 85.83$$

$$\text{میانگین زمان اجرا} = \frac{30+40+20+45+35+15}{6} = \frac{185}{6} = 30.83$$

میانگین زمان انتظار + میانگین زمان اجرا= میانگین زمان بازگشت

116.67 = 30.83 + 85.83

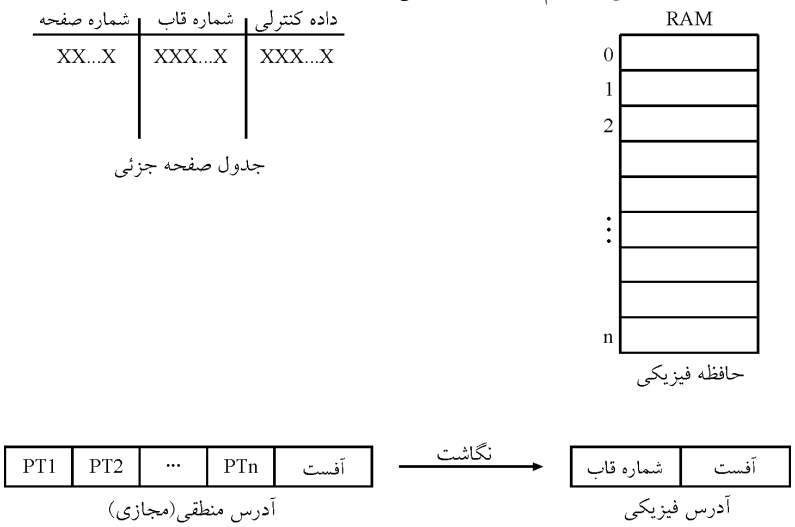
توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد، که فقط در گزینه اول، سوم و چهارم این مورد رعایت شده است. بنابراین گزینهٔ دوم پاسخ سؤال نیست.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است، بنابراین از این نگاه همه گزینه‌ها درست هستند، که در اینجا این نگاه کارآمد نیست.

۴- گزینه (۳) صحیح است.

در اینجا برای جدول صفحه جزئی محدودیتی به اندازه یک قاب داریم.

بنابراین اندازه جدول صفحه جزئی برابر اندازه قاب می‌باشد. بنابراین برای محاسبه تعداد سطرهای جدول صفحه جزئی، کافی است، اندازه قاب که برابر اندازه جدول صفحه جزئی است بر اندازه عرض جدول صفحه جزئی تقسیم گردد. به شکل زیر توجه کنید:



khalilifar.ir

توجه: عرض جدول صفحه همواره برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب است، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد.

بنابراین داریم:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب = عرض جدول صفحه جزئی

10 + تعداد بیت‌های شماره قاب = عرض جدول صفحه جزئی

توجه: 10 بیت کنترلی داریم، مطابق فرض صورت سوال.

تعداد بیت شماره قاب = $\log_2^{\text{تعداد قاب‌های حافظه فیزیکی}}$

تعداد قاب‌های حافظه‌ی فیزیکی = $\frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه قاب}} = \frac{2^{32}B}{2^{10}B} = 2^{22}$

بنابراین:

تعداد بیت شماره قاب = $\log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2 2^{22} = 22$

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب = عرض جدول صفحه جزئی

عرض جدول صفحه جزئی = 22 + 10 = 32bit = 4Byte

تعداد سطرهای جدول صفحه جزئی = $\frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^{10}B}{2^{2}B} = 2^{8} = 256$

اندازه فرآیند (فضای آدرس مجازی) = $2^{40}B$

اندازه صفحه = اندازه قاب = $2^{10}B$

تعداد صفحات فرآیند: f = $\frac{\text{اندازه فرآیند}}{\text{اندازه صفحه}} = \frac{2^{40}}{2^{10}} = 2^{30}$

تعداد سطرهای جدول صفحه جزئی: r = 256 = 2^8

حال اطلاعات کافی برای محاسبه تعداد سطوح جدول صفحه چند سطحی را در اختیار داریم:

روش تجزیه

تعداد صفحات فرآیند باید در اندازه r (2^8) تجزیه گردد:

تعداد صفحات فرآیند = $2^{30} = 2^{⑥} \times 2^{⑧} \times 2^{⑧} \times 2^{⑧}$ (PT1 PT2 PT3 PT4)

توجه: تعداد عملوندها برابر 4 است، بنابراین تعداد سطوح جدول چند سطحی نیز برابر 4 است.

khalilifar.ir

روش لگاریتم

$$\text{تعداد سطوح جدول چند سطحی} = d = \left\lceil \log_r^f \right\rceil = \left\lceil \log_{2^8}^{2^{30}} \right\rceil = 4$$

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح چهارم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{2^{30}}{2^8} = 2^{22}$$

$$\text{تعداد جداول صفحه جزئی در سطح سوم} = \frac{\text{تعداد جداول صفحه جزئی در سطح چهارم}}{r} = \frac{2^{22}}{2^8} = 2^{14}$$

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد جداول صفحه جزئی در سطح سوم}}{r} = \frac{2^{14}}{2^8} = 2^6$$

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{r} = \frac{2^6}{2^8} < 1$$

توجه: سطح اول، یک جدول به حساب می‌آید، که 2^6 سطر بیشتر نیاز ندارد.

توجه: تعداد تقسیم متوالی برابر 4 است، بنابراین تعداد سطوح جدول چند سطحی برابر 4 است.

$$\text{تعداد بیت آفست} = \log_2^{\text{اندازه صفحه}} = \log_2^{2^{10}} = 10 \text{ بیت}$$

بنابراین شکل آدرس منطقی (مجازی) به صورت زیر خواهد بود:

PT1	PT2	PT3	PT4	آفست
6 بیت	8 بیت	8 بیت	8 بیت	10 بیت
30 بیت				

۵- گزینه (۳) صحیح است.

در تکنیک صفحه‌بندی، برای هر فرآیند یک جدول صفحه معمولی تشکیل می‌شود که در آن به ازای همه صفحه‌های یک فرآیند، درایه وجود دارد و هر درایه مشخص می‌کند که کدام قاب فیزیکی به این صفحه اختصاص یافته است. در این روش وقتی فرآیندها بزرگ باشند، هزینه نگهداری جداول صفحه بسیار زیاد می‌شود، در ضمن به ازای هر فرآیند نیز باید یک جدول صفحه داشته باشیم. به عبارتی وقتی تعداد و اندازه فرآیندها بزرگ شود، این روش برای ترجمه آدرس مجازی به فیزیکی مقرون به صرفه نیست.

برای حل این مشکل **جدول صفحه چندسطحی** و **جدول صفحه معکوس** ابداع شده است. جدول صفحه معکوس از نظر سربار حافظه بهتر از روش جدول صفحه چندسطحی است و حالت

khalilifar.ir

حداقلی است. اغلب سیستم‌های کامپیوتری، فضای آدرس منطقی(مجازی) بزرگی را پشتیبانی می‌کنند، مانند کامپیوترهایی که آدرس‌های 32 یا 64 بیتی را پشتیبانی می‌کنند، در چنین محیط‌هایی جدول صفحه معمولی بسیار بزرگ خواهد شد. برای مثال یک سیستم را با 32 بیت فضای آدرس منطقی(مجازی) در نظر بگیرید. اگر اندازه صفحه در این سیستم برابر با 4KB (2^{12}) باشد، در اینصورت جدول صفحه شامل بیش از یک میلیون درایه خواهد بود. ($2^{32} / 2^{12} = 2^{20}$). فرض کنید هر درایه جدول صفحه معمولی، شامل 4 بایت باشد، در اینصورت هر فرآیند به (4×2^{20}) 4MB فضای آدرس فیزیکی فقط برای نگهداری جدول صفحه معمولی نیاز دارد. همچنین مشخص است که با توجه به محدودیت اندازه قاب (4KB) نمی‌توان این جدول صفحه معمولی را بصورت پیوسته درون یک قاب جا داد. 4MB اندازه جدول صفحه معمولی در فضای 4KB مربوط به یک قاب جا نمی‌شود. یک راه حل تقسیم جدول صفحه معمولی به جداول صفحه جزئی و ایجاد جدول صفحه چندسطحی است، در این حالت جدول صفحه معمولی نیز همانند فرآیند صفحه‌بندی می‌شود. راه حل دیگر استفاده از جدول صفحه معکوس است. در راه حل جدول صفحه معکوس به جای اینکه در جداول صفحه مربوط به هر فرآیند، به ازای هر صفحه مجازی یک درایه داشته باشیم، به ازای هر قاب در حافظه فیزیکی یک درایه در جدول صفحه معکوس نگهداری می‌کنیم. در واقع به ازای هر قاب حافظه اصلی اینکه در حال حاضر کدام صفحه مربوط به کدام فرآیند در این قاب ذخیره شده است نگهداری می‌شود. **بنابراین گزینه سوم نادرست است.** زیرا در جدول صفحه وارونه (معکوس)، فقط اطلاعات فضای آدرس منطقی برای یک پردازه (فرآیند) در دسترس نیست، بلکه اطلاعات فضای آدرس مربوط به **صفحات حاضر** در حافظه برای **همه** پردازه‌ها (فرآیندها) در دسترس است، اینکه کدام صفحه مربوط به کدام فرآیند در قاب‌های حافظه ذخیره شده است. همچنین تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی (اصلی) است. یعنی به ازای هر صفحه فیزیکی حافظه در جدول صفحه وارونه تنها یک خانه وجود دارد، **بنابراین گزینه اول درست است.**

مثال: در یک سیستم صفحه‌بندی که دارای 34 بیت آدرس است 23 بیت اول برای شماره صفحه و 11 بیت بعدی برای آدرس‌دهی درون صفحه است. در یک سیستم با صفحه‌بندی معکوس(Inverted Page Table) با 128 مگابایت حافظه، جدول صفحه دارای چند خانه (درایه یا مدخل) است؟

پاسخ: تعداد درایه‌های جدول صفحه معکوس، مطابق رابطه زیر محاسبه می‌گردد:

تعداد قاب‌های حافظه فیزیکی = تعداد درایه‌های جدول صفحه معکوس

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

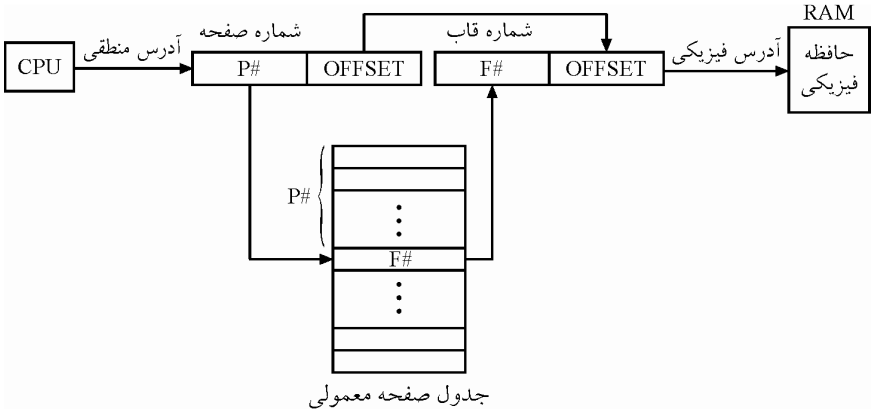
khalilifar.ir

همانطور که گفتیم تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی است، بنابراین تعداد درایه‌های جدول صفحه معکوس به صورت زیر خواهد بود:

$$\text{تعداد درایه‌های جدول صفحه معکوس} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

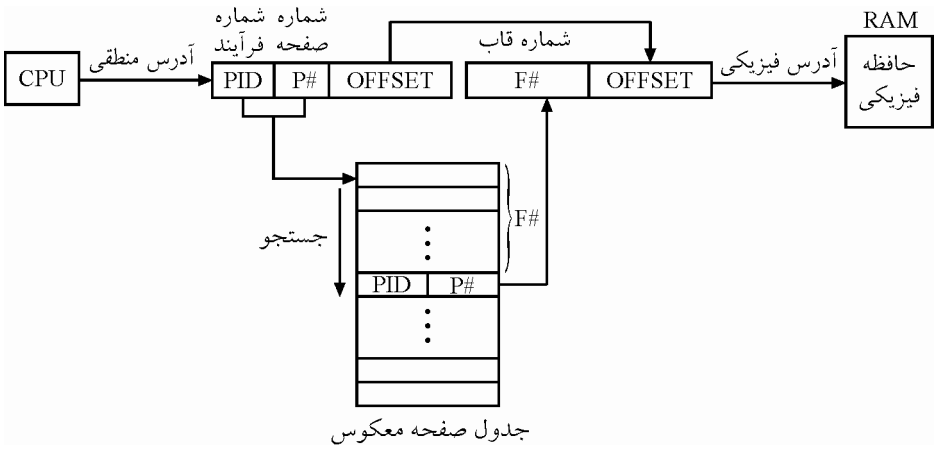
در یک عبارت ساده، هدف از استفاده از جدول صفحه معکوس، **کاهش** میزان حافظه فیزیکی مورد نیاز برای ترجمه آدرس مجازی به فیزیکی است. با استفاده از تکنیک جدول صفحه معکوس، مقدار زیادی در حافظه فیزیکی صرفه‌جویی می‌شود اما تبدیل آدرس مجازی به فیزیکی کندتر و وقت‌گیرتر می‌شود. در واقع **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم. زیرا جدول صفحه معکوس بر اساس #F اندیس شده است و نه مانند جدول صفحه معمولی که بر اساس #P اندیس شده است. در واقع وقتی یک فرآیند با PID مختص به خود به صفحه مجازی #P مراجعه می‌کند، سخت‌افزار دیگر نمی‌تواند از #P به عنوان اندیس جدول صفحه استفاده کند و صفحه فیزیکی را بیابد و از این جهت باید سرتاسر جدول صفحه معکوس(وارونه) را برای یافتن درایه (PID,P#) از آدرس مجازی (PID,P#,OFFSET) جستجو کند. اگر یک تطبیق پیدا شود، در اینصورت آدرس فیزیکی(F#,OFFSET) تولید خواهد شد و اگر هیچ تطبیقی یافته نشود، در اینصورت یک دسترسی آدرس غیر مجاز می‌باشد. ضمن اینکه این جستجو باید به ازای هربار دسترسی به حافظه انجام شود. همانطور که گفتیم در جدول صفحه معکوس **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم، یعنی سرعت نگاشت آدرس منطقی به فیزیکی **کاهش** می‌یابد. درواقع جدول صفحه معکوس **زمان** نگاشت آدرس منطقی به آدرس فیزیکی را **افزایش** می‌دهد. هرچند موجب صرفه‌جویی در مصرف حافظه فیزیکی می‌شود.

نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معمولی** به صورت زیر است:



khalilifar.ir

نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معکوس** به صورت زیر است:

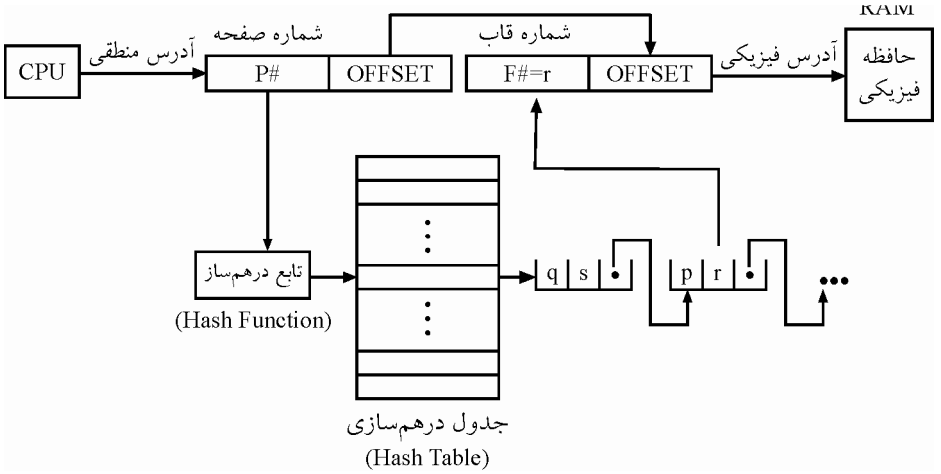


بنابراین مطابق شکل فوق **گزینه چهارم درست است.**

جدول صفحه درهم‌سازی شده

یک راه حل عمومی برای نگاشت فضای آدرس بزرگ به کوچک، مثلا نگاشت فضای آدرس 64 بیتی به 32 بیتی، در سیستمی که فضای آدرس 64 بیتی را پشتیبانی نمی‌کند و فضای 32 بیتی را پشتیبانی می‌کند، استفاده از جدول صفحه درهم‌سازی شده است. تابع درهم‌ساز حجم زیادی از داده را به یک عدد طبیعی تبدیل می‌کند. عدد طبیعی حاصل از تابع درهم‌سازی معمولاً به عنوان اندیس یک آرایه مورد استفاده قرار می‌گیرد. به مقادیر حاصل از تابع درهم‌ساز مقدار درهم (Hashed Value) گفته می‌شود. به عبارت دیگر در اینجا تابع درهم‌ساز مقدار #P از یک آدرس مجازی بزرگ مربوط به یک فرآیند را می‌گیرد و پس از تقسیم بر عدد تابع درهم‌ساز، باقی‌مانده حاصل درایه مشخصی را در جدول درهم‌سازی نشان می‌دهد. هر درایه در جدول درهم‌سازی شامل یک لیست پیوندی از باقی‌مانده‌های یکسان حاصل از تابع درهم‌ساز است. به عبارت دیگر همه صفحات مجازی یک فرآیند که مقدار درهم یکسانی دارند تشکیل یک لیست پیوندی می‌دهند در جلوی یک درایه مشخص از جدول درهم‌سازی. هر گره لیست پیوندی شامل سه فیلد (۱) شماره صفحه مجازی، (۲) شماره قاب و (۳) یک اشاره‌گر به گره بعدی لیست پیوندی است. نحوه کار بدین صورت است که پس از آنکه درایه مرتبط با #P مورد نظر توسط تابع درهم‌ساز، در جدول درهم‌سازی مشخص شد، جستجو در جهت کشف شماره قاب مورد انتظار آغاز می‌شود، بدین صورت که #P مورد نظر با فید اول در گره اول لیست پیوندی مقایسه می‌شود، اگر مطابقت داشت، شماره قاب مورد نظر از فیلد دوم در گره اول جهت ساخت آدرس فیزیکی استخراج می‌شود، در غیر اینصورت گره‌های بعدی جهت کشف شماره قاب مورد انتظار در لیست پیوندی مورد جستجو قرار می‌گیرد. شکل زیر گویای مطلب است:

khalilifar.ir



توجه: برای افزایش سرعت تبدیل آدرس مجازی به فیزیکی در جدول صفحه معکوس می‌توان از تکنیک TLB و یا تفکر جدول درهم‌سازی (Hashe Table) با اندکی تغییر استفاده نمود. بنابراین هر ترجمه آدرس مجازی به فیزیکی نیازمند دوبار مراجعه به حافظه فیزیکی می‌باشد، یکبار برای دسترسی به جدول درهم‌سازی و یکبار دیگر برای دسترسی به جدول صفحه معکوس.

توجه: جدول صفحه معکوس، اطلاعات کاملی در مورد فضای آدرس منطقی یک فرآیند را ندارد. در حالی که سیستم صفحه‌بندی مبتنی بر تقاضا به این اطلاعات کامل جهت پردازش نقص‌های صفحه نیازمند است. برای دسترسی به این اطلاعات کامل، یک **جدول صفحه خارجی**، یکی به ازای هر فرآیند، باید نگهداری شود. جدول صفحه خارجی شامل فیلدهای Valid، Present و آدرس صفحه مجازی دچار نقص صفحه شده بر روی رسانه ذخیره‌سازی است. **بنابراین گزینه دوم درست است.**

توجه: هنگامی که یک نقص صفحه رخ می‌دهد، ممکن است یک **نقص صفحه دیگر** برای بارگذاری **جدول صفحه خارجی** از رسانه ذخیره‌سازی به حافظه فیزیکی رخ دهد. بنابراین در جدول صفحه معکوس، زمان سرویس نقص صفحه (page fault) به دلیل ایجاد یک نقص صفحه دیگر، **افزایش** می‌یابد.

توجه: در سیستم‌هایی که از جدول صفحه معکوس‌شده استفاده می‌کنند، استفاده اشتراکی از صفحات میان فرآیندها در حالت عادی امکان‌پذیر نیست، در حالت کلی اشتراک به شکل چند به یک است، یعنی صفحه مشترک میان فرآیندها در یک قاب مشخص قرار می‌گیرد. که پیاده‌سازی حالت چند به یک در جدول صفحه معکوس درحالت ابتدایی امکان‌پذیر نیست. زیرا در جدول صفحه معکوس برای هر درایه آن فقط می‌توان شماره یک صفحه از یک فرآیند را قرار داد. البته توسط مکانیزم‌هایی می‌توان اشتراک‌گذاری صفحات میان فرآیندها را در جدول صفحه معکوس نیز

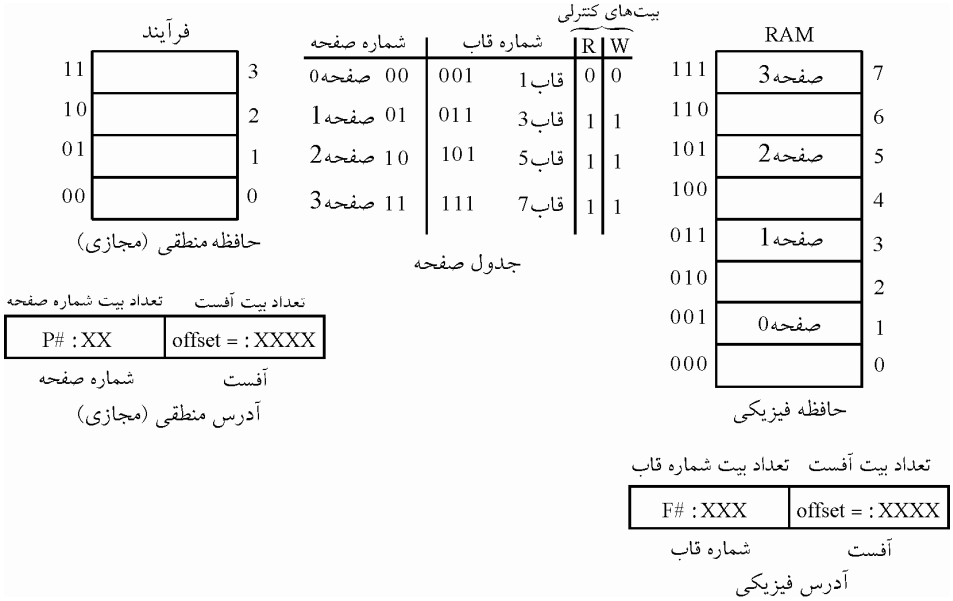
khalilifar.ir

پیاده‌سازی نمود. برای مثال به جدول صفحه معکوس اجازه دهیم برای هر درایه، بیش از یک شماره صفحه را آدرس‌دهی کند.

۶- گزینه (۳) صحیح است.

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بندی به صورت زیر است:

مثال: فرض کنید اندازه‌ی هر صفحه 16 بایت، اندازه فرآیند 64 بایت و اندازه حافظه فیزیکی برابر 128 بایت باشد.



$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{64}{16} = 4$$

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{128}{16} = 8$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^4 = 2\text{bit}$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^8 = 3\text{bit}$$

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{16} = 4\text{bit}$$

khalilifar.ir

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد، به صورت زیر:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب = عرض جدول صفحه

در جدول فوق 2 بیت مربوط به بیت‌های کنترلی و 3 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس عرض جدول صفحه فوق برابر $3\text{bit} + 2\text{bit} = 5\text{bit}$ می‌باشد.

همانطور که گفتیم، اندازه جدول صفحه، از رابطه زیر محاسبه می‌گردد:

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

که مطابق رابطه فوق داریم:

$$\text{اندازه جدول صفحه} = 4 \times 5\,\text{bit} = 20\,\text{bit}$$

همچنین، اندازه آدرس‌های منطقی (مجازی) و فیزیکی به صورت زیر است:

$$\text{اندازه آدرس منطقی} = \text{تعداد بیت شماره صفحه} + \text{تعداد بیت آفست} = 4\text{bit} + 2\text{bit} = 6\,\text{bit}$$

$$\text{اندازه آدرس فیزیکی} = \text{تعداد بیت شماره قاب} + \text{تعداد بیت آفست} = 4\text{bit} + 3\text{bit} = 7\,\text{bit}$$

همچنین داریم:

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{\text{تعداد بیت آدرس منطقی}} = 2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^6 = 2^2 \times 2^4 = 2^6 = 64\,\text{Byte}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{\text{تعداد بیت آدرس فیزیکی}} = 2^{\text{تعداد بیت شماره قاب}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^7 = 2^3 \times 2^4 = 2^7 = 128\ \text{Byte}$$

توجه: بنابراین اندازه آدرس منطقی (مجازی) و فیزیکی (حقیقی) الزاماً برابر نیست. پس گزینه‌ی اول نادرست است.

گزینه‌ی دوم نادرست است. زیرا، عمل تعویض متن (مابین فرآیندها یا نخ‌ها) توسط بخشی از سیستم عامل به نام Dispatcher صورت می‌گیرد. در واقع Dispatcher ماژولی است که کنترل پردازنده را به فرآیندی که توسط زمان‌بند کوتاه‌مدت انتخاب شده است، اعطا می‌کند. همچنین بارگذاری و ذخیره‌سازی PCB و یا به نخی که توسط زمان‌بند نخ (در سطح هسته یا ترکیبی) انتخاب شده است، اعطا می‌کند. همچنین بارگذاری و ذخیره‌سازی TCB.

توجه: نخی که توسط زمان‌بند سطح کاربر، زمان‌بندی می‌شود، توسط سیستم عامل شناخته نمی‌شود، در چنین شرایطی سیستم عامل فقط فرآیند را می‌شناسد.

khalilifar.ir

گزینه‌ی سوم درست است. زیرا، اگر یک نخ (ریسه)، cpu – bound باشد، بهتر است اولویت آن برای دسترسی به I/O بالاتر از نخ‌های I/O – bound باشد، چون زودتر I/O را رها می‌کند و برای استفاده بهتر از پردازنده اقدام می‌کند.

گزینه‌ی چهارم نادرست است. زیرا garbage (زباله) از مشکلات حاصل از استفاده از اشاره‌گرها ایجاد می‌گردد، اما Fragmentation (تکه‌تکه شدن) که بر دو نوع تکه‌تکه شدن داخلی و خارجی است، به دلیل روش‌های مدیریت حافظه ایجاد می‌گردد.

تکه‌تکه شدن داخلی زمانی رخ می‌دهد که حافظه از قبل مرزبندی شده باشد و در هر قسمت بتوان فقط یک قطعه را قرار داد. در این حالت چون احتمالاً قطعه‌ای که برای یک جایگاه انتخاب شده است قدری از آن کوچک‌تر است، در انتهای جایگاه مقداری فضای بدون استفاده به وجود خواهد آمد. به این مشکل تکه‌تکه شدن داخلی گویند.

اما وقتی از قبل مرزبندی مشخصی برای حافظه در نظر نگیریم، احتمال وقوع تکه تکه شدن خارجی وجود دارد. در این حالت احتمالاً فضاهای خالی و قابل استفاده حافظه به صورت جایگاه‌های کوچک، لابه‌لای فرآیندها پخش شده‌اند. که مجموع این جایگاه‌های کوچک پراکنده احتمالاً نیاز ما را برآورده می‌کنند، اما چون همجوار نیستند، بدون استفاده باقی می‌مانند. روش فشرده‌سازی (compaction) در صورت اجرا شدن، این مشکل را برطرف می‌کند. البته در روش قطعه‌بندی نیز با وجود تخصیص غیرهمجوار ولی به دلیل عدم مرزبندی باز هم تکه‌تکه شدن خارجی وجود دارد.

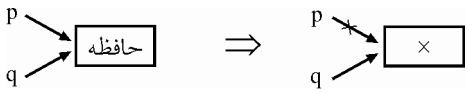
مشکلات اشاره‌گرها

به طور کلی کار کردن با اشاره‌گرها ممکن است باعث ایجاد مشکلاتی گردد این مشکلات عبارتند از:

۱- ارجاع معلق (Dangling Reference): به این معناست که شیء داده‌ای وجود ندارد ولی مسیر دستیابی به آن معتبر است. این مشکل زمانی پیش می‌آید که فضایی که چندین اشاره‌گر به صورت همزمان به آن اشاره می‌کنند توسط یکی از اشاره‌گرها آزاد شده اما بقیه اشاره‌گرها استفاده شوند و از بین نروند.

مثال:

```
int  *p,*q;
p=malloc(20);
q=p
free(p);
```



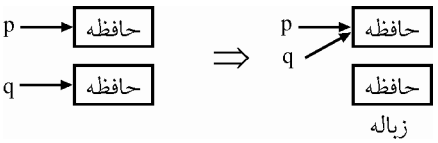
Dangling Reference

۲- زباله (Garbage): به این معناست که شیء داده‌ای وجود دارد ولی هیچ‌گونه امکان دسترسی به آن وجود ندارد. این مشکل زمانی رخ می‌دهد، که فضایی که به یک شیء داده‌ای اختصاص‌یافته است آزاد نشده اما اشاره‌گر به آن شی از بین رفته است.

khalilifar.ir

مثال:

```
int   *p, *q;
p = malloc (20) ;
q = malloc (20);
q = p;
```

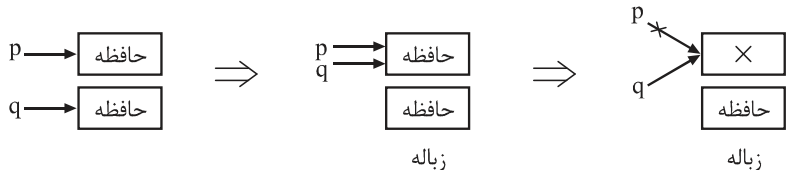


مشکل Garbage

توجه: گاهی ممکن است این دو مشکل به صورت همزمان نیز اتفاق بیفتد.

مثال:

```
int   *p, *q;
p = malloc (20) ;
q = malloc (20);
q = p;
free (p);
```



مشکل Garbage و Dangling Reference به صورت همزمان

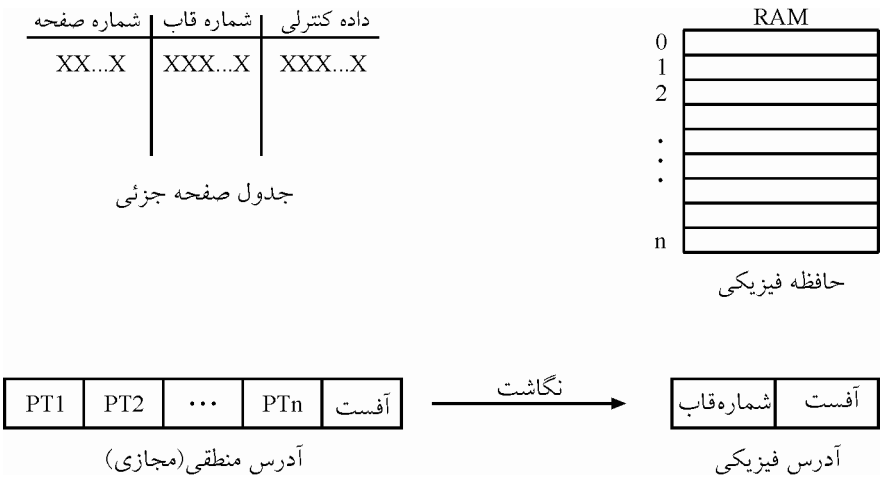
توجه: یک راه حل برای مشکل زباله (Garbage) این است که هرگاه به یک اشاره‌گر فضایی را اختصاص دادیم، به طور اتوماتیک یک فیلد اضافی به نام Reference Count در نظر بگیریم که نشان‌دهنده تعداد اشاره‌گرها به آن محل از حافظه است. هرگاه یک اشاره‌گر جدید به این محل اشاره کند مقدار Count یکی افزایش یافته و هرگاه یک اشاره‌گر آزاد شود مقدار Count یک واحد کاهش می‌یابد. به این ترتیب هرگاه مقدار Count صفر شود هیچ اشاره‌گری به فضای مربوطه اشاره نمی‌کند و فضای مورد نظر می‌تواند آزاد گردد. عمل آزادسازی Garbageها توسط واحدی تحت عنوان Garbage Collector انجام می‌گیرد.

۷- گزینه (۲) صحیح است.

در اینجا برای جدول صفحه جزئی محدودیتی به اندازه یک قاب (صفحه) داریم.

بنابراین اندازه جدول صفحه جزئی برابر اندازه قاب (صفحه) می‌باشد. بنابراین برای محاسبه تعداد سطرهای جدول صفحه جزئی، کافی است، اندازه قاب که برابر اندازه جدول صفحه جزئی است بر اندازه عرض جدول صفحه جزئی تقسیم گردد. به شکل زیر توجه کنید:

khalilifar.ir



توجه: عرض جدول صفحه همواره برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب است، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد.

بنابراین داریم:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

$4B =$ عرض جدول صفحه جزئی

توجه: مطابق فرض سؤال، هر مدخل جدول صفحه (عرض جدول صفحه جزئی) 4 بایت در نظر گرفته شده است.

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^{10}B}{2^{2}B} = 2^{8} = 256$$

$2^{34}B =$ اندازه فرآیند (فضای آدرس مجازی)

$1KB = 1024B = 2^{10}B =$ اندازه صفحه = اندازه قاب

$$f: \text{تعداد صفحات فرآیند} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه}} = \frac{2^{34}}{2^{10}} = 2^{24}$$

$r: \text{تعداد سطرهای جدول صفحه جزئی} = 256 = 2^{8}$

حال اطلاعات کافی برای محاسبه تعداد سطوح جدول صفحه چند سطحی را در اختیار داریم:

روش تجزیه

تعداد صفحات فرآیند باید در اندازه r(2^8) تجزیه گردد:

$$\text{تعداد صفحات فرآیند} = 2^{24} = 2^{\overset{\uparrow}{\textcircled{8}}} \times 2^{\overset{\uparrow}{\textcircled{8}}} \times 2^{\overset{\uparrow}{\textcircled{8}}}$$

PT1 PT2 PT3

khalilifar.ir

توجه: تعداد عملوندها برابر 3 است، بنابراین تعداد سطوح جدول چند سطحی نیز برابر 3 است.

روش لگاریتم

$$\text{تعداد سطوح جدول چند سطحی} = d = \left\lceil \log_r^f \right\rceil = \left\lceil l0g_{2^8}^{2^{24}} \right\rceil = 3$$

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح سوم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{2^{24}}{2^8} = 2^{16}$$

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد جداول صفحه جزئی در سطح سوم}}{r} = \frac{2^{16}}{2^8} = 2^8$$

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{r} = \frac{2^8}{2^8} = 1$$

توجه: سطح اول، یک جدول به حساب می‌آید، که 2^8 سطر دارد.

توجه: تعداد تقسیم متوالی برابر 3 است، بنابراین تعداد سطوح جدول صفحه چند سطحی برابر 3 است.

$$\text{تعداد بیت آفست} = \log_2^{\text{اندازه صفحه}} = \log_2^{2^{10}} = 10 \text{ بیت}$$

بنابراین شکل آدرس منطقی (مجازی) به صورت زیر خواهد بود:

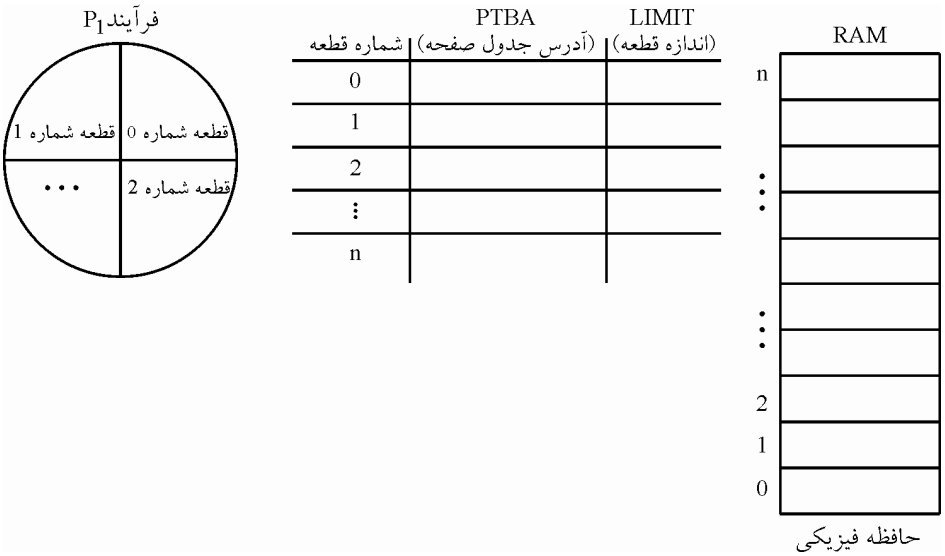
PT1	PT2	PT3	آفست
8 بیت	8 بیت	8 بیت	10 بیت
24 بیت			

۸- گزینه (۱) صحیح است.

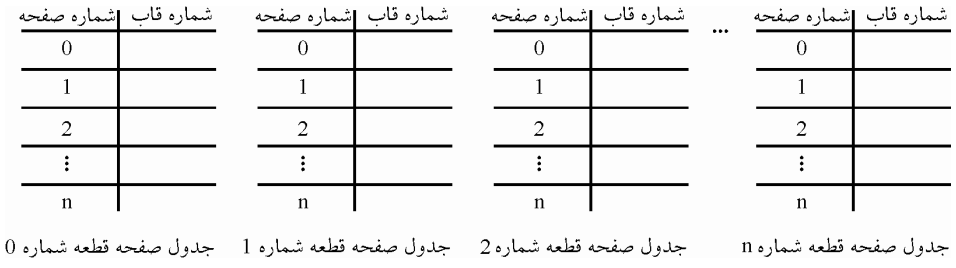
در راه‌حل قطعه‌بندی همراه با صفحه‌بندی، قطعات تولید شده توسط برنامه نویس یا کامپایلر، توسط سیستم عامل صفحه‌بندی می‌شوند. در این حالت به ازای هر فرآیند یک جدول قطعه ایجاد می‌شود که تعداد درایه‌های جدول قطعه، بستگی به اندازه فرآیند و تعداد قطعات حاصل دارد.

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه‌حل قطعه‌بندی همراه با صفحه‌بندی، به صورت زیر است:

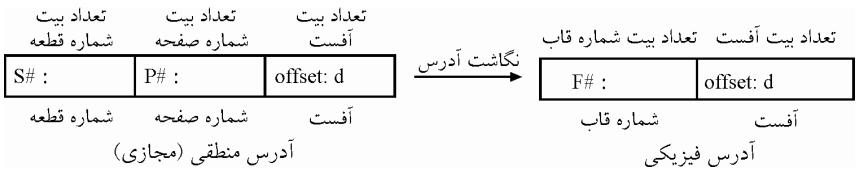
khalilifar.ir



همچنین به ازای هر قطعه، یک جدول صفحه ایجاد می‌شود که تعداد درایه‌های جدول صفحه بستگی به اندازه قطعه و تعداد صفحات حاصل دارد.



به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه‌حل قطعه‌بندی همراه با صفحه‌بندی، به صورت زیر است:



بر اساس آدرس منطقی (مجازی) و مطابق مفروضات صورت سؤال داریم:

اندازه صفحه یا اندازه قاب × تعداد صفحات فرآیند × تعداد قطعات فرآیند = اندازه حافظه مجازی یا منطقی (فرآیند)

تعداد قطعات فرآیند $= 2^{\text{تعداد بیت شماره قطعه}} = 2^4$

تعداد صفحات فرآیند $= 2^{\text{تعداد بیت شماره صفحه}} = 2^{12}$

اندازه صفحه یا اندازه قاب $= 2^{\text{تعداد بیت آفست}} = 4096$ word (کلمه)

khalilifar.ir

بنابراین داریم:

$2^4 \times 2^{12} \times 4096$ = اندازه حافظه مجازی یا منطقی (فرآیند)

همچنین به بیانی دیگر داریم:

$2^{\text{تعداد بیت آدرس منطقی}}$ = اندازه حافظه مجازی یا منطقی (فرآیند)

$2^{\text{تعداد بیت آفست}} \times 2^{\text{تعداد بیت شماره صفحه}} = 2^{\text{تعداد بیت شماره قطعه}}$ =

$2^4 \times 2^{12} \times 4096$ = اندازه حافظه مجازی یا منطقی (فرآیند)

براساس آدرس فیزیکی و مطابق مفروضات صورت سؤال داریم:

توجه: منظور از کلمه «**بلوک**» در جدول مربوط به صورت سؤال همان فریم یا قاب است.

اندازه صفحه یا اندازه قاب × تعداد قاب‌های حافظه اصلی = اندازه حافظه اصلی یا فیزیکی (RAM)

$2^{12} = 2^{\text{تعداد بیت شماره قاب}}$ = تعداد قاب‌های حافظه اصلی (فیزیکی)

$2^{\text{تعداد بیت آفست}} = 4096$ word (کلمه) = اندازه صفحه یا اندازه قاب

بنابراین داریم:

$2^{12} \times 4096$ = اندازه حافظه اصلی یا فیزیکی (RAM)

همچنین به بیانی دیگر داریم:

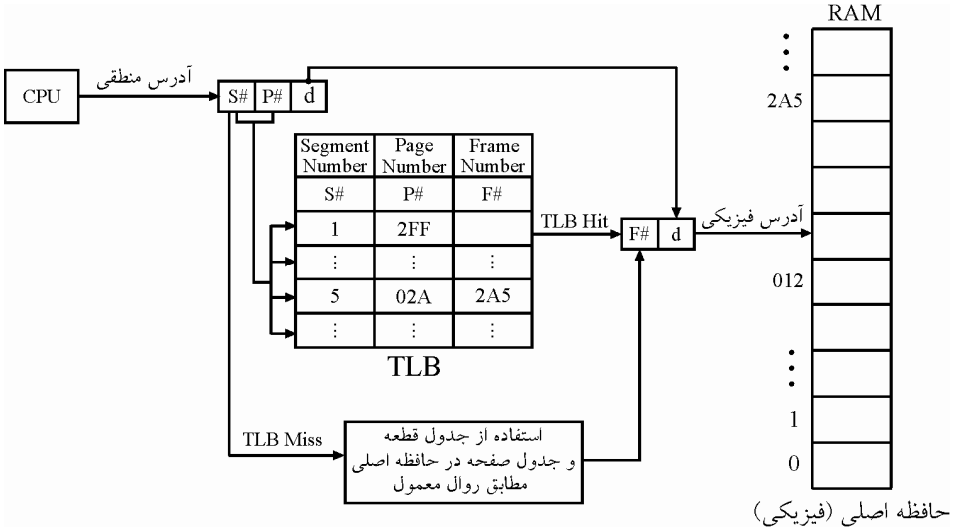
$2^{\text{تعداد بیت آفست}} \times 2^{\text{تعداد بیت شماره قاب}} = 2^{\text{تعداد بیت آدرس فیزیکی}}$ = اندازه حافظه اصلی یا فیزیکی (RAM)

$2^{12} \times 4096$ = اندازه حافظه اصلی یا فیزیکی (RAM)

توجه: جهت افزایش سرعت نگاشت آدرس منطقی به فیزیکی می‌توان از TLB نیز استفاده نمود. TLB از حافظه‌های با سرعت بسیار بالا و گران‌قیمت ساخته شده است که در واقع مجموعه‌ای از رجیسترها موسوم به Associative Register هستند. در سیستم قطعه‌بندی همراه با صفحه‌بندی، وقتی فرآیندی جهت اجرا انتخاب شود، بخشی از درایه‌های جدول قطعه و جدول صفحه آن به TLB منتقل می‌شود.

در این حالت وقتی CPU یک آدرس منطقی (مجازی) تولید می‌کند، شماره قطعه و شماره صفحه آن ابتدا در TLB جستجو می‌شود، اگر شماره قطعه و شماره صفحه در TLB یافت شد که آدرس قاب متناظر در حافظه اصلی (فیزیکی) به دست می‌آید. اما اگر شماره قطعه و شماره صفحه در TLB نباشد، آنگاه طبق روال معمول به سراغ جدول قطعه و جدول صفحه در حافظه اصلی می‌رویم. مطابق مفروضات صورت سؤال داریم:

khalilifar.ir



در صورت سؤال مطرح شده است که حجم حافظه مجازی (منطقی) چند برابر حافظه اصلی (فیزیکی) است، که مطابق آنچه بررسی کردیم، به صورت زیر قابل محاسبه است:

$$\frac{\text{حجم یا اندازه حافظه مجازی (منطقی)}}{\text{حجم یا اندازه حافظه اصلی (فیزیکی)}} = \frac{2^4 \times 2^{12} \times 4096}{2^{12} \times 4096} = 2^4 = 16$$

۹- گزینه (۲) صحیح است.

پساکنکور: اغلب بچه‌ها بعد از کنکور دولتی ۱۳۹۵ و همزمان با اوج‌گیری اعتراضات و اعلام کلید اولیه آزمون دولتی ۹۵ از سوی سازمان سنجش آموزش کشور به دلیل وجود دو ابهام در صورت سؤال، به این سؤال معترض و خواهان حذف این سؤال از سوی سازمان سنجش آموزش کشور بودند، اما حکایت به گونه‌ای دیگر رقم خورد و سازمان محترم سنجش و آموزش کشور بی‌اعتنا و بی‌توجه به اعتراض‌های به حق بچه‌ها، در نهایت در کلید نهایی خود (موسوم به کلید ثانویه) نیز مجدداً گزینه دوم را به عنوان پاسخ نهایی اعلام کرد.

ابهام اول: در صورت سؤال زمان خواندن از حافظه اصلی یعنی 100ms در واحد میلی ثانیه بیان شده است. اما درخواست سؤال مبنی بر محاسبه زمان خواندن از TLB در واحد نانوثانیه است که منجر به این می‌شود، اطلاعات سؤال با گزینه‌ها سازگاری نداشته باشد. که به نظر می‌رسد به هنگام حروفچینی سؤال 100ns به شکل 100ms تایپ شده است (خطای حروفچینی).

توجه: ما در هنگام حل سؤال زمان خواندن از حافظه اصلی را 100ns در واحد نانوثانیه در نظر می‌گیریم.

ابهام دوم: در صورت سؤال به وضوح مشخص نشده است که منظور از کارایی سیستم، چه نوع کارایی است. اینکه **کارایی زمان دسترسی سیستم** مدنظر طراح است؟ و یا **کارایی زمان ترجمه**

khalilifar.ir

سیستم؟ اما طراح با فرض کارایی زمان دسترسی سیستم، سؤال را طرح کرده است (خطای طراح سؤال).

توجه: ما در هنگام حل سؤال، هر دو فرض را در نظر می‌گیریم.

در صورت سؤال مطرح شده است که سیستمی علاوه بر ذخیره جدول صفحه در حافظه اصلی، از جدول TLB نیز با نرخ miss برابر 20% استفاده می‌کند. اگر خواندن از حافظه اصلی 100ns زمان بردارد و درصد کارایی سیستم در صورت استفاده نکردن از جدول TLB برابر 80% باشد، خواندن از TLB چند نانوثانیه زمان لازم دارد؟

فرض اول: کارایی زمان دسترسی سیستم

مطابق رابطه زیر داریم:

زمان دسترسی سیستم بدون استفاده از TLB × کارایی زمان دسترسی سیستم = زمان دسترسی مؤثر سیستم با استفاده از TLB

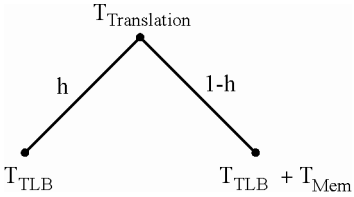
توجه: زمان دسترسی مؤثر سیستم با استفاده از جدول TLB از رابطه زیر محاسبه می‌گردد:

$$T_{eff} = T_{Access} = T_{Translation} + T_{Destination}$$

که پارامترهای آن به صورت زیر است:

$T_{Translation}$: میانگین زمان ترجمه

برای محاسبه $T_{Translation}$ درخت زیر را در نظر بگیرید:



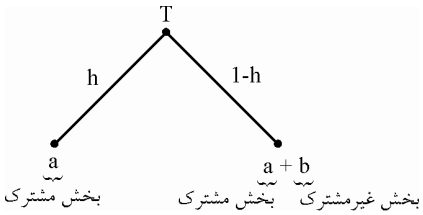
رابطه درخت فوق به صورت زیر خواهد بود:

$$T_{Translation} = h \times T_{TLB} + (1-h) \times (T_{TLB} + T_{Mem})$$

که پس از ساده سازی داریم:

$$T_{Translation} = T_{TLB} + (1-h) \times T_{Mem}$$

توجه: جهت ساده‌سازی رابطه درخت فوق، همواره می‌توان از **اتحاد درخت احتمال** به صورت زیر، استفاده نمود:



khalilifar.ir

$T =$ بخش مشترک $+(1-h)\times$ بخش غیرمشترک

$$T = a + (1-h)\times b$$

پس از ساده‌سازی درخت مطرح شده براساس اتحاد درخت احتمال، رابطه زیر را برای محاسبه $T_{Translation}$ خواهیم داشت:

$$T_{Translation} = T_{TLB} + (1-h)\times T_{Mem}$$

$T_{Destination}$: میانگین زمان دسترسی به مقصد مورد نظر

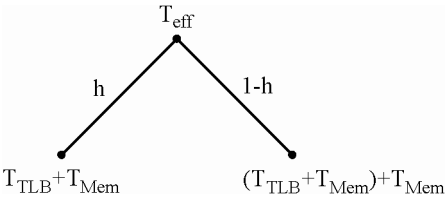
از آنجاییکه مطابق مفروضات مسأله، فقط حافظه اصلی به عنوان مقصد موردنظر، درنظر گرفته شده است، پس $T_{Destination}$ به صورت زیر محاسبه می‌گردد:

$$T_{Destination} = T_{Mem}$$

پس در نهایت زمان دسترسی مؤثر سیستم با استفاده از TLB مطابق آنچه گفتیم به صورت زیر محاسبه می‌گردد:

$$T_{eff} = T_{Access} = T_{TLB} + (1-h)T_{Mem} + T_{Mem}$$

توجه: در یک نگاه کلی‌تر، زمان دسترسی مؤثر سیستم با استفاده از جدول TLB را می‌توان از طریق درخت زیر نیز محاسبه نمود:



مطابق اتحاد درخت احتمال، رابطه زیر را خواهیم داشت:

$$T_{eff} = T_{Access} = T_{TLB} + T_{Mem} + (1-h)\times T_{Mem}$$

توجه: زمان دسترسی سیستم بدون استفاده از جدول TLB از رابطه زیر محاسبه می‌گردد:

$$T_{Access} = T_{Translation} + T_{Destination}$$

که پارامترهای آن به صورت زیر است:

$T_{Translation}$: زمان ترجمه

$$T_{Translation} = T_{Mem}$$

$T_{Destination}$: زمان دسترسی به مقصد مورد نظر

$$T_{Destination} = T_{Mem}$$

پس در نهایت زمان دسترسی سیستم بدون استفاده از جدول TLB مطابق آنچه گفتیم به صورت زیر محاسبه می‌گردد:

$$T_{Access} = T_{Mem} + T_{Mem} = 2\times T_{Mem}$$

khalilifar.ir

در نهایت مطابق آنچه گفتیم داریم:

زمان دسترسی سیستم بدون استفاده از TLB × کارایی زمان دسترسی سیستم = زمان دسترسی مؤثر سیستم با استفاده از TLB

$$T_{TLB} + (1-h)T_{Mem} + T_{Mem} = 0.8 \times (2 \times T_{Mem})$$

$$T_{TLB} + 0.2 \times 100 + 100 = 0.8 \times (2 \times 100)$$

$$T_{TLB} + 20 + 100 = 160$$

$$T_{TLB} = 160 - 120 = 40 \text{ ns}$$

بنابراین اگر منظور از کارایی مطرح شده در صورت سؤال، کارایی زمان دسترسی سیستم باشد، آنگاه گزینه دوم پاسخ سؤال خواهد بود.

فرض دوم: کارایی زمان ترجمه سیستم

مطابق رابطه زیر داریم:

زمان ترجمه سیستم بدون استفاده از TLB × کارایی زمان ترجمه سیستم = زمان ترجمه موثر سیستم با استفاده از TLB

توجه: مطابق آنچه گفتیم، زمان ترجمه سیستم با استفاده از جدول TLB از رابطه زیر محاسبه می‌گردد:

$$T_{Translation} = T_{TLB} + (1-h) \times T_{Mem}$$

توجه: همچنین، مطابق آنچه گفتیم، زمان ترجمه سیستم بدون استفاده از جدول TLB از رابطه زیر محاسبه می‌گردد:

$$T_{Translation} = T_{Mem}$$

در نهایت مطابق آنچه گفتیم داریم:

زمان ترجمه سیستم بدون استفاده از TLB × کارایی زمان ترجمه سیستم = زمان ترجمه موثر سیستم با استفاده از TLB

$$T_{TLB} + (1-h) \times T_{Mem} = 0.8 \times T_{Mem}$$

$$T_{TLB} + 0.2 \times 100 = 0.8 \times 100$$

$$T_{TLB} + 20 = 80$$

$$T_{TLB} = 60 \text{ ns}$$

بنابراین اگر منظور از کارایی مطرح شده در صورت سؤال، کارایی زمان ترجمه سیستم باشد، آنگاه گزینه چهارم پاسخ سؤال خواهد بود. همانطور که گفتیم سازمان سنجش آموزش کشور، در کلید اولیه و نهایی خود، گزینه دوم را به عنوان پاسخ اعلام کرده بود.

khalilifar.ir

۱۰- گزینه (۳) صحیح است.

یک راه حل کلی جهت مقابله با تکه تکه شدن خارجی این است که اجازه دهیم یک فرآیند در قسمت‌های همجوار در حافظه قرار گیرد. یکی از روش‌هایی که از این ایده استفاده می‌کند، تکنیک صفحه‌بندی است. در این روش حافظه به بخش‌هایی با اندازه‌ی یکسان به نام قاب (Frame) تقسیم می‌شود. از طرفی برنامه‌ها نیز به قسمت‌های مساوی و هم‌اندازه با قاب‌ها تقسیم می‌شوند که به آن‌ها صفحه (Page) می‌گویند. حال هنگامی که برنامه‌ای به حافظه منتقل می‌شود باید تمام صفحاتش به داخل قاب‌های خالی آورده شوند. در این حالت اصلاً نیازی نیست صفحات مربوط به یک فرآیند در قاب‌های همجوار قرار گیرند.

مزیت عمده این روش از بین بردن تکه تکه شدن خارجی و به حداقل رساندن تکه تکه شدن داخلی می‌باشد، اما در عوض عملیات محاسبه آدرس‌ها و مدیریت این صفحات قدری هزینه‌بر و زمان‌گیر است.

توجه: برای پیاده‌سازی این روش به پشتیبانی سخت‌افزار نیاز است.

توجه: این روش از دید کاربر و برنامه‌نویس مخفی می‌ماند.

توجه: برای پیاده‌سازی این روش و مدیریت صفحه‌ها و از همه مهم‌تر تبدیل و نگاشت آدرس‌ها باید یک جدول صفحه به ازای هر فرآیند در نظر گرفت. در واقع جدول صفحه‌ی هر فرآیند دارای یک درایه به ازای هر صفحه می‌باشد که مشخص می‌کند هر صفحه از یک فرآیند در کدام قاب حافظه نگهداری می‌شود.

کارکرد اصلی جدول صفحه، تبدیل و نگاشت آدرس‌های مجازی به آدرس‌های فیزیکی می‌باشد. از نظر ریاضی جدول صفحه در واقع فقط یک تابع است که ورودی آن شماره صفحه مجازی و خروجی آن شماره قاب فیزیکی می‌باشد.

اما در این بین دو مسئله اساسی وجود دارد:

۱- نگاشت آدرس‌ها باید بسیار سریع صورت گیرد. هنگامی که جداول صفحه در حافظه اصلی باشند برای دسترسی به یک خانه حافظه، باید دو یا چند بار به حافظه سر زد. در اینصورت وقتی جداول به صورت چند سطحی پیاده‌سازی شوند، این تعداد بیشتر هم می‌شود و سرعت دسترسی به حافظه به شدت کاهش می‌یابد. در این حالت برای افزایش سرعت دسترسی به حافظه از تکنیک TLB استفاده می‌شود. TLB از حافظه‌های با سرعت بسیار بالا و گران قیمت ساخته شده است که در واقع مجموعه‌ای از رجیسترها موسوم به Associative Register هستند. وقتی فرآیندی جهت اجرا انتخاب شود، بخشی از درایه‌های جدول صفحه آن به TLB منتقل می‌شود. در این حالت وقتی CPU یک آدرس منطقی تولید می‌کند، شماره صفحه آن ابتدا در TLB جست‌وجو می‌شود، اگر شماره صفحه در TLB یافت شد که آدرس قاب متناظر به دست می‌آید، اما اگر شماره صفحه در TLB نباشد آن‌گاه طبق روال قبل به سراغ جدول صفحه در حافظه می‌رویم. نکته مهم در مورد TLB این است که عمل جست‌وجو در یک لحظه و همزمان در تمام سطرها صورت می‌گیرد. به

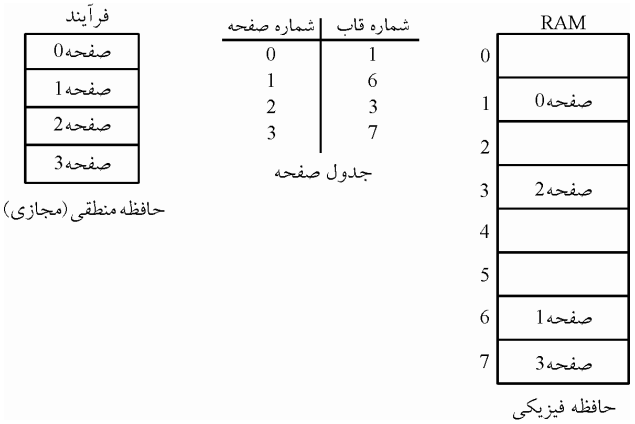
khalilifar.ir

همین دلیل سرعت دسترسی به آن بسیار بالاست. این مساله برای کاربرنهـایی اهمیـت دارد. کـاربر نهایی زمان پاسخ کوتاه را در انجام کارها انتظار دارد.

۲- جدول صفحه می‌تواند بسیار بزرگ شود. سیستم صفحه‌بندی، با محدودیت اندازه قـاب مواجـه است، و اگر فرآیندی درون یک قاب جا نشود آنگاه باید در چند قـاب قـرار گیـرد. همچنـین ایـن روند برای جدول صفحه مربوط به یک فرآیند نیز وجود دارد و اگر جدول صفحه فرآینـدی درون یک قاب جا نشود آنگاه باید در چند قاب قرار گیرد. و همین مسئله منجر به ابداع جـداول صـفحه چند سطحی شد.

برای درک این ساختار، به مثال‌های زیر توجه کنید:

مثال اول: اگر فرآیندی که از چهار صفحه منطقی تشکیل شده است به صورت زیر در قاب‌های حافظه فیزیکی قرار گرفته باشد، آنگاه جدول صفحه به صورت زیر در خواهد آمد:

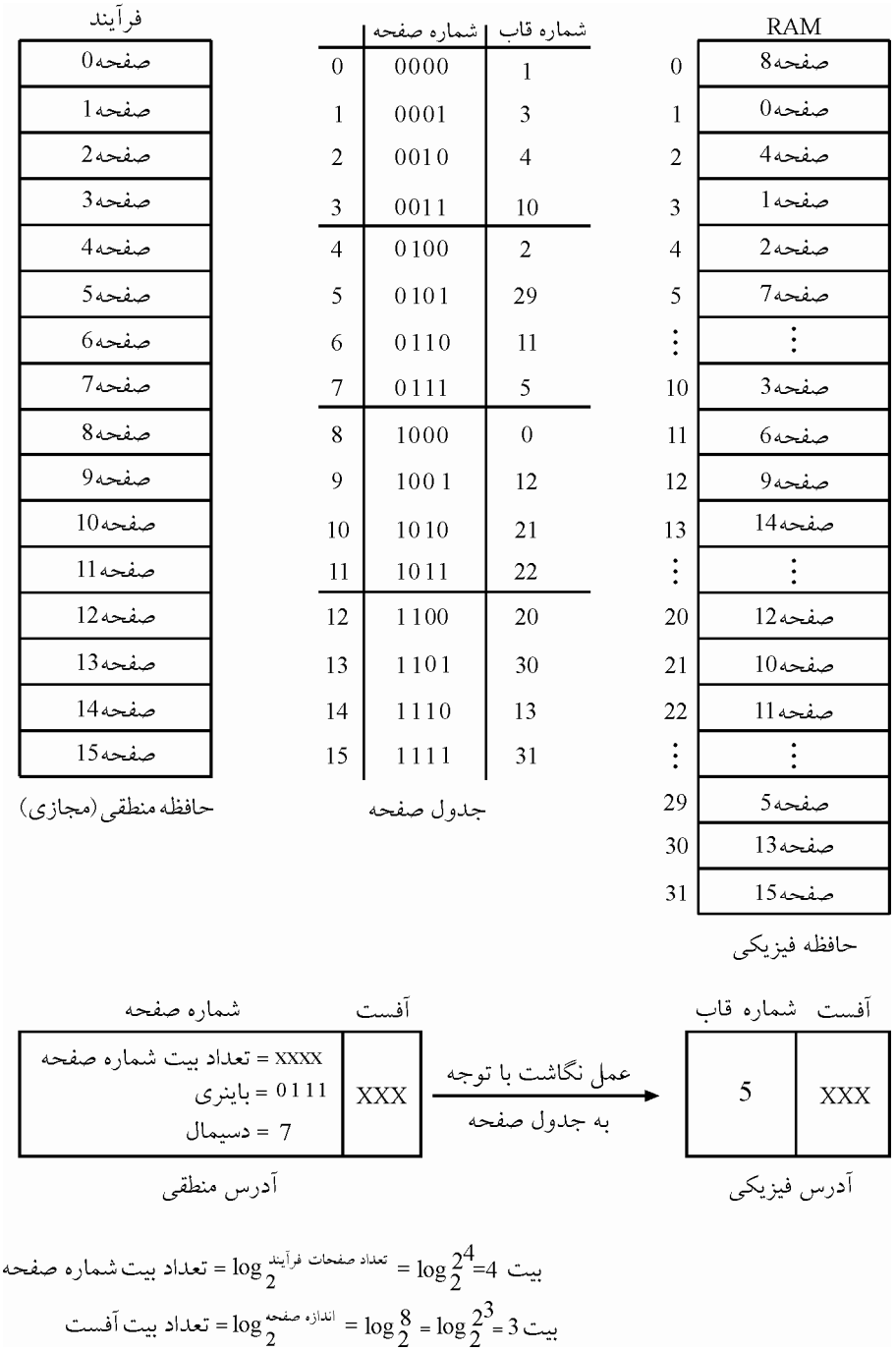


در راه‌حل فوق، از یک جدول صفحه تک سطحی، برای مدیریت صفحه‌بندی فرآیند استفاده شـده است. فرآیند فوق شامل چهار صفحه می‌باشد و واضح است که جدول صفحه نیز باید شامل چهار سطر باشد. برای مدیریت صفحه‌ها و نگاشت آدرس‌ها باید یک جدول صفحه به ازای هـر فرآینـد در نظر گرفت. در واقع جدول صفحه هر فرآیند دارای یک درایه به ازای هر صفحه مـی‌باشـد کـه مشخص می‌کند هر صفحه از یک فرآیند در کدام قاب حافظه نگهداری می‌شود.

مثال دوم: فرآیندی شامل 16 صفحه می‌باشد، اگر اندازه‌ی جـدول صـفحه دارای محـدودیت باشـد و حداکثر هر جدول صفحه بتواند دارای چهار سطر باشد، آنگاه جدول صفحه این فرآیند چنـد سـطحی خواهد بود؟ (اندازه‌ی صفحات 8 بایت است)

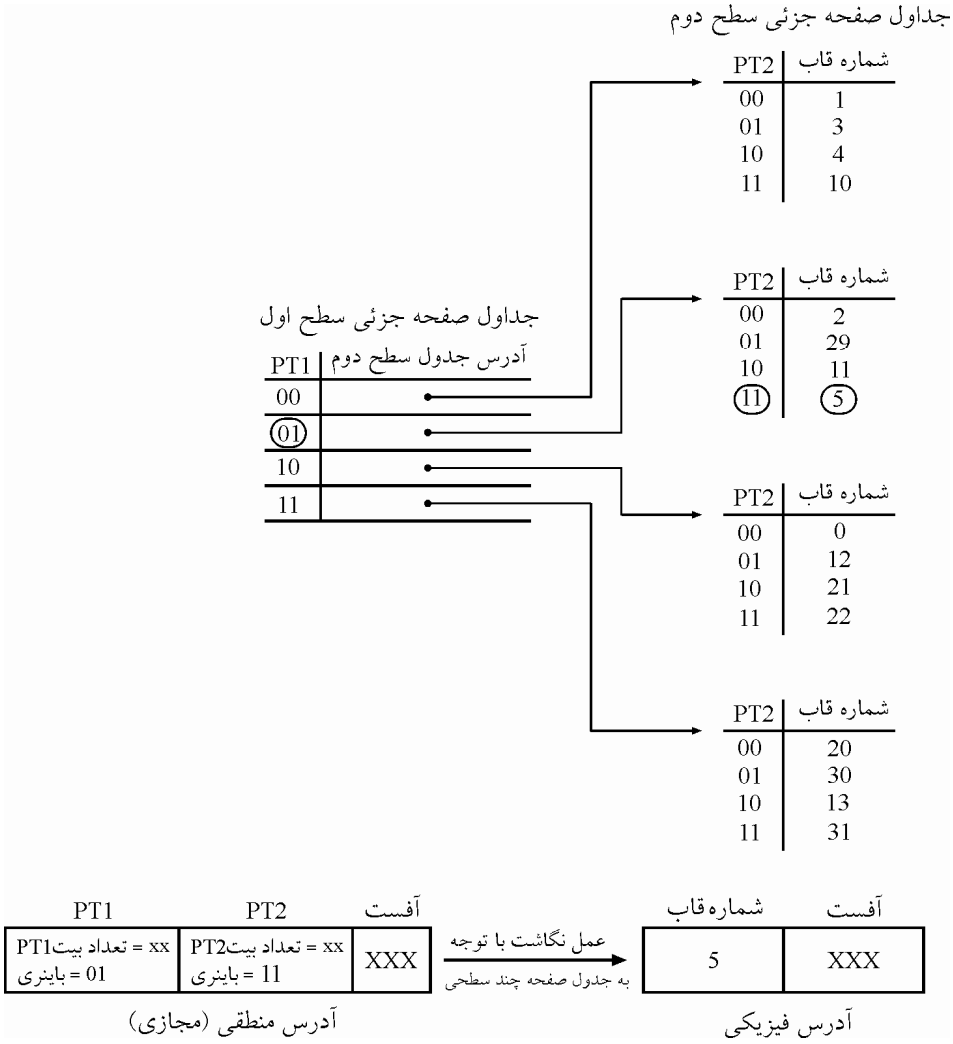
پاسخ: واضح است که اگر اندازه‌ی جدول صفحه دارای محدودیت نباشد، ایـن فرآینـد دارای یـک جدول صفحه تک سطحی با 16 سطر خواهد بود. شکل زیر جدول صفحه تک سطحی این فرآیند را نشان می‌دهد:

khalilifar.ir



khalilifar.ir

اما اگر اندازه‌ی جدول صفحه دارای محدودیت باشد و حداکثر بتواند دارای 4 سطر باشد، واضح است که در این حالت باید از راه‌حل جدول صفحه چند سطحی استفاده شود. بنابراین با 16 صفحه مواجه هستیم که هر 4 صفحه آن می‌تواند داخل یک جدول صفحه جزئی قرار بگیرد.
در شکل زیر واضح است که با توجه به شرایط مسأله یک جدول صفحه دو سطحی مسأله را حل می‌کند.



وقتی که یک آدرس مجازی به مدیر حافظه برسد، ابتدا فیلد PT1 این آدرس استخراج شده و به کمک آن درایه موردنظر در جدول سطح اول پیدا می‌شود. هر یک از درایه‌های این جدول، نماینده یک جدول سطح دوم می‌باشد. در واقع از مقادیر موجود در درایه‌های جدول سطح اول برای پیدا

khalilifar.ir

کردن جدول سطح دوم مناسب استفاده می‌شود. اکنون از فیلد PT2 به عنوان اندیس جدول سطح دوم استفاده و به کمک آن، شماره فیزیکی قاب حافظه استخراج می‌شود و در نهایت بخش آفست نیز به این آدرس متصل می‌گردد.

توجه: در مثال فوق مشاهده می‌شود که برای دسترسی به هر قاب فقط به 2 جدول نیاز است. یکی جدول سطح اول و دیگری یکی از جداول سطح دوم. در واقع مزیت این روش از اینجا ناشی می‌شود که **فقط جداولی به حافظه آورده می‌شوند که مورد نیاز هستند.** دقت کنید این مفهوم با مفهوم حافظه مجازی متفاوت است. در مبحث حافظه مجازی مسأله این است که همه صفحات یک فرآیند به حافظه آورده نشوند. اما اینجا جداول صفحه به حافظه آورده نمی‌شوند.

توجه: به جداول موجود در سطوح مختلف جدول چند سطحی، **جداول جزئی** نیز گفته می‌شود. مجموع این جداول جزئی، جدول چند سطحی را ایجاد می‌کنند.

روابط زیر در جدول صفحه چند سطحی برقرار است:

f: تعداد صفحات فرآیند = 16

r: تعداد سطرهای جدول صفحه جزئی = 4

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{16}{4} = 4$$

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{r} = \frac{4}{4} = 1$$

روال فوق گویای این مفهوم می‌باشد، که این تقسیم متوالی تا جایی ادامه پیدا می‌کند که خارج قسمت کوچکتر یا برابر یک شود. یعنی به یک جدول برسیم، به اندازه‌ی محدودیت.

همچنین این نتیجه، تعداد سطوح جدول چند سطحی را نیز مشخص می‌کند. یعنی تعداد تقسیم از ابتدا تا به انتها با شرط مذکور.

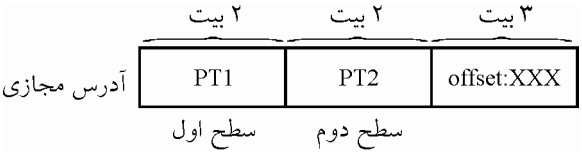
توجه: در اینجا تعداد تقسیم متوالی برابر 2 است، بنابراین تعداد سطوح جدول چندسطحی برابر 2 است.

توجه: روال فوق را نیز می‌توان در مفهوم لگاریتم نیز جست‌وجو کرد، در واقع تقسیم فوق مفهوم لگاریتم را پیاده‌سازی می‌کند:

$$\text{تعداد سطوح جدول صفحه چند سطحی} = d = \lceil \log_r^f \rceil = \lceil \log_4^{16} \rceil = \left\lceil \log_{2^2}^{2^4} \right\rceil = 2$$

بنابراین در این مثال، آدرس‌های مجازی به سه بخش تقسیم می‌شوند:

khalilifar.ir



اندازه فرآیند = اندازه صفحه × تعداد صفحات $=2^4 \times 2^3=2^7=128$ بایت

تعداد بیت آفست $= \log_2^{\text{اندازه صفحه}} = \log_2^8 = \log_2^{2^3} = 3$ بیت

$PT1 + PT2 = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^{2^4} = 4$

$PT2 = \log_2^r = \log_2^4 = \log_2^{2^2} = 2$

توجه: فرمول PT2 خیلی ساده است، جدول صفحه حداکثر می‌تواند چهار سطر داشته باشد، چهار سمبل داریم، برای آدرس‌دهی چهار سمبل به چند بیت نیاز داریم، واضح است که دو بیت، این همان مفهوم لگاریتم است.

$PT1 = (PT1+PT2) - PT2 = 2$ بیت

راه‌حل ساده‌تر (تجزیه): اندازه فرآیند را در اندازه r تجزیه کنید:

توجه: اندازه r، برابر تعداد سطرهای جداول صفحه است.

تعداد صفحات فرآیند $= 2^4 = 2^{\overset{\uparrow}{②}} \times 2^{\overset{\uparrow}{②}}$ (PT1 PT2)

توجه: تجزیه را از راست به چپ و از اندیس n به 1 شروع کنید و کمترین مقدار در اندیس 1 قرار داده شود.

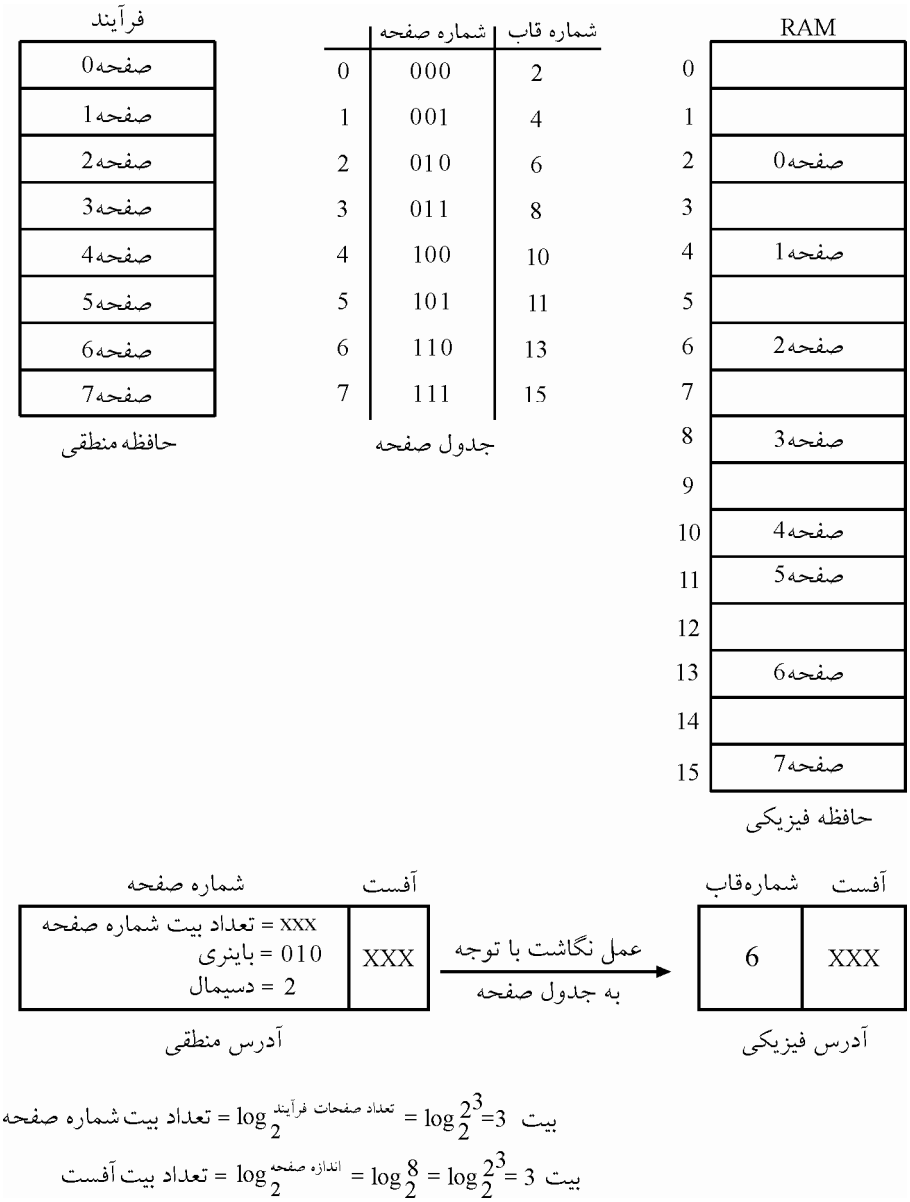
توجه: توان کمتر همواره PT1 است که در این مثال PT1 و PT2 برابر هستند.

توجه: تعداد عملوندها در رابطه بالا برابر تعداد سطوح جدول چند سطحی نیز می‌باشد، در رابطه فوق تعداد عملوندها برابر 2 است. بنابراین تعداد سطوح جدول چندسطحی نیز برابر 2 است، این نتیجه از آن‌جایی ناشی می‌شود که تجزیه فوق همان مفهوم لگاریتم را پیاده‌سازی می‌کند.

مثال سوم: فرآیندی شامل 8 صفحه می‌باشد، اگر اندازه‌ی جدول صفحه دارای محدودیت باشد و حداکثر هر جدول صفحه بتواند دارای 2 سطر باشد، آنگاه جدول صفحه این فرآیند چند سطحی خواهد بود؟ (اندازه صفحات 8 بایت است.)

پاسخ: واضح است که اگر اندازه‌ی جدول صفحه دارای محدودیت نباشد، این فرآیند دارای یک جدول صفحه تک‌سطحی با 8 سطر خواهد بود. شکل زیر جدول صفحه تک‌سطحی این فرآیند را نشان می‌دهد:

khalilifar.ir



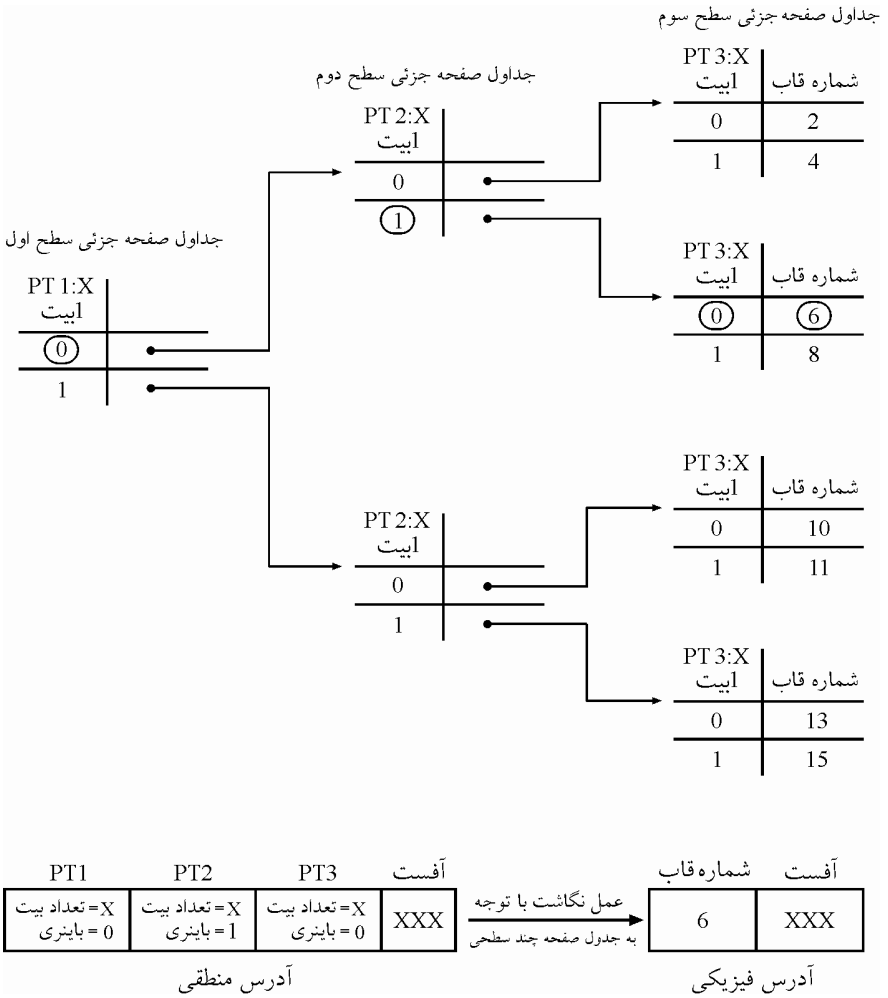
$$\text{تعداد بیت شماره صفحه} = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2 \frac{2^3}{2} = 3 \text{ بیت}$$

$$\text{تعداد بیت آفست} = \log_2^{\text{اندازه صفحه}} = \log_2 \frac{8}{2} = \log_2 \frac{2^3}{2} = 3 \text{ بیت}$$

اما اگر اندازه جدول صفحه دارای محدودیت باشد و حداکثر بتوانـد دارای 2 سـطر باشـد، واضـح است که در این حالت باید از راه‌حل جدول صفحه چندسطحی استفاده شود. بنابراین با 8 صـفحه مواجه هستیم که هر 2 صفحه آن می‌تواند داخل یک جدول صفحه جزئی قرار بگیرد.

در شکل زیر واضح است که با توجه به شرایط مسأله یک جدول صفحه سه سطحی مسأله را حـل می‌کند.

khalilifar.ir



روابط زیر در جدول صفحه چندسطحی برقرار است:

f: تعداد صفحات فرآیند = 8

r : تعداد سطرهای جدول سطحی جزئی = 2

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح سوم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{8}{2} = 4$$

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد جداول در سطح سوم}}{r} = \frac{4}{2} = 2$$

khalilifar.ir

تعداد جداول صفحه جزئی در سطح اول = $\frac{\text{تعداد جداول در سطح دوم}}{r} = \frac{2}{2} = 1$

توجه: در اینجا تعداد تقسیم متوالی برابر 3 است، بنابراین تعداد سطوح جدول چندسطحی برابر 3 است.

روش لگاریتم

تعداد سطوح جدول صفحه چند سطحی = $d = \lceil \log_r^f \rceil = \lceil \log_2^8 \rceil = \left\lceil \log_2^{2^3} \right\rceil = 3$

روش تجزیه

تعداد صفحات فرآیند باید در اندازه r، تجزیه گردد.

تعداد صفحات فرآیند = $2^3 = 2^{\overset{\uparrow}{①}PT1} \times 2^{\overset{\uparrow}{①}PT2} \times 2^{\overset{\uparrow}{①}PT3}$

توجه: تعداد عملوندها برابر 3 است، بنابراین تعداد سطوح جدول چندسطحی نیز برابر 3 است.

توجه: ایده اصلی حافظه مجازی این است که حتی اگر به اندازه کافی فضای خالی بر روی حافظه در اختیار نداشته باشیم، به یک فرآیند اجازه اجرا بدهیم. نکته اصلی این است که یک فرآیند در آن واحد به همه داده‌ها و کد خود نیاز ندارد بلکه در هر لحظه فقط به بخشی از داده و قسمتی از کد نیازمند است. در تکنیک حافظه مجازی فقط قسمتی از فرآیند به حافظه آورده می‌شود که در حال حاضر به آن نیاز است و مابقی فرآیند می‌تواند کماکان بر روی دیسک قرار گیرد و در طول اجرای یک فرآیند این جابه‌جایی‌ها بین حافظه و دیسک مرتباً صورت گیرد. با این کار می‌توانیم فرآیندهای بیشتری را در داخل حافظه‌ی اصلی نگهداری کنیم. اگر این تکنیک از دید فرآیند مخفی بماند، فرآیند گمان می‌کند تمام فضای درخواستی وی به او تخصیص داده شده است، در صورتی که چنین نیست و در واقع سیستم عامل با تلاشی مضاعف و با انجام جابه‌جایی‌های پی‌درپی این دید را برای فرآیند ایجاد کرده است.

توجه: با توجه به مسائل مطرح شده، مجموع کلیه فضای آدرس فرآیندهایی که در حال اجرا هستند، می‌تواند از اندازه حافظه فیزیکی بیشتر شود و این یعنی حافظه مجازی. در واقع همه فرآیندها گمان می‌کنند به طور کامل در حافظه قرار دارند، غافل از اینکه قسمت اعظم هریک از آن‌ها بر روی دیسک است.

توجه: ایده‌ی حافظه مجازی معمولاً با تکنیک صفحه‌بندی راحت‌تر پیاده‌سازی می‌شود که به آن صفحه‌بندی بر حسب نیاز (Demand Paging) گویند. البته حافظه مجازی را می‌توان با روش‌های دیگری، مانند قطعه‌بندی نیز پیاده‌سازی کرد، اما روال کار پیچیده‌تر می‌شود.

یکی از بهترین روش‌ها جهت پیاده‌سازی ایده حافظه مجازی، استفاده از تکنیک صفحه‌بندی است. در این حالت تنها تعداد اندکی از صفحه‌های یک فرآیند به حافظه آورده و مابقی صفحات بر روی دیسک نگهداری می‌شوند. در این شیوه اگر به صفحات موجود بر روی دیسک نیاز پیدا کردیم، آن

khalilifar.ir

صفحات به جای صفحات قدیمی به حافظه آورده شده و در عوض صفحات قدیمی به دیسک منتقل می‌شوند.

توجه: در این روش باید مشخص شود کدام صفحات در حافظه و کدام صفحات بر روی دیسک قرار دارند. برای این منظور روش‌های مختلفی وجود دارد اما عموماً از یک بیت در جدول صفحه استفاده می‌کنند. این بیت که آن را بیت اعتباری می‌نامیم، مشخص می‌کند صفحه موردنظر در حافظه قرار دارد یا خیر. برای مثال اگر مقدار این بیت به ازای یک درایه در جدول صفحه یک بود، به این معناست که صفحه موردنظر معتبر (valid) است و در حافظه قرار دارد، اما اگر این بیت صفر بود به این معناست که صفحه موردنظر نامعتبر (invalid) است و بر روی دیسک قرار دارد.

توجه: اگر فرآیندی به یکی از صفحاتش که در حافظه موجود نیست (بیت اعتبار آن با مقدار نامعتبر پر شده است) نیاز داشته باشد، یک وقفه خطای صفحه (Page Fault) رخ می‌دهد که سیستم عامل باید برای این صفحه، یک قاب در حافظه تهیه کرده و آن را به حافظه منتقل کند.

توجه: آدرسی که توسط فرآیند مورد ارجاع قرار می‌گیرد، آدرس مجازی نام دارد و آدرس‌های حافظه اصلی را آدرس‌های حقیقی گویند. با این تعریف محدوده آدرس‌های مجازی که یک فرآیند می‌تواند به آنها ارجاع کند، فضای آدرس مجازی نام دارد و محدوده آدرس‌های حقیقی موجود در یک سیستم را فضای آدرس حقیقی آن کامپیوتر گویند. هنگامی که فرآیندی در حال اجراست، آدرس‌های مجازی باید به آدرس‌های حقیقی تبدیل شوند.

توجه: هنگامی که یک صفحه با وقفه نقص صفحه مواجه شد، سیستم عامل باید هر چه سریع‌تر آن را به یکی از قاب‌های حافظه منتقل کند. در این حالت اگر هیچ قاب آزادی در حافظه موجود نباشد، یکی از صفحات باید به دیسک منتقل شود تا یک قاب حافظه برای صفحه جدید آزاد گردد. با این شرط، چگونگی انتخاب یک صفحه برای ترک حافظه بسیار مهم است و تأثیر مستقیمی بر کارایی و تعداد وقفه‌های نقص صفحه در آینده دارد.

توجه: به جدول صفحه موجود در سطح اول از یک ساختار چندسطحی، جدول صفحه **ریشه** یا جدول صفحه بیرونی گفته می‌شود. جدول صفحه ریشه در حافظه اصلی و درون یکی از قاب‌های حافظه اصلی قرار دارد. صفحه مربوط به جدول صفحه ریشه هرگز نباید از حافظه اصلی خارج شود، مگر به طور کامل مورد استفاده نباشد. به بیان دیگر جدول صفحه ریشه، اغلب در حافظه اصلی قرار دارد و جداول صفحه بعدی (جداول صفحه جزئی) در صورت نیاز به حافظه اصلی منتقل می‌گردند و تا زمانیکه مورد نیاز و رجوع هستند نباید از حافظه اصلی خارج گردند، در غیر اینصورت در دیسک قرار می‌گیرند. برای یک فرآیند، جدول صفحه ریشه اغلب داخل حافظه اصلی است، اما سایر جداول جزئی در سطوح بعدی، در صورت نیاز و عدم وجود در حافظه اصلی پس از یک نقص صفحه حاصل از عدم وجود صفحه مربوط به جدول صفحه جزئی مورد نیاز به حافظه اصلی بارگذاری می‌گردند. دقت کنید که این نقص صفحه حاصل از عدم وجود صفحه مربوط به جدول صفحه جزئی مورد نیاز است. و نه نقص صفحه مربوط به صفحات یک

khalilifar.ir

فرآیند. در روند تبدیل آدرس مجازی به فیزیکی پس از جور شدن همه جداول صفحه جزئی مورد نیاز و آوردن آنها به حافظه اصلی، ممکن است در آخرین سطح از ساختار چند سطحی، جلوی آدرس مجازی مورد نظر مقداری برای قاب آن نباشد، که این رویداد منجر به ایجاد نقص صفحه مربوط به صفحات فرآیند می‌گردد که مطابق روال مربوط اینبار باید صفحه مورد نظر در حافظه اصلی بارگذاری گردد. برای نمونه، در مثال دوم که به صورت جدول صفحه دو سطحی است، فرض کنید جلوی آدرس مجازی 7، شماره قاب 5 نباشد و به جای مقدار 5 مقدار NULL باشد، در اینصورت هرچند جداول صفحه جزئی مورد نیاز درون حافظه اصلی قرار دارند، اما اینبار صفحه مربوط به صفحه مورد نیاز یک فرآیند درون حافظه اصلی قرار ندارد، که این مساله منجر به نقص صفحه مربوط به صفحه مورد نظر یک فرآیند می‌گردد. اما اگر در روند تبدیل آدرس مجازی به فیزیکی پس از جور شدن همه جداول صفحه جزئی مورد نیاز و آوردن آنها به حافظه اصلی، در آخرین سطح از ساختار چند سطحی، جلوی آدرس مجازی مورد نظر مقداری برای قاب آن باشد، آنگاه این مقدار در سمت چپ مقدار آفست قرار می‌گیرد و آدرس فیزیکی خلق می‌گردد.
توجه: در طرح جدول‌های صفحه چندسطحی، چون ترجمه آدرس از جدول ریشه(بیرونی) شروع شده و به طرف سطوح بعدی حرکت می‌کند، این طرح به جدول صفحه نگاشت **پیشرو** نیز معروف است.

در صورت سوال مطرح شده است که در یک ساختار حافظه، سیستم صفحه‌بندی سه سطحی از آدرس مجازی مطابق فرمت زیر استفاده می‌کند، همچنین مطرح شده است که اگر نرخ برخورد هر سطح 90% باشد، احتمال برخورد تبدیل یک آدرس مجازی به فیزیکی چقدر است؟ البته با فرض اینکه احتمال برخورد هر سطح مستقل از احتمال برخورد در سطوح دیگر باشد.

P3	P2	P1	d

در روند تبدیل آدرس مجازی به فیزیکی، نرخ برخورد 90% در سطح اول، یعنی در 90% موارد جدول صفحه جزئی سطح دوم مورد نیاز درون حافظه اصلی قرار دارد یعنی در 90% موارد نقص صفحه حاصل از عدم وجود صفحه مربوط به جدول صفحه جزئی سطح دوم مورد نیاز رخ نمی‌دهد.

در روند تبدیل آدرس مجازی به فیزیکی، نرخ برخورد 90% در سطح دوم، یعنی در 90% موارد جدول صفحه جزئی سطح سوم مورد نیاز درون حافظه اصلی قرار دارد یعنی در 90% موارد نقص صفحه حاصل از عدم وجود صفحه مربوط به جدول صفحه جزئی سطح سوم مورد نیاز رخ نمی‌دهد.

در روند تبدیل آدرس مجازی به فیزیکی، نرخ برخورد 90% در سطح سوم، یعنی در 90% موارد صفحات مورد نیاز مربوط به یک فرآیند درون حافظه اصلی قرار دارد و برای آنها مقدار قاب نیز مشخص شده است. یعنی در 90% موارد نقص صفحه حاصل از عدم وجود صفحات مورد نیاز مربوط به یک فرآیند رخ نمی‌دهد.

khalilifar.ir

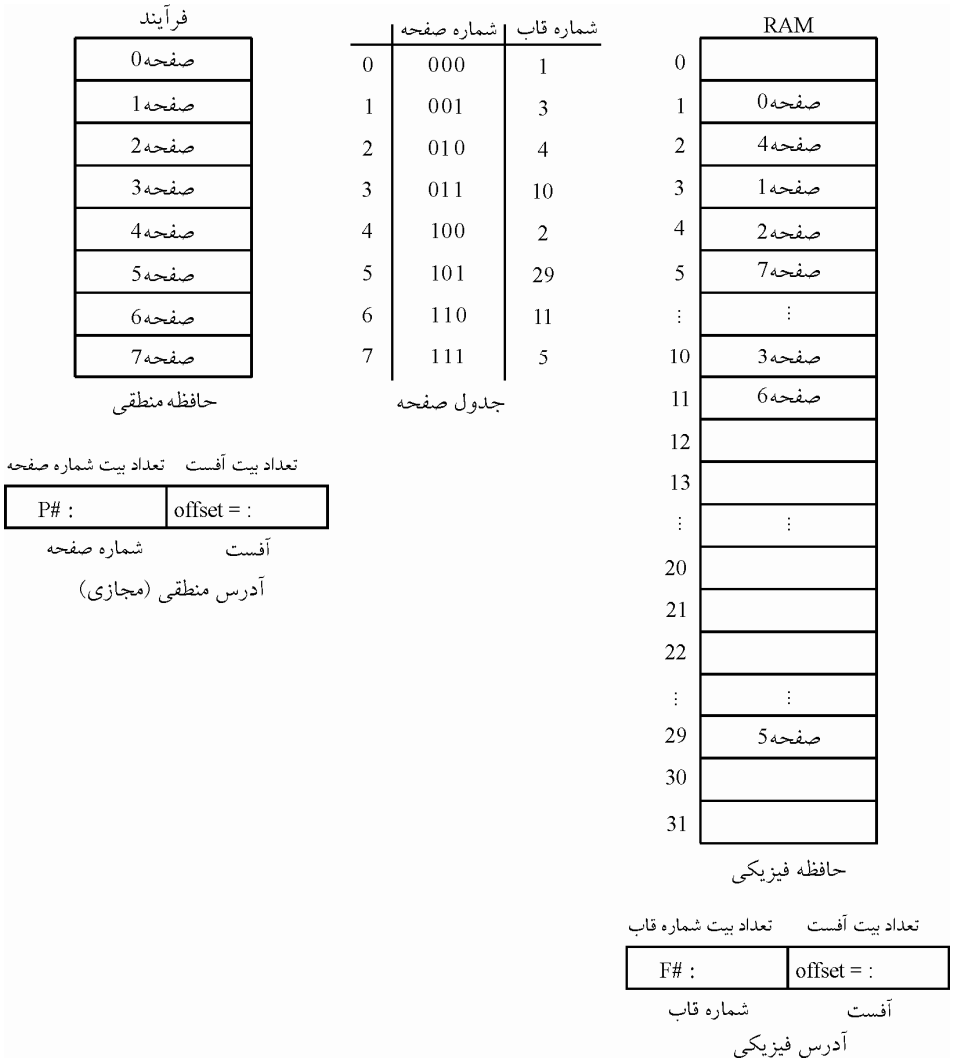
$0.9 \times 0.9 \times 0.9 = 0.729$ = احتمال برخورد تبدیل یک آدرس مجازی به فیزیکی

بنابراین گزینه سوم پاسخ سوال خواهد بود.

۱۱- گزینه (۳) صحیح است.

در صورت سوال مطرح شده است که اندازه فضای آدرس منطقی، 8 صفحه 1024 بایتی و اندازه حافظه فیزیکی 32 قاب است. همچنین خواسته شده است طول آدرس منطقی و آدرس فیزیکی محاسبه گردد.

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بندی به صورت زیر است:



khalilifar.ir

$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}} = 8$$

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = 32$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^8 = 3bit$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{32} = 5bit$$

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{1024} = 10bit$$

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد، به صورت زیر:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب = عرض جدول صفحه

در سوال مطرح شده 0 بیت مربوط به بیت‌های کنترلی و 5 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس: عرض جدول صفحه فوق برابر $5bit + 0bit = 5bit$ می‌باشد.

همانطور که گفتیم، اندازه جدول صفحه، از رابطه زیر محاسبه می‌گردد:

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

که مطابق رابطه فوق داریم:

$$\text{اندازه جدول صفحه} = 8 \times 5bit = 40bit = 5Byte$$

همچنین، اندازه آدرس‌های منطقی (مجازی) و فیزیکی به صورت زیر است:

$3bit + 10bit = 13bit$ = تعداد بیت آفست + تعداد بیت شماره صفحه = طول آدرس منطقی

$5bit + 10bit = 15bit$ = تعداد بیت آفست + تعداد بیت شماره قاب = طول آدرس فیزیکی

همچنین داریم:

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}} = 2^{\text{تعداد بیت آدرس منطقی}}$$

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{13} = 2^3 \times 2^{10} = 2^{13} = 8KB$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{\text{تعداد بیت شماره قاب}} \times 2^{\text{تعداد بیت آفست}} = 2^{\text{تعداد بیت آدرس فیزیکی}}$$

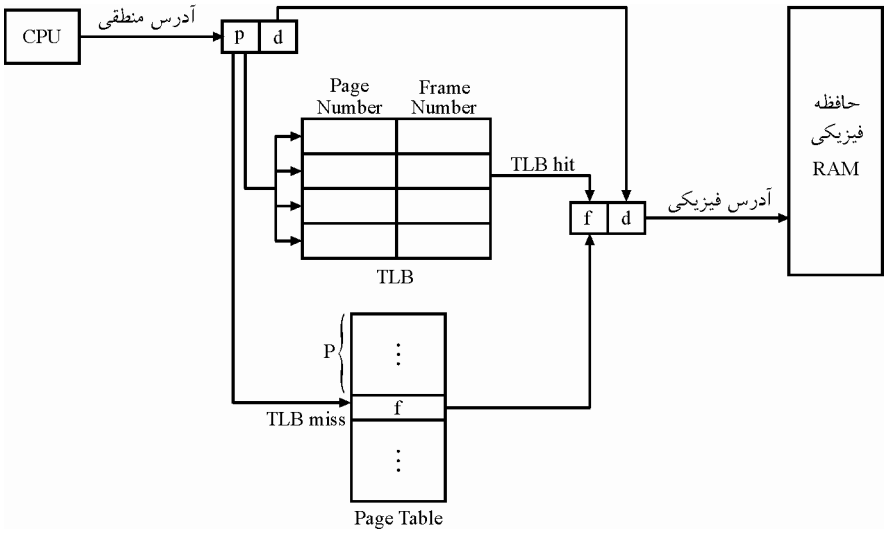
$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{15} = 2^5 \times 2^{10} = 2^{15} = 32\ KB$$

khalilifar.ir

توجه: بنابراین اندازه آدرس منطقی (مجازی) و فیزیکی (حقیقی) الزاماً برابر نیست. بنابراین گزینه سوم پاسخ سوال خواهد بود.

۱۲- گزینه (۴) صحیح است.

بر اساس صورت سوال، در **گزینه اول** گفته شده‌است که اگر تعداد خانه‌های TLB بیشتر شود نرخ برخورد در TLB بیشتر می‌شود. این عبارت درست است. هنگامی که جداول صفحه در حافظه اصلی باشند برای دسترسی به یک خانه حافظه، باید دو یا چند بار به حافظه سر زد. در اینصورت وقتی جداول به صورت چند سطحی پیاده‌سازی شوند، این تعداد بیشتر هم می‌شود و سرعت دسترسی به حافظه به شدت کاهش می‌یابد. در این حالت برای افزایش سرعت دسترسی به حافظه از تکنیک TLB استفاده می‌شود. TLB از حافظه‌های با سرعت بسیار بالا و گران قیمت ساخته شده است که در واقع مجموعه‌ای از رجیسترها موسوم به Associative Register هستند. وقتی فرآیندی جهت اجرا انتخاب شود، بخشی از درایه‌های جدول صفحه آن به TLB منتقل می‌شود. در این حالت وقتی CPU یک آدرس منطقی تولید می‌کند، شماره صفحه آن ابتدا در TLB جست‌وجو می‌شود، اگر شماره صفحه در TLB یافت شد که آدرس قاب متناظر به دست می‌آید، اما اگر شماره صفحه در TLB نباشد آن‌گاه طبق روال قبل به سراغ جدول صفحه در حافظه می‌رویم. نکته مهم در مورد TLB این است که عمل جست‌وجو در یک لحظه و همزمان در تمام سطرها صورت می‌گیرد. به همین دلیل سرعت دسترسی به آن بسیار بالاست. این مساله برای کاربرنهایی اهمیت دارد. کاربر نهایی زمان پاسخ کوتاه را در انجام کارها انتظار دارد.



توجه: با توجه به ساختار TLB و گران بودن آن، فقط بخش کوچکی از جدول صفحه این شانس را پیدا می‌کند که به TLB وارد شود. به این ترتیب اگر هنگام تبدیل آدرس، شماره صفحه در

khalilifar.ir

TLB موجود باشد اصطلاحاً HIT و در غیر این صورت MISS رخ داده است. «**بنابراین اگر تعداد خانه‌های TLB بیشتر شود، این رویداد منجر به این می‌شود که شماره صفحات بیشتری از یک فرآیند درون TLB یافت شود که این یعنی افزایش نرخ برخورد.**» پس «**اگر تعداد خانه‌های TLB بیشتر شود نرخ برخورد در TLB بیشتر می‌شود.**»

همچنین ضریب موفقیت یا نسبت اصابت به صورت زیر محاسبه می‌شود:

$$\text{HIT Ratio} = \frac{\text{HIT}}{\text{HIT} + \text{MISS}}$$

مثال: فرض کنید در یک سیستم، زمان دسترسی به حافظه 60 نانو ثانیه و زمان دسترسی به TLB، برابر 5 نانو ثانیه باشد. اگر احتمال وجود شماره صفحه در TLB برابر 80% باشد، زمان دسترسی به حافظه چقدر است؟

پاسخ: هنگامی که یک درخواست برای حافظه از راه می‌رسد، دو حالت ممکن است رخ دهد:

در حالت اول، شماره صفحه در TLB موجود است، که در این صورت یک بار در TLB و یک بار به حافظه رجوع می‌کنیم، یعنی جمعاً 60+5=65ns.

اما در حالت دوم، شماره صفحه در TLB پیدا نمی‌شود، در این صورت یک بار به TLB و دو بار به حافظه رجوع می‌کنیم (یک بار برای رجوع به جدول صفحه و پیدا کردن شماره صفحه و یک بار هم برای دستیابی به داده مورد نظر)، یعنی جمعاً 60+60+5=125ns.

لذا زمان کل دسترسی مؤثر به حافظه با توجه به ضریب موفقیت برابر است با:

$$0.8 \times 65 + 0.2 \times 125 = 77\text{ns}$$

بر اساس صورت سوال، در **گزینه دوم** گفته شده‌است که اگر اندازه صفحه‌ها بزرگتر شود نرخ برخورد در TLB بیشتر می‌شود. این عبارت درست است. اگر اندازه صفحه‌ها بزرگتر شود، تعداد صفحات و در نتیجه تعداد درایه‌های جدول صفحه کمتر می‌شود. یعنی فضای نمونه کمتر شده و در نتیجه احتمال و نرخ برخورد در TLB بیشتر می‌شود. وقتی فرآیندی جهت اجرا انتخاب می‌شود، بخشی از درایه‌های جدول صفحه آن به TLB منتقل می‌شود. حال اگر تعداد صفحات و تعداد درایه‌های جدول صفحه به دلیل افزایش اندازه صفحات، کمتر و کمتر شود، آنگاه تعداد درایه‌های بیشتر و بیشتری از جدول صفحه این شانس را خواهند داشت تا وارد فضای محدود TLB شوند و در بهترین حالت شاید همه وارد TLB شوند و این یعنی افزایش نرخ برخورد در TLB. پس «**اگر اندازه صفحه‌ها بزرگتر شود نرخ برخورد در TLB بیشتر می‌شود.**»

بر اساس صورت سوال، در **گزینه سوم** گفته شده‌است که برای صفحه‌های با اندازه ثابت، اگر طول آدرس فیزیکی بیشتر شود نرخ برخورد در TLB تغییر نمی‌کند. این عبارت درست است. نرخ برخورد در TLB تابعی از تعداد خانه‌های TLB و تعداد صفحات و در نتیجه تعداد درایه‌های

khalilifar.ir

جدول صفحه است. یعنی مطابق آنچه در گزینه‌های اول و دوم گفتیم اگر تعداد خانه‌های TLB بیشتر شود نرخ برخورد در TLB بیشتر می‌شود. و اگر اندازه صفحه‌ها بزرگتر شود نرخ برخورد در TLB بیشتر می‌شود. افزایش یا کاهش نرخ برخورد در TLB رابطه‌ای با افزایش یا کاهش طول آدرس فیزیکی ندارد. و اصلا به هم مرتبط نیستند، پس «**برای صفحه‌های با اندازه ثابت، اگر طول آدرس فیزیکی بیشتر شود نرخ برخورد در TLB تغییر نمی‌کند.**»

بر اساس صورت سوال، در **گزینه چهارم** گفته شده‌است که برای صفحه‌های با اندازه ثابت، با افزایش اندازه قطعه (segment)، نرخ برخورد در TLB بیشتر می‌شود. این عبارت نادرست است. در راه حل قطعه‌بندی همراه با صفحه‌بندی، قطعات تولید شده توسط برنامه‌نویس یا کامپایلر، توسط سیستم عامل صفحه‌بندی می‌شود. در این حالت به ازای هر فرآیند یک جدول قطعه ایجاد می‌شود که تعداد درایه‌های جدول قطعه، بستگی به اندازه فرآیند و تعداد قطعات حاصل دارد. همچنین به ازای هر قطعه (segment)، یک جدول صفحه ایجاد می‌شود که تعداد درایه‌های جدول صفحه، بستگی به اندازه قطعه و تعداد صفحات حاصل دارد. جهت افزایش سرعت نگاشت آدرس منطقی به فیزیکی می‌توان از TLB نیز استفاده نمود. TLB از حافظه‌های با سرعت بسیار بالا و گران قیمت ساخته شده است که در واقع مجموعه‌ای از رجیسترها موسوم به Associative Register هستند. در سیستم قطعه‌بندی همراه با صفحه‌بندی، وقتی فرآیندی جهت اجرا انتخاب شود، بخشی از درایه‌های جدول قطعه و جدول صفحه آن به TLB منتقل می‌شود. در این حالت وقتی CPU یک آدرس منطقی تولید می‌کند، شماره قطعه و شماره صفحه آن ابتدا در TLB جست‌وجو می‌شود، اگر شماره قطعه و شماره صفحه در TLB یافت شد که آدرس قاب متناظر در حافظه اصلی به دست می‌آید، اما اگر شماره قطعه و شماره صفحه در TLB نباشد، آن‌گاه طبق روال معمول به سراغ جدول قطعه و جدول صفحه در حافظه اصلی می‌رویم. نکته مهم در مورد TLB این است که عمل جست‌وجو در یک لحظه و همزمان در تمام سطرها صورت می‌گیرد. به همین دلیل سرعت دسترسی به آن بسیار بالاست. این مساله برای کاربرنهایی اهمیت دارد. کاربر نهایی زمان پاسخ کوتاه را در انجام کارها انتظار دارد. همانطور که گفتیم به ازای هر قطعه (segment)، یک جدول صفحه ایجاد می‌شود که تعداد درایه‌های جدول صفحه، بستگی به اندازه قطعه و تعداد صفحات حاصل دارد. اگر اندازه قطعه (segment)، بزرگتر شود تعداد صفحات و در نتیجه تعداد درایه‌های جدول صفحه مربوط به قطعه مورد نظر بیشتر می‌شود. یعنی فضای نمونه بیشتر شده و در نتیجه احتمال و نرخ برخورد در TLB کمتر می‌شود. وقتی فرآیندی جهت اجرا انتخاب شود، بخشی از درایه‌های جدول قطعه و جدول صفحه آن به TLB منتقل می‌شود. حال اگر تعداد صفحات و تعداد درایه‌های جدول صفحه قطعه مورد نظر به دلیل افزایش اندازه قطعات، بیشتر و بیشتر شود، آنگاه تعداد درایه‌های کمتری از جدول صفحه قطعه مورد نظر این شانس را خواهند داشت تا وارد فضای محدود TLB شوند و این یعنی کاهش نرخ برخورد در

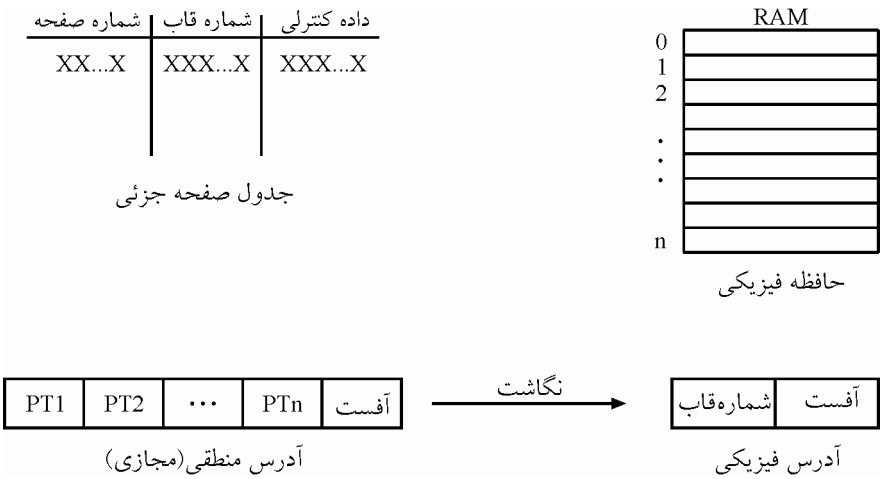
khalilifar.ir

TLB. پس «**برای صفحه‌های با اندازه ثابت، با افزایش اندازه قطعه (segment)، نـرخ برخـورد در TLB کمتر می‌شود و نه بیشتر.**»

۱۳- گزینه (۱) صحیح است.

در اینجا برای جدول صفحه جزئی محدودیتی به اندازه یک قاب (صفحه) داریم.

بنابراین اندازه جدول صفحه جزئی برابر اندازه قاب (صفحه) می‌باشد. بنابراین برای محاسبه تعـداد سطرهای جدول صفحه جزئی، کافی است، اندازه قاب که برابر اندازه جدول صفحه جزئی است بر اندازه عرض جدول صفحه جزئی تقسیم گردد. به شکل زیر توجه کنید:



توجه: عرض جدول صفحه همواره برابر حاصل جمع تعداد بیت‌های کنترلـی و تعـداد بیـت‌هـای شماره قاب است، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد.

بنابراین داریم:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

$4B$ = عرض جدول صفحه جزئی

توجه: مطابق فرض سؤال، هر مدخل جدول صفحه (عرض جدول صفحه جزئی) 4 بایت در نظـر گرفته شده است.

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^3 \times 2^{10}B}{2^2 B} = 2^{11} = 2048$$

$2^{46}B$ =اندازه فرآیند (فضای آدرس مجازی)

$8KB = 8192B = 2^3 \times 2^{10}B = 2^{13}B$ =اندازه صفحه = اندازه قاب

khalilifar.ir

$$\text{تعداد صفحات فرآیند : f} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه}} = \frac{2^{46}}{2^{13}} = 2^{33}$$

$$\text{تعداد سطرهای جدول صفحه جزئی : r} = 2048 = 2^{11}$$

حال اطلاعات کافی برای محاسبه تعداد سطوح جدول صفحه چند سطحی را در اختیار داریم:

روش تجزیه

تعداد صفحات فرآیند باید در اندازه r(2^{11}) تجزیه گردد:

$$\text{تعداد صفحات فرآیند} = 2^{33} = 2^{\overset{\uparrow}{\textcircled{11}}\text{PT1}} \times 2^{\overset{\uparrow}{\textcircled{11}}\text{PT2}} \times 2^{\overset{\uparrow}{\textcircled{11}}\text{PT3}}$$

توجه: تعداد عملوندها برابر 3 است، بنابراین تعداد سطوح جدول چند سطحی نیز برابر 3 است.

روش لگاریتم

$$\text{تعداد سطوح جدول چند سطحی} = d = \left\lceil \log_r^f \right\rceil = \left\lceil \log_{2^{11}}^{2^{33}} \right\rceil = 3$$

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح سوم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{2^{33}}{2^{11}} = 2^{22}$$

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد جداول صفحه جزئی در سطح سوم}}{r} = \frac{2^{22}}{2^{11}} = 2^{11}$$

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{r} = \frac{2^{11}}{2^{11}} = 1$$

توجه: سطح اول، یک جدول به حساب می‌آید، که 2^{11} سطر دارد.

توجه: تعداد تقسیم متوالی برابر 3 است، بنابراین تعداد سطوح جدول صفحه چند سطحی برابر 3 است.

$$\text{تعداد بیت آفست} = \log_2^{\text{اندازه صفحه}} = \log_2^{2^{13}} = 13 \text{ بیت}$$

بنابراین شکل آدرس منطقی (مجازی) به صورت زیر خواهد بود:

PT1	PT2	PT3	آفست
11 بیت	11 بیت	11 بیت	13 بیت
33 بیت			

مطابق آنچه گفتیم سه دسترسی به جداول صفحه جزئی سطح اول، دوم و سوم برای ترجمه آدرس و یک دسترسی به داده اصلی (مقصد مورد نظر) لازم است، که مجموع آن شامل چهار دسترسی به حافظه می‌گردد. بنابراین برای خواندن یک کلمه 32 بیتی نیاز به چهار دسترسی به حافظه است.

khalilifar.ir

۱۴- گزینه (۱) صحیح است.

در تکنیک صفحه‌بندی، برای هر فرآیند یک جدول صفحه تشکیل می‌شود که در آن به ازای همه صفحه‌های یک فرآیند، درایه وجود دارد و هر درایه مشخص می‌کند که کدام قاب فیزیکی به این صفحه اختصاص یافته است. در این روش وقتی فرآیندها بزرگ باشند، هزینه نگهداری جداول صفحه بسیار زیاد می‌شود، در ضمن به ازای هر فرآیند نیز باید یک جدول صفحه داشته باشیم. به عبارتی وقتی تعداد و اندازه فرآیندها بزرگ شود، این روش مقرون به صرفه نیست.

برای حل این مشکل از جداول صفحه معکوس استفاده می‌کنیم. در این حالت به جای اینکه در جداول صفحه مربوط به هر فرآیند، به ازای هر صفحه مجازی یک درایه داشته باشیم، به ازای هر قاب در حافظه فیزیکی یک درایه در جدول صفحه معکوس نگهداری می‌کنیم. در واقع به ازای هر قاب حافظه اصلی اینکه در حال حاضر کدام صفحه مربوط به کدام فرآیند در این قاب ذخیره شده است، نگهداری می‌شود. بنابراین تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی(اصلی) است.

با استفاده از تکنیک جدول صفحه معکوس، مقدار زیادی در حافظه صرفه‌جویی می‌شود اما تبدیل آدرس مجازی به فیزیکی سخت‌تر و زمان‌گیرتر می‌شود.

در واقع وقتی یک فرآیند با PID مختص به خود به صفحه مجازی P# مراجعه می‌کند، سخت‌افزار دیگر نمی‌تواند از P# به عنوان اندیس جدول صفحه استفاده کند و صفحه فیزیکی را بیابد و از این جهت باید سرتاسر جدول صفحه وارونه را برای یافتن درایه (PID, P#) جستجو کند.

تعداد درایه‌های جدول صفحه معکوس، مطابق رابطه زیر محاسبه می‌گردد:

تعداد قاب‌های حافظه فیزیکی = تعداد درایه‌های جدول صفحه معکوس

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

همانطور که گفتیم تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی است، بنابراین تعداد درایه‌های جدول صفحه معکوس به صورت زیر خواهد بود:

$$\text{تعداد درایه‌های جدول صفحه معکوس} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

بنابراین گزینه اول پاسخ سوال است.

همچنین اندازه حافظه منطقی (فرآیند) مطابق رابطه زیر قابل محاسبه است:

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{\text{تعداد بیت آدرس منطقی}} = 2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{34} = 2^{23} \times 2^{11} = 2^{34}\ \text{Byte} = 2^4 \times 2^{30} = 16\ \text{GB}$$

khalilifar.ir

۱۵- گزینه ( ) صحیح است.

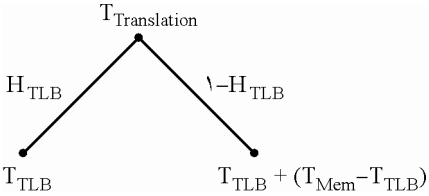
مطابق فرض سوال دو سیستم S_1 و S_2 همزمان با ارسال درخواست به TLB، آنرا به حافظه اصلی نیز ارسال می‌کنند تا در صورت عدم یافتن آدرس در TLB، زمان پاسخ کاهش یابد. به عبارت دیگر نحوه عملکرد دو سیستم S_1 و S_2 **موازی** در نظرگرفته شده‌است، به صورت زیر:

$\mathbf{T_{Translation}}$: میانگین زمان ترجمه در حالت **موازی**

توجه: مطابق فرض سوال در **بخش ترجمه** یعنی $T_{Translation}$، دو عنصر T_{TLB} و T_{Mem} برای ترجمه وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

برای محاسبه $T_{Translation}$ میانگین T_{TLB} و T_{Mem} را محاسبه می‌کنیم که همان $T_{Translation}$ است. به صورت زیر:

برای بدست آوردن $T_{Translation}$ درخت زیر را در نظر بگیرید:



پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Translation}$ خواهیم داشت.

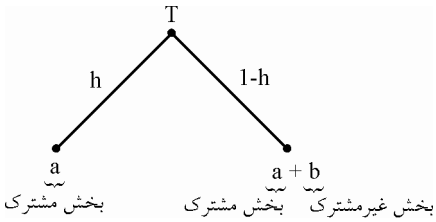
$$T_{Translation} = T_1 = T_{TLB} + (1\text{-}H_{TLB}) \times (T_{Mem} - T_{TLB})$$

$$T_{Translation} = T_1 = 120 + (1\text{-}0.6) \times (T_{Mem} - 120) = 120 + 0.4 \times T_{Mem} - 48 = 72 + 0.4 \times T_{Mem}$$

$$T_{Translation} = T_2 = T_{TLB} + (1\text{-}H_{TLB}) \times (T_{Mem} - T_{TLB})$$

$$T_{Translation} = T_2 = 150 + (1\text{-}0.8) \times (T_{Mem} - 150) = 150 + 0.2 \times T_{Mem} - 30 = 120 + 0.2 \times T_{Mem}$$

توجه: جهت ساده‌سازی رابطه درخت فوق، همواره می‌توان از **اتحاد درخت احتمال** به صورت زیر، استفاده نمود:



$$T = \text{بخش مشترک} + (1-h) \times \text{بخش غیرمشترک}$$

$$T = a + (1-h) \times b$$

khalilifar.ir

پس از ساده‌سازی درخت مطرح شده براساس **اتحاد درخت احتمال**، رابطه زیر را بـرای محاسـبه $T_{Translation}$ خواهیم داشت:

$$T_{Translation} = T_1 = T_{TLB} + (1\text{-}H_{TLB}) \times (T_{Mem} - T_{TLB})$$

$$T_{Translation} = T_1 = 120 + (1\text{-}0.6) \times (T_{Mem} - 120) = 120 + 0.4 \times T_{Mem} - 48 = 72 + 0.4 \times T_{Mem}$$

$$T_{Translation} = T_2 = T_{TLB} + (1\text{-}H_{TLB}) \times (T_{Mem} - T_{TLB})$$

$$T_{Translation} = T_2 = 150 + (1\text{-}0.8) \times (T_{Mem} - 150) = 150 + 0.2 \times T_{Mem} - 30 = 120 + 0.2 \times T_{Mem}$$

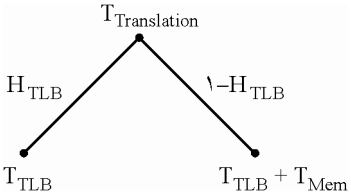
همچنین مطابق فـرض سـوال سیسـتم S_3 پـس از دریافـت پاسـخ از TLB و عـدم یـافتن آدرس، درخواست را به حافظه اصلی ارسال می‌کند. به عبارت دیگـر نحـوه عملکـرد سیسـتم S_3 **ترتیبـی** (سری) در نظرگرفته شده‌است، به صورت زیر:

$\mathbf{T_{Translation}}$: میانگین زمان ترجمه در حالت **ترتیبی**

توجه: مطابق فرض سوال در **بخـش ترجمـه** یعنـی $T_{Translation}$، دو عنصـر T_{TLB} و T_{Mem} بـرای ترجمه وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

برای محاسبه $T_{Translation}$ میانگین T_{TLB} و T_{Mem} را محاسبه می‌کنیم که همـان $T_{Translation}$ اسـت. به صورت زیر:

برای بدست آوردن $T_{Translation}$ درخت زیر را در نظر بگیرید:



پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Translation}$ خواهیم داشت.

$$T_{Translation} = T_3 = T_{TLB} + (1\text{-}H_{TLB}) \times T_{Mem}$$

$$T_{Translation} = T_3 = 120 + (1\text{-}0.7) \times T_{Mem} = 120 + 0.3 \times T_{Mem}$$

خواسته سوال این است که **کمترین** و **بیشترین** میانگین زمان پاسخ این سـه سیسـتم بـه ترتیـب از راست به چپ چند نانوثانیه است؟

که مطابق آنچه پیشتر گفتیم، داریم:

$$T_{Translation} = T_1 = 120 + (1\text{-}0.6) \times (T_{Mem} - 120) = 120 + 0.4 \times T_{Mem} - 48 = 72 + 0.4 \times T_{Mem}$$

$$T_{Translation} = T_2 = 150 + (1\text{-}0.8) \times (T_{Mem} - 150) = 150 + 0.2 \times T_{Mem} - 30 = 120 + 0.2 \times T_{Mem}$$

$$T_{Translation} = T_3 = 120 + (1\text{-}0.7) \times T_{Mem} = 120 + 0.3 \times T_{Mem}$$

khalilifar.ir

متاسفانه مقدار T_{Mem} در صورت سوال توسط طراح محترم بیان نشده است، که باعث شده ادامه حل مساله امکان‌پذیر نباشد، اما اگر مقدار T_{Mem} را برابر 600 ns در نظر بگیریم، آنگاه نتایج زیر را خواهیم داشت:

$T_{Translation} = T_1 = 72 + 0.4 \times T_{Mem} = 72 + 240 = 312$ ns

$T_{Translation} = T_2 = 120 + 0.2 \times T_{Mem} = 120 + 120 = 240$ ns

$T_{Translation} = T_3 = 120 + 0.3 \times T_{Mem} = 120 + 180 = 300$ ns

اگر مقدار T_{Mem} از 600 ns بیشتر شود، آنگاه **بیشترین** میانگین زمان پاسخ که مربوط به T_1 و برابر 312 است، از 312 بیشتر می‌شود که در گزینه‌ها بیشتر از مقدار 312 وجود ندارد. بنابراین در این حالت **کمترین** و **بیشترین** میانگین زمان پاسخ این سه سیستم به ترتیب برابر 312 و 240 نانوثانیه است که در هیچیک از گزینه‌ها وجود ندارد.

توجه: منظور طرح از زمان پاسخ، محاسبه زمان ترجمه یعنی $T_{Translation}$ در شرایط مطرح شده است، و نه محاسبه زمان دسترسی به یک مقصد مورد نظر، بنابراین در محاسبات فوق از زمان مقصد یعنی $T_{Destination}$ استفاده نکردیم.

توجه: سازمان سنجش آموزش کشور در کلید اولیه خود ابتدا گزینه دوم را به عنوان پاسخ اعلام نمود، سپس در کلید نهایی نظر خود را عوض کرد و کلاً تست را حذف نمود، که عمل درستی را انجام داده است که البته عمل درست‌تر آن است که سؤال از ابتدا، درست طرح شود.

khalilifar.ir

تست‌های فصل پنجم: مدیریت حافظه مجازی

۱- **با توجه به بحث Copy-On-Write بین فرآیندهای پدر (Parent) و فرزند (Child) در فراخوان سیستمی fork، در راستای افزایش کارایی، کدام جمله در مورد تقدم و تأخر اجرای این فرآیندها در لحظه ایجاد فرآیند فرزند صحیح است؟** (مهندسی کامپیوتر – دولتی ۸۹)

۱) با توجه به آگاهی زمان‌بند از محتوای (برنامه) فرآیند پدر، بهتر است فرآیند فرزند زودتر اجرا شود.

۲) با توجه به آگاهی زمان‌بند از محتوای (برنامه) فرآیند پدر، بهتر است فرآیند پدر زودتر اجرا شود.

۳) با توجه به عدم آگاهی زمان‌بند از محتوای (برنامه) فرآیند فرزند، بهتر است فرآیند پدر زودتر اجرا شود.

۴) با توجه به عدم آگاهی زمان‌بند از محتوای (برنامه) فرآیند فرزند، بهتر است فرآیند فرزند زودتر اجرا شود.

۲- **یک حافظه مجازی با این مشخصات در نظر بگیرید. زمان دسترسی حافظه 50ns، زمان دستیابی TLB 2ns، نسبت اصابت TLB (hit ratio)، 98% و احتمال خطای صفحه برای کل دسترسی‌ها به حافظه 2×10^{-6} است. زمان انتقال صفحه از دیسک را 10ms فرض کنید. برای سرعت بخشیدن به این حافظه از حافظه پنهان (cache) با این مشخصات استفاده شده است. زمان دسترسی حافظه پنهان 10ns، نسبت اصابت حافظه پنهان 90%، جریمه هر عدم اصابت در حافظه پنهان 100ns است. میانگین زمان دسترسی به حافظه برای هر آدرس به کدام یک از گزینه‌های زیر نزدیک‌تر است؟** (مهندسی IT- دولتی۸۹)

۱) 75ns　　۲) 45ns

۳) 73ns　　۴) 43ns

۳- **به فرآیندی 4 قاب (frame) تخصیص یافته است. (تمام اعداد دهدهی هستند و همه شماره‌گذاری‌ها از صفر شروع شده است.) زمان آخرین بارشدن یک صفحه در یک قاب، زمان آخرین دستیابی به صفحه، شماره صفحه مربوط به هر قاب، بیت‌های مراجعه (R) و تغییر (M) برای هر قاب در جدول زیر نشان داده شده‌اند (زمان‌ها برحسب ضربان ساعت و از شروع فرآیند در زمان صفر است.)** (مهندسی IT- دولتی۸۹)

khalilifar.ir

شماره صفحه	شماره قاب	زمان بار شدن صفحه در حافظه	زمان آخرین مراجعه	بیت مراجعه (R)	بیت تغییر (M)
2	0	60	160	0	1
1	1	130	161	0	0
0	2	26	163	1	0
3	3	20	162	1	1

یک خطای صفحه برای صفحه 4 رخ داده است. برای هر یک از سیاست‌های مدیریت حافظه FIFO و LRU و Clock (اشاره‌گر روی صفحه صفر است و اولویت با انتخاب صفحه تغییر نیافته است) به ترتیب چه صفحه‌ای برای جایگزینی انتخاب می‌شود؟

۱) 0 و 1 و 1 ۲) 1 و 0 و 2 ۳) 3 و 2 و 1 ۴) 1 و 1 و 2

۴- یک سیستم حافظه مجازی صفحه‌بندی را در نظر بگیرید که از دیسک با زمان دسترسی و انتقال 3 میلی‌ثانیه برای هر صفحه استفاده می‌کند. هر دسترسی به حافظه دارای زمان 50 نانوثانیه است. برای بهبود کارآیی میانگیر دم‌دستی ترجمه (TLB) اضافه شده است که دارای زمان دسترسی 2 نانوثانیه می‌باشد. فرض کنید 98% دسترسی‌ها از طریق TLB انجام می‌گیرد. اگرچه می‌توان میانگین زمان دسترسی به یک آدرس حافظه مجازی را محاسبه نمود، ولی زمان‌های واقعی دسترسی‌ها متعدد هستند و با مقدار میانگین تفاوت دارند. چند زمان واقعی دسترسی عبارتند از: (از اعداد بسیار کوچک در مقابل اعداد بسیار بزرگ صرف‌نظر کنید.) (مهندسی IT- دولتی۹۰)

۱) 52 نانوثانیه، 6 میلی‌ثانیه، 3 میلی‌ثانیه ۲) 50 نانوثانیه، 3.5 میلی‌ثانیه، 100 نانوثانیه

۳) 50 نانوثانیه، 100 نانوثانیه، 7 میلی‌ثانیه ۴) 102 میلی‌ثانیه، 3.5 میلی‌ثانیه، 7 میلی‌ثانیه

۵- راه‌حل‌های سخت‌افزاری متعددی به همراه سیستم عامل به اجرای برنامه‌ها کمک می‌کنند. در اینجا به چهار مورد از این راه‌حل‌ها اشاره شده است. در کدام گزینه هر چهار مورد راه‌حل سخت‌افزاری است؟ (مهندسی IT- دولتی۹۰)

۱) ۱- کنترل جدول TLB، ۲- تبدیل آدرس منطقی به فیزیکی، ۳- ثبت تغییر یافتن (modify) یک صفحه، ۴- تشخیص دستورالعمل غیرمجاز

۲) ۱- اعلام پایان سهم زمانی، ۲- حافظه پنهان (cache)، ۳- تشخیص نبودن صفحه در حافظه اصلی، ۴- بار کردن ثبات‌های مدیریت حافظه

۳) ۱- ثبت دسترسی (access) به یک صفحه، ۲- بردن فرآیند از حالتی (state) به حالت دیگر، ۳- تشخیص سرریز در یک محاسبه، ۴- خروج فرآیند

۴) ۱- تبدیل آدرس قطعه‌بندی شده به آدرس حافظه اصلی، ۲- جلوگیری از اجرای دستورالعمل تعریف نشده، ۳- یافتن صفحه خطا خورده (fault) در دیسک، ۴- حفاظت از فضای حافظه فرآیندها

khalilifar.ir

۶- تبدیل آدرس منطقی به فیزیکی در مدیریت صفحه‌بندی در یک پردازنده در شکل زیر مشاهده می‌شود (صفحه‌بندی دو سطحی). اطلاعات موجود عبارتند از:

اندازه هر درایه (entry) جدول صفحه 4 بایت است.

زمان دسترسی به TLB برابر 2ns

زمان دسترسی به حافظه برابر 50ns

زمان دسترسی به حافظه پنهان (cache) برابر 10ns

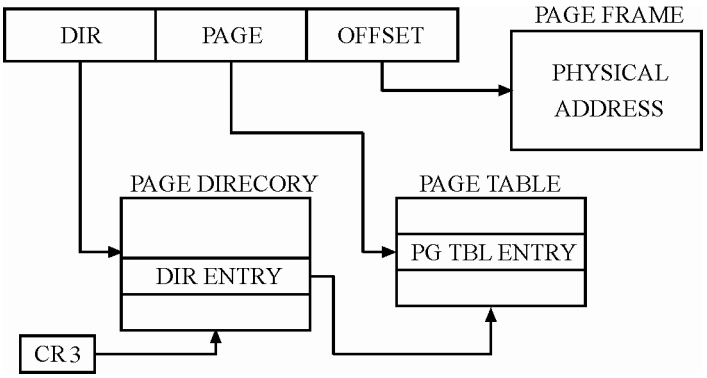
جریمه cache miss برابر 100ns

زمان تبادل یک صفحه بین حافظه و دیسک برابر 5ms

فرمت آدرس منطقی پردازنده



تبدیل آدرس منطقی به فیزیکی در پردازنده



کدام گزینه زمان‌های ممکن برای دسترسی به یک مکان حافظه که با آدرس منطقی مشخص شده است را نشان می‌دهد؟ (مهندسی کامپیوتر – دولتی۹۱)

۱) 32ns, 162ns, 152ns, 482ns, 10ms, 15ms

۲) 32ns, 162ns, 132ns, 282ns, 382ns, 15ms, 10ms

۳) 12ns, 82ns, 162ns, 282ns, 332ns, 5ms, 10ms, 15ms

۴) 12ns, 32ns, 112ns, 132ns, 232ns, 332ns, 5ms, 10ms

۷- تبدیل آدرس منطقی به فیزیکی در مدیریت صفحه‌بندی (دو سطحی) برای یک پردازنده مفروض در شکل زیر مشاهده می‌شود. اطلاعات زیر موجود است: (مهندسی IT- دولتی۹۱)

زمان دسترسی به TLB برابر 2ns

زمان دسترسی به حافظه برابر 50ns

khalilifar.ir

زمان دسترسی به حافظه پنهان (cache) برابر 10ns

جریمه cache miss برابر 100ns

میانگین زمان تبادل صفحه بین حافظه و دیسک برابر 5ms

نسبت اصابت TLB (hit ratio) برابر 98%

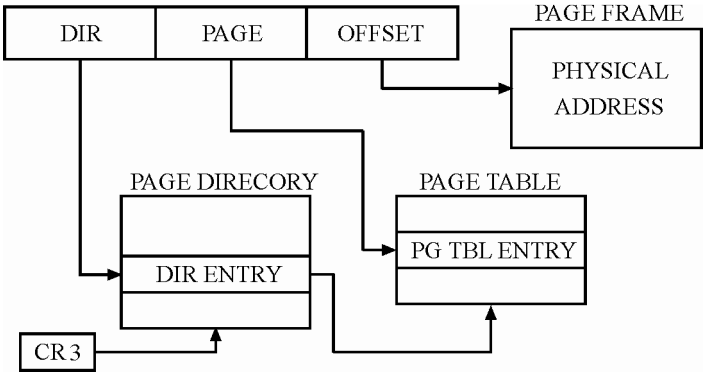
نسبت اصابت (hit ratio) حافظه پنهان برابر 90%

احتمال روی دادن خطای فقدان صفحه برای هر دسترسی به حافظه $p = 10^{-6}$ است.

فرمت آدرس منطقی پردازنده مفروض



تبدیل آدرس منطقی به فیزیکی در پردازنده مفروض



میانگین زمان دسترسی به یک مکان حافظه که با آدرس منطقی مشخص شده به کدام یک از اعداد زیر نزدیک‌تر است؟

۱) 128 نانوثانیه ۲) 57 نانوثانیه ۳) 107 نانوثانیه ۴) 28 نانوثانیه

۸- کدام یک از فن‌های برنامه‌نویسی و ساختارها «مناسب» محیط صفحه‌بندی بر پایه تقاضا هستند؟ و کدام یک مناسب نیستند؟ **(مهندسی IT- دولتی۹۱)**

۱- پشته، ۲- جدول سمبل hashed، ۳- جستجوی ترتیبی، ۴- جستجوی دودویی، ۵- کُد خالص، ۶- عملیات برداری، ۷- کارکرد غیرمستقیم

۱) ۱- نامناسب، ۲- نامناسب، ۳- مناسب، ۴- نامناسب، ۵- مناسب، ۶- نامناسب، ۷- نامناسب

۲) ۱- مناسب، ۲- نامناسب، ۳- مناسب، ۴- نامناسب، ۵- مناسب، ۶- مناسب، ۷- نامناسب

۳) ۱- نامناسب، ۲- نامناسب، ۳- مناسب، ۴- مناسب، ۵- مناسب، ۶- نامناسب، ۷- نامناسب

۴) ۱- مناسب، ۲- مناسب، ۳- مناسب، ۴- مناسب، ۵- مناسب، ۶- نامناسب، ۷- مناسب

---

khalilifar.ir

۹- کدام گزینه زیر درباره جدول صفحه معکوس (Inverted Page Table) درست نیست؟

(مهندسی کامپیوتر – دولتی۹۲)

۱) این نوع جدول زمان نگاشت آدرس منطقی به آدرس فیزیکی را کاهش می‌دهد.
۲) این نوع جدول صفحه سبب کاهش اندازه حافظه فیزیکی جهت ذخیره‌سازی آن می‌شود.
۳) در این نوع جدول صفحه زمان سرویس نقص صفحه (page fault) به دلیل ایجاد یک نقص صفحه دیگر افزایش می‌یابد.
۴) برای این نوع جدول صفحه می‌بایست یک جدول صفحه خارجی نیز ذخیره شود.

۱۰- کدام یک از گزینه‌های زیر درباره مدیریت حافظه با روش صفحه‌بندی درست نیست؟

(مهندسی IT- دولتی۹۲)

۱) کاهش اندازه صفحه سبب کاهش زمان سرویس نقص صفحه می‌شود.
۲) کاهش اندازه صفحه سبب افزایش بهره‌وری حافظه و افزایش زمان I/O می‌شود.
۳) کاهش اندازه صفحه سبب کاهش تکه تکه شدن خارجی (external fragmentation) حافظه می‌شود.
۴) کاهش اندازه صفحه‌ها سبب کاهش تکه تکه شدن داخلی (internal fragmentation) حافظه می‌شود.

۱۱- در یک سیستم مدیریت حافظه که به فناوری copy-on-write مجهز است پس از تولید تعدادی فرزند توسط یک فرآیند، کدام گزینه زیر درست است؟ **(مهندسی IT- دولتی۹۲)**

۱) هر یک از فرزندان و یا فرآیند اصلی در هر لحظه می‌توانند به کار خود پایان دهند.
۲) فرآیند پدر در صورتی می‌تواند به کار خود پایان دهد که فرزندان همه صفحه‌های مربوط به خود را تغییر داده باشند.
۳) فرآیند پدر در صورتی می‌تواند به کار خود پایان دهد که دست کم، یک فرزند، یکی از صفحه‌های خود را تغییر داده باشد.
۴) هر سه پاسخ درست می‌باشد.

۱۲- در یک سیستم صفحه‌بندی برحسب تقاضا، اگر احتمال نقص صفحه برابر p باشد و زمان انتقال یک صفحه بین حافظه جانبی و حافظه اصلی برابر با d باشد و به طور میانگین نیمی از صفحات در حافظه اصلی تغییر پیدا کرده باشند. اگر از یک حافظه جانبی با سرعت ۲ برابر استفاده شود، آنگاه متوسط زمان دسترسی موثر به حافظه، چقدر کاهش خواهد یافت؟ **(مهندسی کامپیوتر – دولتی۹۳)**

۱) $\frac{1}{2}pd$ ۲) $\frac{2}{3}pd$ ۳) $\frac{3}{4}pd$ ۴) pd

---

۱۳- اگر یک پردازه با دستور ( )fork پردازه جدیدی را ایجاد نماید، کدام یک از داده‌های زیر بین پدر و فرزند به اشتراک گذاشته نمی‌شود؟ **(مهندسی IT- دولتی۹۳)**

۱) Process id ۲) Heap ۳) Stack ۴) Code

khalilifar.ir

۱۴- سیستم عامل می‌تواند بر کوبیدگی (Thrashing) غلبه نماید اگر .....

(مهندسی IT- دولتی۹۳)

۱) اندازه صفحه افزایش یابد.
۲) درجه چند برنامگی را افزایش دهد.
۳) سرعت ورودی و خروجی افزایش یابد.
۴) تخصیص حافظه به پردازه‌ها با توجه به اندازه پنجره‌کاری آنها تنظیم گردد.

۱۵- چگونه سیستم عامل می‌تواند بر کوبیدگی (thrashing) غلبه کند؟ (مهندسی کامپیوتر-دولتی۹۴)

۱) اندازه صفحه افزایش یابد.
۲) درجه چند برنامگی افزایش یابد.
۳) سرعت ورودی و خروجی افزایش یابد.
۴) تخصیص حافظه به پردازه‌ها با توجه به اندازه پنجره کاری آنها تنظیم گردد.

۱۶- در رشته مرجع صفحه زیر اگر از الگوریتم LRU(Least Recently Used) برای جایگزینی صفحات استفاده شود چند خطای صفحه رخ می‌دهد؟ (مهندسی IT - دولتی۹۴)

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

۱) 14 ۲) 13 ۳) 12 ۴) 11

۱۷- فرض کنید در یک سیستم حافظه، زمان دسترسی به کش (حافظه نهان) 10 نانو ثانیه است و زمان دسترسی به حافظه اصلی 200 نانو ثانیه است. زمان دسترسی موثر در این سیستم، 10 درصد بیشتر از زمان دسترسی به کش است. نرخ اصابت (Hit Ratio) در این سیستم چقدر است؟

(مهندسی IT -دولتی۹۴)

۱) $\frac{200}{210}$ ۲) $\frac{199}{200}$ ۳) $\frac{190}{200}$ ۴) $\frac{110}{200}$

---

khalilifar.ir

پاسخ تست‌های فصل پنجم: مدیریت حافظه مجازی

۱- گزینه (۴) صحیح است.

اصولاً سیستم عامل و بخش‌های مختلف آن از جمله زمان‌بند، از محتوای برنامه‌های کاربر مطلع نیست. به بیان دیگر، زمان‌بند، برنامه‌های کاربر را به صورت یک Black Box (جعبه سیاه) می‌بیند و از محتوای آن اطلاعی ندارد. بنابراین گزینه‌های اول و دوم نادرست است.

سیستم عامل Unix برای ایجاد یک فرآیند جدید از فراخوان سیستمی Fork استفاده می‌کند. در این سیستم عامل جهت پیاده‌سازی مفهوم حافظه مجازی از تکنیک Copy-On-Write استفاده می‌گردد. Copy-On-Write یکی از فیلدهای جدول صفحه است و هنگامی که بیش از یک فرآیند در یک صفحه باشد، این فیلد براساس تعداد فرآیندهای موجود در یک صفحه مقدار می‌گیرد.

در سیستم عامل Unix، هرگاه فرآیندی (فرآیند پدر) فراخوانی سیستمی Fork را انجام دهد فرآیند جدیدی (فرآیند فرزند) ایجاد می‌شود که کدها و داده‌های آن عیناً همان کدها و داده‌های فرآیند پدر است. در واقع جهت صرفه‌جویی در حافظه، فرآیندهای پدر و فرزند در ابتدای کار از صفحات کد و داده به صورت مشترک استفاده می‌کنند، البته تا زمانی که فقط عمل خواندن (Read) بین فرآیندها مدنظر باشد، این صفحات به صورت مشترک استفاده می‌گردد.

اما بر طبق تکنیک Copy-On-Write هرگاه یکی از دو فرآیند پدر یا فرزند بخواهد صفحه‌ای از صفحات مشترک را تغییر دهد (مثلاً چیزی بنویسد) یک کپی جداگانه از آن فرآیند ساخته می‌شود و فرآیند از آن به بعد از آن استفاده می‌کند (به جای صفحه مشترک) و فرآیندهای دیگر از صفحات اصلی (مشترک) استفاده می‌کنند. بدین ترتیب محرمانگی داده‌ها حفظ شده و تغییرات صورت گرفته توسط یک فرآیند بر روی سایر فرآیندها اثر نخواهد داشت.

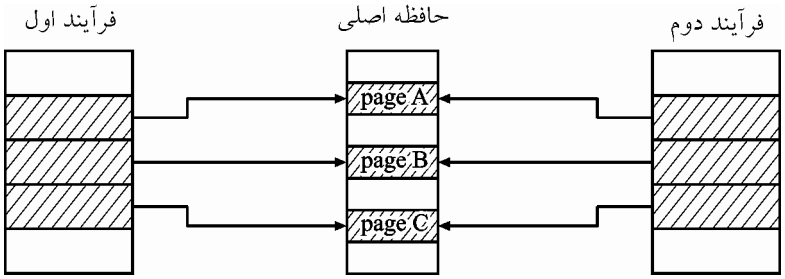
در سیستم عامل Unix یک فرآیند فرزند پس از ایجاد توسط فراخوانی Fork، می‌تواند بلافاصله از فراخوانی سیستمی ()exec استفاده کند و کل فضای حافظه‌ی خود را جایگزین کند و از ادامه شراکت با پدر صرفه نظر کند. بنابراین، با این کار، حافظه‌ی جداگانه‌ای برای فرآیند فرزند ایجاد می‌شود. با توجه به این مطلب اگر بعد از اجرای Fork و ایجاد فرآیند فرزند، ابتدا فرآیند پدر اجرا گردد، ممکن است فرآیند پدر بخواهد بطور خصوصی در یکی از صفحات چیزی بنویسد و باعث ایجاد یک کپی از صفحه بر اساس تکنیک Copy-On-Write شود. حال اگر در ادامه فرآیند فرزند اجرا گردد و در همان ابتدا از فراخوان سیستمی ()exec استفاده کند و راه خود را از پدر جدا کند و شراکت را برهم زند، صفحاتی که پدر به دلیل تغییرات خود ایجاد کرده بود، سربار به حساب می‌آیند و کار بیهوده تلقی می‌گردد، مانند پدری که پس از فرزنددار شدن برای صرفه‌جویی در هزینه‌ها، از خانه‌ی خود به شکل اشتراکی استفاده می‌کند. اما این پدر بعدها به دلیل کارهای شخصی خود خانه‌ی دیگری را نیز تهیه می‌کند و بعد از تهیه خانه‌ی دوم متوجه می‌شود، که فرزند راه خود را جدا کرده است، و شراکت را بر هم زده است، و فرزند نیز خانه‌ای برای خود تهیه

khalilifar.ir

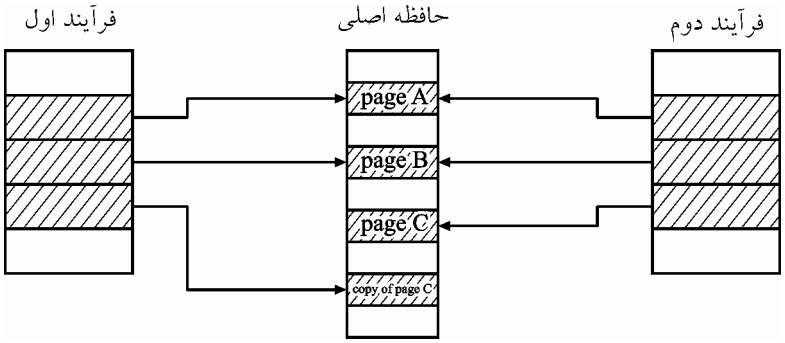
کرده است، حال خانه‌ی دوم پدر برای رسیدگی به امور شخصی بلااستفاده می‌ماند و این سربار است، زیرا دیگر شراکتی در کار نیست، فرزندی نیست، همان خانه‌ی اول برای پدر کافی بود. بهتر بود پدر صبر می‌کرد، تا اول فرزند تصمیم بگیرد، سپس بر اساس تصمیم فرزند، پدر نیز تصمیم خود را می‌گرفت.

حال مجدداً برگردید به وادی کامپیوتر، در صورتی که اگر بعد از اجرای فراخوانی سیستمی Fork، ابتدا فرآیند فرزند فراخوانی و اجرا شود، ممکن است در همان ابتدای اجرایش دستور ()exec را اجرا کند و در نتیجه از آن به بعد، از فضای حافظه‌ی شخصی خود استفاده کند، که در این صورت اگر فرآیند پدر بخواهد چیزی بر روی صفحات مشترک شده بنویسد، دیگر نیازی به کپی‌کردن آن صفحه نخواهد بود و این یعنی حذف سربار و افزایش کارایی.

با توجه به مطالب مطرح شده، گزینه چهارم درست و گزینه سوم نادرست خواهد بود.



قبل از اینکه فرآیند اول صفحه C را تغییر دهد.



بعد از اینکه فرآیند اول صفحه C را تغییر دهد.

۲- گزینه (۴) صحیح است.

به طور کلی میانگین زمان دسترسی سیستم به مقصد نهایی برای هر آدرس سیستم از رابطه زیر محاسبه می‌گردد:

$$T_{Access} = T_{Translation} + T_{Destination}$$

khalilifar.ir

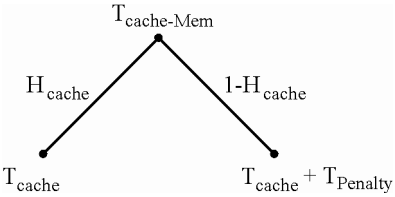
که پارامترهای آن شامل $T_{Translation}$ یعنی **بخش ترجمه** و $T_{Destination}$ یعنی **بخش مقصد** می‌باشد.

$T_{Translation}$: میانگین زمان ترجمه

توجه: مطابق فرض سوال در **بخش ترجمه** یعنی $T_{Translation}$، سه عنصر T_{TLB}، T_{Cache} و T_{Mem} برای ترجمه وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

برای محاسبه $T_{Translation}$ ابتدا میانگین T_{Cache} و T_{Mem} را محاسبه می‌کنیم به شکل $T_{Cache-Mem}$، در نهایت هم، میانگین $T_{Cache-Mem}$ و T_{TLB} را محاسبه می‌کنیم که همان $T_{Translation}$ است. به صورت زیر:

برای بدست آوردن $T_{Cache-Mem}$ درخت زیر را در نظر بگیرید:



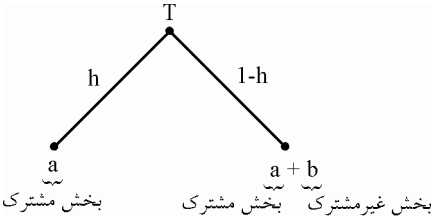
توجه: PAGE FRAME یا در Cache قرار دارد یا ندارد، اگر PAGE FRAME در Cache نباشد، پس باید به Cache آورده شود، که این زمان به Cache آوردن برابر $T_{Penalty}$ است. در واقع سیستم از Cache می‌خواند. بنابراین در درخت فوق به جای استفاده از T_{Mem} از $T_{Penalty}$ استفاده کردیم که شامل رفتن به Memory و آوردن به Cache است.

پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Cache-Mem}$ خواهیم داشت:

$$T_{Cache-Mem} = T_{Cache} + (1-H_{Cache}) \times T_{Penalty}$$

$$T_{Cache-Mem} = 10 + (1-0.9) \times 100 = 20 \text{ ns}$$

توجه: جهت ساده‌سازی رابطه درخت فوق، همواره می‌توان از **اتحاد درخت احتمال** به صورت زیر، استفاده نمود:



$$T = \text{بخش مشترک} + (1-h) \times \text{بخش غیرمشترک}$$

$$T = a + (1-h) \times b$$

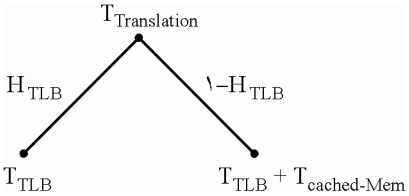
khalilifar.ir

پس از ساده‌سازی درخت مطرح شده براساس **اتحاد درخت احتمال**، رابطه زیر را بـرای محاسـبه $T_{Cache-Mem}$ خواهیم داشت:

$$T_{Cache-Mem} = T_{Cache} + (1-H_{Cache}) \times T_{Penalty}$$

$$T_{Cache-Mem} = 10 + (1-0.9) \times 100 = 20 \text{ ns}$$

در انتها برای بدست آوردن $T_{Translation}$ درخت زیر را در نظر بگیرید:



پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Translation}$ خواهیم داشت:

$$T_{Translation} = T_{TLB} + (1-H_{TLB}) \times T_{Cache-Mem}$$

$$T_{Translation} = 2 + (1-0.98) \times 20 = 2.4 \text{ ns}$$

$T_{Destination}$: میانگین زمان دسترسی به مقصد موردنظر

توجه: مطابق فرض سوال در **بخش مقصد** یعنی $T_{Destination}$، سه عنصر T_{Cache}، T_{Mem} و T_{Disk} برای دسترسی به مقصد مورد نظر وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

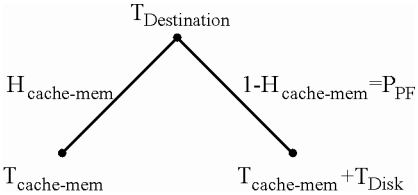
بـرای محاسـبه $T_{Destination}$ ابتـدا میـانگین T_{Cache} و T_{Mem} را محاسـبه مـی‌کنـیم بـه شـکل $T_{Cache-Mem}$، در نهایـت هـم، میـانگین $T_{Cache-Mem}$ و T_{Disk} را محاسـبه مـی‌کنـیم کـه همـان $T_{Destination}$ است. به صورت زیر:

مطابق آنچه پیشتر گفتیم، $T_{Cache-Mem}$ به صورت زیر محاسبه شد:

$$T_{Cache-Mem} = T_{Cache} + (1-H_{Cache}) \times T_{Penalty}$$

$$T_{Cache-Mem} = 10 + (1-0.9) \times 100 = 20 \text{ ns}$$

در انتها برای بدست آوردن $T_{Destination}$ درخت زیر را در نظر بگیرید:



پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Destination}$ خواهیم داشت:

$$T_{Destination} = T_{Cache-Mem} + (1-H_{Cache-Mem}) \times T_{Disk}$$

khalilifar.ir

$T_{Destination} = T_{Cache\text{-}Mem} + (P_{PF}) \times T_{Disk}$

$T_{Destination} = 20 + (2\times10^{-6}) \times (10\times10^{6}) = 20+20 = 40\ ns$

و در نهایت براساس رابطه کل زمان دسترسی سیستم خواهیم داشت:

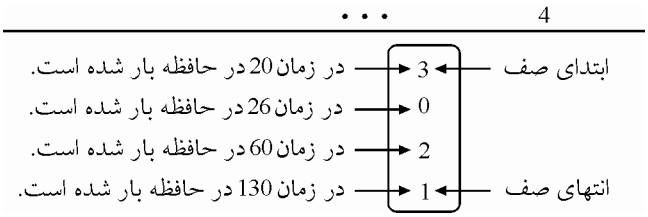
$T_{Access} = T_{Translation} + T_{Destination}$

$T_{Access} = 2.4+40 = 42.4ns \cong 43\ ns$

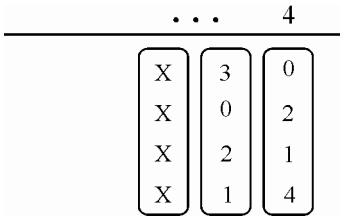
۳- گزینه(۳) صحیح است.

صفحه‌ای که به دلیل خطای نقص صفحه باید خارج گردد، مطابق الگوریتم FIFO براساس خروج به ترتیب ورود می‌باشد. بنابراین در این الگوریتم، زمان بارشدن صفحه در حافظه اهمیت دارد.

الگوی زیر را در نظر بگیرید:



بنابراین صفحه شماره 3 برای خروج انتخاب می‌گردد، زیرا در ابتدای صف قرار دارد، پس از جایگزینی صفحه 4 با صفحه شماره 3، شکل حافظه به صورت زیر خواهد بود:

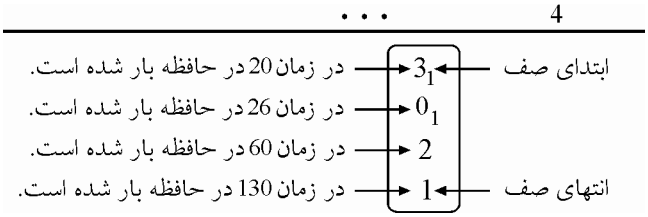


بنابراین گزینه سوم درست است.

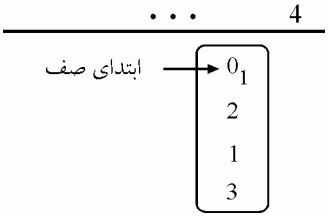
صفحه‌ای که به دلیل خطای نقص صفحه باید خارج گردد، مطابق الگوریتم Clock همانند الگوریتم FIFO براساس خروج به ترتیب ورود می‌باشد. با این تفاوت که صفحه‌ای که برای خروج انتخاب می‌گردد، ابتدا بیت R آن بررسی می‌شود، اگر بیت R آن صفر بود، صفحه کاندید، از حافظه خارج می‌گردد، اما اگر صفحه‌ای که برای خروج انتخاب شده است، بیت R آن برابر یک باشد، ابتدا بیت R این صفحه کاندید برابر صفر شده و سپس به انتهای صف منتقل می‌گردد، در واقع شانس می‌آورد که خارج نمی‌شود چون یک شانس حضور دارد، سپس توسط جلو رفتن اشاره‌گر، نوبت به صفحه بعدی داده می‌شود که بیت R آن برای خروج بررسی می‌گردد و این روال تا پیدا کردن یک صفحه با بیت R برابر با صفر ادامه پیدا می‌کند.

khalilifar.ir

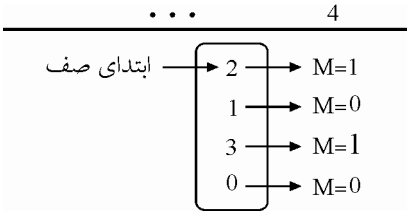
الگوی زیر را در نظر بگیرید:



در صورت سوال مطرح شده است که در حال حاضر اشاره‌گر روی صفحه شماره صفر قرار دارد، یعنی صفحه‌ی شماره 3 یک شانس دارد و به انتهای صف منتقل می‌گردد. حال مطابق فرض مسأله اشاره‌گر به صفحه‌ی شماره صفر در حال اشاره است.



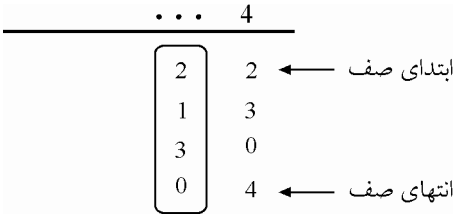
اما صفحه شماره‌ی صفر مقدار بیت R آن برابر یک می‌باشد، بنابراین مقدار آن صفر می‌شود و مجدداً اشاره‌گر یکی رو به جلو می‌رود. بنابراین داریم:



اگرچه برای صفحه شماره 2 بیت R برابر صفر است و طبق الگوریتم ساعت معمولی، این صفحه باید خارج گردد، اما دقت کنید که در صورت سوال، مطرح شده است که «الویت با انتخاب صفحه تغییر نیافته است (M=0)». در واقع با یک الگوریتم ساعت توسعه‌یافته مواجه هستیم که علاوه بر بیت R، بیت M را نیز برای خروج صفحه در نظر می‌گیرد. در واقع در این الگوریتم ساعت توسعه یافته از بین صفحات کاندید برای خروج با بیت R برابر با صفر، آن صفحه‌ای خارج می‌گردد که هم بیت R آن برابر صفر باشد و هم بیت M آن برابر صفر باشد. بنابراین مطابق مطالب بیان شده، صفحه‌ی شماره 1 نسبت به صفحه شماره 2 از اولویت بالاتری برای خروج برخوردار است. بنابراین صفحه شماره 1 به جای صفحه شماره 2 خارج می‌گردد. اگر یک دور می‌زدیم و هیچ

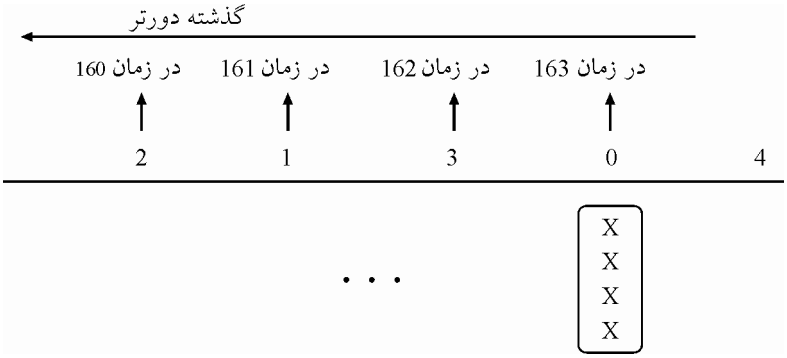
khalilifar.ir

صفحه‌ای هر دو بیت R و M آن هر دو صفر نبود، همان صفحه شماره 2 خارج می‌گردید. پس از جایگزینی صفحه شماره 4 با صفحه شماره 1، شکل حافظه به صورت زیر خواهد بود:



صفحه‌ای که به دلیل خطای نقص صفحه باید خارج گردد، مطابق الگوریتم LRU، صفحه‌ای است که در گذشته‌ی دورتری مورد مراجعه قرار گرفته است. بنابراین در این الگوریتم، زمان آخرین مراجعه به صفحه اهمیت دارد.

الگوی زیر را در نظر بگیرید:



توجه: کاراکتر X در شکل فوق به این معنی است که شماره صفحات 0، 1، 2 و 3 به هر ترتیبی ممکن است وجود داشته باشد، اما به هر ترتیبی که باشند، به هنگام نقص صفحه، مطابق الگوریتم LRU، صفحه‌ای برای جایگزینی انتخاب می‌گردد، که در گذشته‌ی دورتری، مورد استفاده قرار گرفته باشد. بنابراین از آنجایی که صفحه‌ی شماره 2، در گذشته‌ی دورتری مورد استفاده قرار گرفته است، پس صفحه‌ی شماره 2 برای خروج انتخاب می‌گردد.

۴- گزینه (۱) صحیح است.

به طور کلی میانگین زمان دسترسی سیستم به مقصد نهایی برای هر آدرس سیستم از رابطه زیر محاسبه می‌گردد:

$$T_{Access} = T_{Translation} + T_{Destination}$$

که پارامترهای آن شامل $T_{Translation}$ یعنی **بخش ترجمه** و $T_{Destination}$ یعنی **بخش مقصد** می‌باشد.

khalilifar.ir

توجه: در صورت سؤال مطرح شده است که، اگرچه می‌توان میانگین زمان دسترسی به یک آدرس حافظه مجازی را محاسبه نمود، ولی زمان‌های واقعی دسترسی‌ها متعدد هستند و با مقدار میانگین تفاوت دارند، در واقع خواسته سؤال، محاسبه تک به تک زمان دسترسی به یک آدرس حافظه مجازی است و نه میانگین زمان دسترسی به یک آدرس حافظه مجازی.

توجه: مطابق فرض سؤال در **بخش ترجمه** یعنی $T_{Translation}$، دو عنصر T_{TLB} و T_{Mem} برای ترجمه وجود دارد.

توجه: مطابق فرض سؤال در **بخش مقصد** یعنی $T_{Destination}$، دو عنصر T_{Mem} و T_{Disk} وجود دارد.

برخی از انواع زمان‌های دسترسی به یک آدرس حافظه مجازی به صورت زیر است:

حالت اول: صفحه مورد درخواست در حافظه است و آدرس آن در TLB موجود است.

$$T = T_{Translation} + T_{Destination}$$

$$T = T_{TLB} + T_{Mem} = 2ns + 50ns = 52ns$$

حالت دوم: صفحه مورد درخواست در حافظه است، اما آدرس آن در TLB نیست، در واقع تمام جدول صفحه در حافظه قرار دارد، بنابراین برای ترجمه نیز باید سراغ حافظه رفت، زیرا آدرس در جدول صفحه درون حافظه قرار دارد.

$$T = T_{Translation} + T_{Destination}$$

$$T = (T_{TLB}+T_{Mem}) + T_{Mem} = (2ns+50ns) + 50ns = 102ns$$

حالت سوم: صفحه مورد درخواست در حافظه نیست، یعنی نقص صفحه رخ می‌دهد، بنابراین صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه منتقل شود، اما قاب خالی در حافظه موجود نیست، بنابراین یک صفحه باید از حافظه خارج گردد تا فضای کافی برای ورود صفحه مورد درخواست فراهم گردد. ولی صفحه‌ای که باید از حافظه خارج گردد، **تمیز** است، یعنی کپی صفحه موجود در حافظه با کپی صفحه موجود در دیسک یکی است، یعنی صفحه تغییر نکرده است. بنابراین نیاز به دوباره نویسی ندارد. یعنی صفحه جدید بر روی صفحه کاندید برای خروج جایگزین می‌گردد، بدون آنکه نیاز باشد صفحه کاندید برای خروج، اطلاعاتش بر روی دیسک ذخیره گردد.

$$T = T_{Translation} + T_{Destination}$$

$$T = (T_{TLB} + T_{Mem}) + (T_{Mem} + T_{Disk}) = (2ns + 50ns) + (50ns + 3ms) \cong 3ms$$

توجه: مطابق فرض مسأله از اعداد بسیار کوچک در مقابل اعداد بسیار بزرگ صرف‌نظر شده است.

حالت چهارم: صفحه مورد درخواست در حافظه نیست، یعنی نقص صفحه رخ می‌دهد، بنابراین صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه منتقل شود، اما قاب خالی در حافظه هم موجود است. بنابراین صفحه مورد درخواست در قاب خالی قرار می‌گیرد.

khalilifar.ir

$T = T_{Translation} + T_{Destination}$

$T = (T_{TLB}+T_{Mem}) + (T_{Mem}+T_{Disk}) = (2ns+50ns) + (50ns+3ms) \cong 3ms$

حالت پنجم: صفحه مورد درخواست در حافظه نیست، یعنی نقص صفحه رخ می‌دهد، بنابراین صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه منتقل شود، اما قاب خالی در حافظه موجود نیست، بنابراین یک صفحه باید از حافظه خارج گردد تا فضای کافی برای ورود صفحه مورد درخواست فراهم گردد. ولی صفحه‌ای که باید از حافظه خارج گردد، **کثیف** است، یعنی کپی صفحه موجود در حافظه با کپی صفحه موجود در دیسک یکی نیست، یعنی صفحه تغییر کرده است. بنابراین نیاز به دوباره نویسی دارد. یعنی قبل از آنکه صفحه جدید بر روی صفحه کاندید برای خروج جایگزین گردد، ابتدا اطلاعات صفحه کاندید برای خروج بر روی دیسک ذخیره می‌گردد که یک $T_{RW} = T_{Disk}$ زمان نیاز دارد، سپس صفحه جدید بر روی صفحه کاندید برای خروج جایگزین می‌گردد، که مجدداً یک T_{Disk} زمان نیاز دارد.

$T = T_{Translation} + T_{Destination}$

$T = (T_{TLB}+T_{Mem}) + (T_{Mem}+T_{RW}+T_{Disk}) = (2ns+50ns) + (50ns+ 3ms+3ms) \cong 6ms$

۵- گزینه (۱) صحیح است.

در کامپیوترها فقط صرف اجرای برنامه‌ها در دستور کار قرار ندارد، بلکه اجرای امن برنامه‌ها در دستور کار قرار دارد، امن بودن به این معنی است که اجرای برنامه‌ها باعث مختل شدن بخشی از سیستم کامپیوتری یا تمام سیستم نگردد. بهتر بود در صورت سؤال جمله اول به شکل زیر مطرح می‌شد:

راه‌حل‌های سخت‌افزاری متعددی به همراه سیستم عامل به اجرای **امن** برنامه‌ها کمک می‌کند. اما اینکه وجود سخت‌افزار برای اجرا لازم است، کاملاً بدیهی است. ولی آیا سخت‌افزار برای برقراری امنیت هم کاربرد دارد؟ مگر سیستم عامل به تنهایی و بدون پشتیبانی سخت‌افزار قادر به برقراری امنیت نیست؟ بدیهی است که خیر، زیرا وقتی سیستم عامل پردازنده را به یک برنامه کاربر می‌سپارد تا آن را اجرا کند، برای مدتی هیچ برنامه‌ای جز برنامه مورد نظر اجرا نمی‌شود. بنابراین سیستم عامل که خودش یک برنامه است و تا اجرا نشود کاری نمی‌تواند بکند، چگونه ممکن است در لحظه وقوع نقص حفاظتی از این امر مطلع شود و برای جلوگیری از آن کاری بکند، مگر آنکه سخت‌افزار حفاظت این نقص‌ها را کشف کرده و سیستم عامل را وارد صحنه کند.

نتیجه اینکه سخت‌افزار هم برای اجرای برنامه‌ها، به برنامه‌ها کمک می‌کند و هم برای برقراری امنیت اجرای برنامه‌ها، به برنامه‌ها کمک می‌کند، در این صورت یک برنامه، فقط اجرا نمی‌شود که هر کاری که دلش می‌خواهد انجام دهد و به تمام بخش‌های کامپیوتر دسترسی داشته باشد، بلکه با این تفاسیر، برنامه می‌تواند اجرا شود، اما باید اجرای **امن** شود، یعنی در یک چارچوب از قبل مشخص شده و مبتنی بر قوانین سیستم اجرا شود، در واقع سخت‌افزار کمک می‌کند تا برنامه‌ها

khalilifar.ir

اجرا شوند، اما هرجا که لازم باشد و هرجا که برنامه پا را از گلیم بیشتر دراز کند، آنگاه سخت‌افزار جلوی اجرای برنامه خاطی را می‌گیرد و پردازنده را هم از او می‌گیرد و سیستم عامل را وارد صحنه می‌کند تا به امور برنامه خاطی رسیدگی کند.

انسان حرف‌های خود را توسط برنامه و به واسطه زبان‌های برنامه‌نویسی، برای اجرا به کامپیوتر می‌گوید. اما این کامپیوتر ننشسته است که هرچه برنامه بگوید، چشم بسته، اجرا کند. عملیات کامپیوتر حساب دارد، کتاب دارد، در واقع هر سیستمی نیاز به رسیدگی و کنترل دارد، وگرنه هرج و مرج می‌شود و کارکرد سیستم مختل می‌گردد.

در سیستم‌های کامپیوتری، مسئولیت خواندن و اجرای دستورات برنامه کاربر را مجموعه پردازنده مرکزی برعهده گرفته است. حال این پردازنده خط به خط برنامه‌های کاربر را می‌خواند، اگر حرف خوب بود اجرا می‌کند و اگر حرف بد بود اجرا نمی‌کند. حرف خوب از نگاه پردازنده، کار مجاز (کارغیرممتاز) است و حرف بد از نگاه پردازنده کار غیرمجاز (کار ممتاز) است. اگر کار غیرمجاز دید، بزرگتر خود، سیستم عامل را وارد صحنه می‌کند.

توجه: از آنجا که در هنگام اجرای برنامه کاربر، سیستم عامل حضور ندارد، بنابراین کشف و تشخیص هرگونه کار غیرمجاز (انجام کار ممتاز) به واسطه برنامه کاربر، توسط پردازنده و سخت‌افزار انجام می‌شود.

توجه: به طور کلی، تمامی کارهای حمایتی که در هنگام اجرای برنامه‌های کاربر انجام می‌گردد، همه سخت‌افزاری هستند، چون در لحظه اجرای برنامه کاربر، سیستم عامل اصلاً حضور ندارد.

توجه: عملیات ورودی و خروجی، عمل تقسیم بر صفر، تغییر تایمر سیستم، تغییر ثبات‌های حفاظتی، از کار انداختن وقفه، دسترسی به حافظه سیستم عامل، دسترسی بدون اجازه به حافظه فرآیندهای دیگر نمونه‌هایی ازکارهای ممتاز محسوب می‌گردند. در یک بیان کلی‌تر، هرکاری که استفاده نادرست ازآن باعث مختل شدن بخشی از سیستم کامپیوتری یا کل سیستم کامپیوتری گردد، کار ممتاز به حساب می‌آید.

توجه: انجام کار ممتاز برای یک فرآیند کاربر، کار غیرمجاز است و فقط انجام کار غیرممتاز است که برای یک فرآیند کاربر، مجاز محسوب می‌شود.

توجه: هرگاه یک فرآیند کاربر قصد اجرای یک کار ممتاز را توسط پردازنده داشته باشد، پردازنده فوراً توسط یک وقفه، سیستم عامل را باخبر می‌کند و کنترل به سیستم عامل واگذار می‌گردد تا سیستم عامل در پاسخ به آن وقفه یک روال پاسخ به وقفه را اجرا کند.

توجه: دقت کنید که در هر لحظه فقط یک برنامه قادر به اجرا توسط پردازنده است، در واقع در لحظه‌ای که برنامه کاربر در حال اجرا است و تصمیم به انجام یک کار ممتاز می‌گیرد، سیستم عامل اصلاً در حال اجرا نیست که بخواهد جلوی این کار غیرمجاز فرآیند در حال اجرا را بگیرد، اما سیستم عامل این مسأله را قبلاً با پردازنده و سخت‌افزار هماهنگ کرده است، پردازنده و سخت‌افزار در غیاب سیستم عامل، به محض مشاهده یک کار غیرمجاز از فرآیند کاربر در حال

khalilifar.ir

اجرا، فوراً توسط صدور یک وقفه، سیستم عامل را وارد صحنه می‌کند. در واقع به واسطه وقفه، پردازنده از فرآیند خاطی گرفته شده و در اختیار سیستم عامل قرار داده می‌شود تا سیستم عامل بتواند به امور فرآیند خاطی رسیدگی کند.

توجه: در سطح پردازنده دو نوع کار انجام می‌گردد، (۱) کارهای ممتاز، (۲) کارهای غیر ممتاز، کارهای ممتاز در مود هسته پردازنده و کارهای غیرممتاز در مود کاربر پردازنده اجازه‌ی اجرا دارند. بنابراین هیچگاه یک کار ممتاز اجازه اجرا در مود کاربر را نخواهد داشت.

کارهای حمایتی سخت‌افزار در هنگام اجرای برنامه کاربر (حینا اجرا) عبارتند از:

- در هنگام اجرای برنامه کاربر از TBL استفاده می‌گردد، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش MMU) حمایت و کنترل می‌کند. بنابراین کنترل جدول TLB در هنگام اجرای یک برنامه کاربر بر عهده‌ی سخت‌افزار است.
- در هنگام اجرای برنامه کاربر نیاز به نگاشت آدرس منطقی به فیزیکی است، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش MMU) حمایت و کنترل می‌کند. بنابراین تبدیل آدرس منطقی به فیزیکی در هنگام اجرای یک برنامه کاربر، بر عهده‌ی سخت‌افزار است.
- در هنگام اجرای برنامه کاربر ممکن است، نیاز به ثبت مقدار 1 در بیت Modified یک صفحه باشد، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش MMU) حمایت و کنترل می‌کند. بنابراین ثبت تغییر یافتن (Modify) یک صفحه در هنگام اجرای یک برنامه کاربر بر عهده سخت‌افزار است.
- در هنگام اجرای برنامه کاربر ممکن است یک کار غیرمجاز از سوی برنامه انجام شود، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (مود حفاظت شده CPU) حمایت و کنترل می‌کند. بنابراین کشف و تشخیص کار غیرمجاز (دستورالعمل غیرمجاز) یک برنامه کاربر در حال اجرا، برعهده‌ی سخت‌افزار است.
- در صورت وجود برش زمانی در سیستم، در هنگام اجرای برنامه کاربر، لازم است پایان برش زمانی به برنامه اعلام شود، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار حمایت و کنترل می‌کند (توسط Timer). بنابراین اعلام پایان برش زمانی در هنگام اجرای یک برنامه کاربر، بر عهده سخت‌افزار است.
- در هنگام اجرای برنامه کاربر از حافظه پنهان (Cache) استفاده می‌گردد، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش CCU) حمایت و کنترل می‌کند. بنابراین کنترل حافظه‌ی پنهان (Cache) در هنگام اجرای یک برنامه کاربر، بر عهده‌ی سخت‌افزار است.
- در هنگام اجرای برنامه‌ی کاربر ممکن است نقص صفحه رخ دهد، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش MMU) حمایت و کنترل می‌کند.

khalilifar.ir

بنابراین کشف و تشخیص نقص صفحه در هنگام اجرای یک برنامه کاربر برعهده‌ی سخت‌افزار است.

- در هنگام اجرای برنامه کاربر ممکن است، نیاز به ثبت مقدار 1 در بیت دسترسی (Referenced Access) یک صفحه باشد، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش MMU) حمایت و کنترل می‌کند، بنابراین ثبت دسترسی (Access) یک صفحه در هنگام اجرای یک برنامه‌ی کاربر، برعهده‌ی سخت‌افزار است.
- در هنگام اجرای برنامه کاربر ممکن است، سرریزی (Over Flow) در محاسبات پردازنده رخ دهد، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار حمایت و کنترل می‌کند، بنابراین کشف و تشخیص سرریزی (Over Flow) در هنگام اجرای یک برنامه‌ی کاربر، برعهده‌ی سخت‌افزار است.

توجه: پس از این کشف و تشخیص سرریزی (Over Flow) توسط سخت‌افزار، سخت‌افزار توسط یک وقفه سیستم عامل را باخبر و وارد صحنه می‌کند، تا به این مسأله توسط روال پاسخ وقفه، پاسخ دهد. در واقع برنامه کاربر دچار یک کار غیرمجاز شده است، که می‌بایست سخت‌افزار توسط یک وقفه و گرفتن پردازنده از برنامه کاربر، سیستم عامل را با خبر و وارد صحنه کند، تا به امور برنامه خاطی رسیدگی کند.

- در هنگام اجرای برنامه کاربر نیاز به نگاشت آدرس منطقی (آدرس قطعه‌بندی شده) به فیزیکی (آدرس حافظه‌ی اصلی) است، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش MMU) حمایت و کنترل می‌کند. بنابراین تبدیل آدرس منطقی به فیزیکی در هنگام اجرای یک برنامه کاربر، بر عهده‌ی سخت‌افزار است.
- در هنگام اجرای برنامه کاربر ممکن است یک کار تعریف نشده و غیرمجاز از سوی برنامه کاربر انجام شود، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (مود حفاظت شده CPU) حمایت و کنترل می‌کند. بنابراین کشف، تشخیص و جلوگیری از اجرای کار و دستورالعمل تعریف نشده و غیرمجاز برنامه‌ی کاربر در حال اجرا، بر عهده‌ی سخت‌افزار است.
- در هنگام اجرای برنامه ممکن است، برنامه بخواهد پا را از گلیم خود فراتر قرار دهد و به فضای حافظه‌ی سایر فرآیندها سَرَک بکشد، که یک کار غیرمجاز است، که در آن لحظه سیستم عامل حضور ندارد، اما سخت‌افزار (بخش MMU) حمایت و کنترل می‌کند. بنابراین حفاظت از فضای حافظه‌ی سایر فرآیندها در قبال درخواست دسترسی غیرمجاز یک برنامه کاربر در حال اجرا، بر عهده‌ی سخت‌افزار است.

تا به اینجا به این نتیجه رسیدیم که سیستم عامل در لحظه‌ی اجرای یک برنامه کاربر حمایتی را در قبال برنامه کاربر در حال اجرا انجام نمی‌دهد، مگر سخت‌افزار توسط تولید وقفه، درخواست ورود سیستم عامل را داشته باشد که در این صورت برنامه کاربر در حال اجرا پردازنده را واگذار می‌کند

khalilifar.ir

و سیستم عامل وارد صحنه می‌گردد. بنابراین مسئولیت **کارهای حمایتی**، در قبال برنامه در حال اجرا (حینا اجرا) را، **سخت افزار** بر عهده می‌گیرد. اما مسئولیت **کارهای حمایتی**، قبل (پیشا اجرا) و بعد از اجرای برنامه کاربر (پسا اجرا) را، **سیستم عامل** بر عهده می‌گیرد.

کارهای حمایتی سیستم عامل قبل و بعد از اجرای برنامه کاربر (پیشا اجرا و پسا اجرا) عبارتند از:

- قبل از اجرای برنامه کاربر، ثبات‌های مدیریت حافظه (بخش MMU) و تمام ثبات‌های پردازنده بر اساس PCB فرآیند، بار می‌شوند، در این لحظه سیستم عامل حضور دارد و حمایت و کنترل می‌کند. بنابراین بار کردن ثبات‌های مدیریت حافظه و تمام ثبات‌های پردازنده قبل از اجرای برنامه کاربر، برعهده‌ی سیستم عامل است. بنابراین گزینه‌ی دوم نادرست است.

توجه: بار کردن ثبات‌های مدیریت حافظه و پردازنده در بخش Dispatcher سیستم عامل صورت می‌گیرد.

- بعد از اجرای برنامه کاربر و متوقف شدن برنامه کاربر به دلیل وقفه‌های نرم‌افزاری یا سخت‌افزاری، مانند پایان برش زمانی، پردازنده از برنامه کاربر گرفته می‌شود و به سیستم عامل واگذار می‌شود تا سیستم عامل پاسخ مناسب را ارائه کند. در این حالت سیستم عامل بر اساس شرایط، برنامه کاربر را در **حالت مسدود** (مثلاً به دلیل عملیات ورودی و خروجی) و یا در **حالت آماده** (به دلیل پایان برش زمانی) قرار می‌دهد. همچنین سیستم عامل پس از اتمام عملیات ورودی و خروجی برنامه کاربر، برنامه را در صف آماده قرار می‌دهد و یا قبل از اجرای برنامه، سیستم عامل توسط زمان‌بند کوتاه مدت یک برنامه را از صف آماده برای در اختیار گرفتن پردازنده و عملیات اجرا، انتخاب می‌کند. بنابراین گزینه سوم نادرست است.
- بعد از اجرای برنامه و توقف برنامه کاربر به دلیل درخواست خروج توسط خود برنامه (اجرای یک فراخوان سیستمی)، توسط صدور یک وقفه نرم‌افزاری به سیستم عامل، سیستم عامل فرآیند مورد نظر را از سیستم خارج می‌کند.

توجه: کشف و تشخیص این وقفه توسط سخت‌افزار است اما خروج فرآیند توسط سیستم عامل انجام می‌گردد. بنابراین گزینه سوم نادرست است.

- بعد از اجرای برنامه و توقف برنامه کاربر به دلیل نقص صفحه (Page Fault)، سیستم عامل عملیات جستجوی صفحه مورد نظر را بر روی دیسک آغاز می‌کند و سپس صفحه یافت شده را در حافظه قرار می‌دهد.

khalilifar.ir

توجه: کشف و تشخیص نقص صفحه توسط سخت‌افزار است، اما جستجوی صفحه مورد نظر بر روی دیسک و انتقال از دیسک به حافظه توسط سیستم عامل انجام می‌گردد. بنابراین گزینه چهارم نادرست است.

توجه: MMU سرواژه عبارت Memory Mangement Unit و به معنی واحد مدیریت حافظه است.

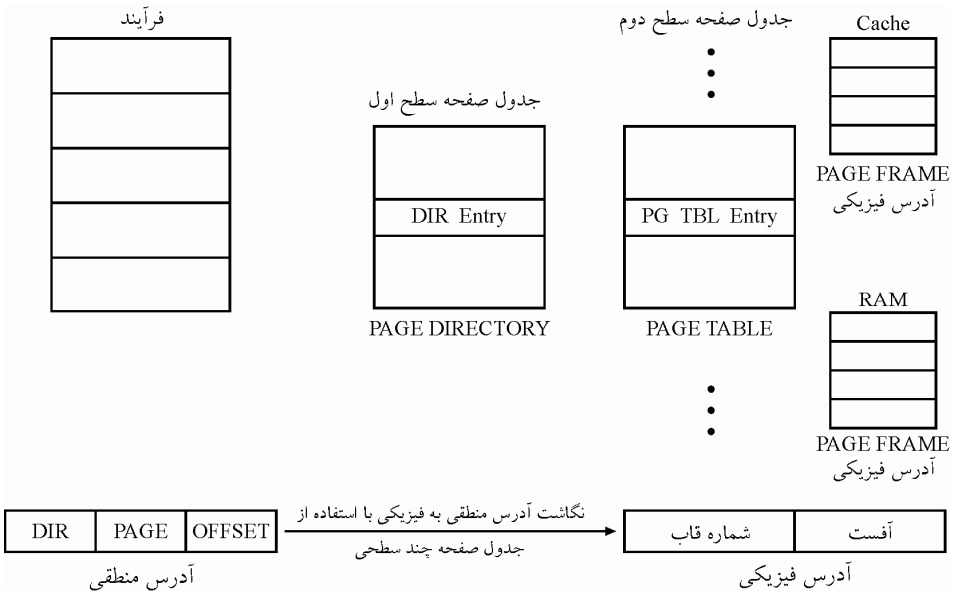
توجه: CCU سرواژه عبارت Cache Control Unit و به معنی واحد کنترل حافظه نهان است.

۶- گزینه (۴) صحیح است.

آدرس منطقی به صورت زیر است:



مطابق فرض مسأله، برای نگاشت آدرس منطقی به فیزیکی از جدول صفحه چند سطحی (دو سطحی) استفاده شده است.



به طور کلی میانگین زمان دسترسی سیستم به مقصد نهایی برای هر آدرس سیستم از رابطه زیر محاسبه می‌گردد:

$$T_{Access} = T_{Translation} + T_{Destination}$$

khalilifar.ir

که پارامترهای آن شامل $T_{Translation}$ یعنی **بخش ترجمه** و $T_{Destination}$ یعنی **بخش مقصد** می‌باشد.

توجه: در صورت سؤال مطرح شده است که، مقدار زمان‌های ممکن برای دسترسی به یک مکان حافظه که با آدرس منطقی مشخص شده است، چقدر است؟ در واقع خواسته سؤال، محاسبه تک به تک زمان دسترسی به یک آدرس حافظه منطقی (مجازی) است و نه میانگین زمان دسترسی به یک آدرس حافظه منطقی.

۱- اگر تعدادی از درایه‌های **مرتبط** دو جدول PAGE DIRECTORY و PAGE TABLE داخل TLB باشد.

توجه: در سیستم جداول صفحه چندسطحی درایه‌های موجودی که در TLB قرار دارند، شامل اطلاعات مرتبط همه سطوح است، نمی‌شود سطحی باشد و سطحی نباشد، در سؤال مذکور یا تعدادی از درایه‌های مرتبط دو جدول PAGE DIRECTORY و PAGE TABLE به طور همزمان داخل TLB هستند و یا به طور همزمان داخل TLB نیستند.

الف) مقصد یعنی PAGE FRAME در Cache باشد:

$$T=T_{Translation}+T_{Destination}$$

$$T=T_{TLB}+T_{Cache}=2+10=12ns$$

ب) مقصد یعنی PAGE FRAME در Cache نباشد:

پس باید به Cache آورده شود، که این زمان به Cache آوردن برابر $T_{Cache\text{-}Miss\text{-}Penalty}$ است.

$$T=T_{Translation}+T_{Destination}$$

$$T=T_{TLB}+(T_{Cache}+T_{Cache\text{-}Miss\text{-}Penalty})=2+(10+100)=112ns$$

۲- اگر هیچ یک از درایه‌های **مرتبط** دو جدول PAGE DIRECTORY و PAGE TABLE داخل TLB نباشد.

الف) جداول PAGE DIRECTORY و PAGE TABLE و مقصد یعنی PAGE FRAME در Cache باشند:

$$T=T_{Translation}+T_{Destination}$$

$$T=(T_{TLB}+T_{Cache\text{-}DIR}+T_{Cache\text{-}PAGE\text{-}TABLE})+T_{Cache\text{-}FRAME}$$

$$T=(2+10+10)+10=32ns$$

توجه: حال اگر در ادامه هر یک از جداول PAGE DIRECTORY و PAGE TABLE و یا مقصد PAGE FRAME در Cache نباشند، باید به Cache آورده شوند، که این زمان به Cache آوردن برابر $T_{Cache\text{-}Miss\text{-}Penalty}$ است.

ب) یک مورد از سه مورد فوق در cache نباشد:

$$T=32+1\times T_{Cache\text{-}Miss\text{-}Penalty}=32+100=132ns$$

ج) دو مورد از سه مورد فوق در Cache نباشد:

$$T=32+2\times T_{Cache\text{-}Miss\text{-}Penalty}=32+2\times100=232ns$$

khalilifar.ir

د) سه مورد از سه مورد فوق در Cache نباشد:

$T=32+3\times T_{Cache\text{-}Miss\text{-}Penalty}=32+3\times100=332ns$

۳- اگر نقص صفحه رخ دهد و مقصد یعنی PAGE FRAME در دیسک باشد.

توجه: مطابق گزینه‌ها از اعداد بسیار کوچک در مقابل اعداد بسیار بزرگ صرفه نظر شده است.

الف) صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه و در نهایت به Cache منتقل شود، اما قاب خالی در حافظه موجود نیست، بنابراین یک صفحه باید از حافظه خارج گردد، تا فضای کافی برای ورود صفحه مورد درخواست فراهم گردد. ولی صفحه‌ای که باید از حافظه خارج گردد، **تمیز** است، یعنی کپی صفحه موجود در حافظه با کپی موجود در دیسک یکی است، یعنی صفحه تغییر نکرده است، بنابراین نیاز به دوباره نویسی ندارد. یعنی صفحه جدید بر روی صفحه کاندید برای خروج، جایگزین می‌گردد، بدون آنکه صفحه کاندید برای خروج، اطلاعاتش بر روی دیسک ذخیره گردد.

$T=T_{Translation}+T_{Destination}\cong T_{Disk}=5ms$

ب) صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه و در نهایت به Cache منتقل شود، اما قاب خالی در حافظه موجود است. بنابراین صفحه مورد درخواست در قاب خالی قرار می‌گیرد.

$T=T_{Translation}+T_{Destination}\cong T_{Disk}=5ms$

ج) صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه و در نهایت به Cache منتقل شود، اما قاب خالی در حافظه موجود نیست، بنابراین یک صفحه باید از حافظه خارج گردد تا فضای کافی برای ورود صفحه مورد درخواست فراهم گردد. ولی صفحه‌ای که باید از حافظه خارج گردد، **کثیف** است، یعنی کپی صفحه‌ی موجود در حافظه با کپی صفحه موجود در دیسک یکی نیست، یعنی صفحه تغییر کرده است، بنابراین نیاز به دوباره‌نویسی دارد. یعنی قبل از آنکه صفحه جدید بر روی صفحه‌ی کاندید برای خروج جایگزین گردد، ابتدا اطلاعات صفحه کاندید برای خروج بر روی دیسک ذخیره می‌گردد که یک $T_{RW}=T_{Disk}$ زمان نیاز دارد، سپس صفحه جدید بر روی صفحه کاندید برای خروج، جایگزین می‌گردد، که مجددا یک T_{Disk} زمان نیاز دارد.

$T=T_{Translation}+T_{Destination}\cong T_{RW}+T_{Disk}=10ms$

۷- گزینه(۴) صحیح است.

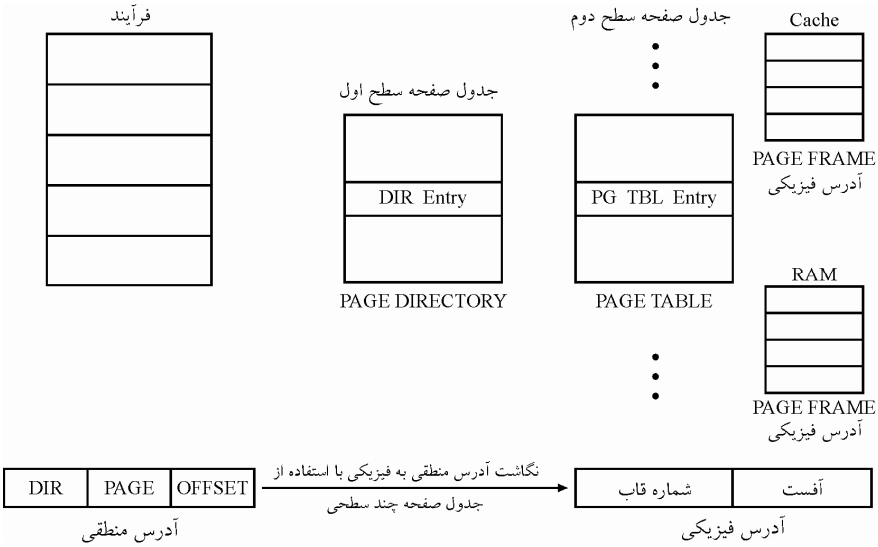
آدرس منطقی به صورت زیر است:



آدرس منطقی

khalilifar.ir

مطابق فرض مساله، برای نگاشت آدرس منطقی به فیزیکی از جداول صفحه چند سطحی (دوسطحی) استفاده شده است.



به طور کلی میانگین زمان دسترسی سیستم به مقصد نهایی برای هر آدرس سیستم از رابطه زیر محاسبه می‌گردد:

$$T_{Access} = T_{Translation} + T_{Destination}$$

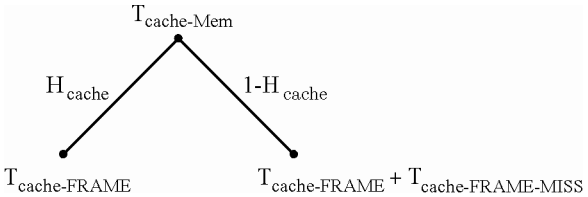
که پارامترهای آن شامل $T_{Translation}$ یعنی **بخش ترجمه** و $T_{Destination}$ یعنی **بخش مقصد** می‌باشد.

$\mathbf{T_{Translation}}$: میانگین زمان ترجمه

توجه: مطابق فرض سوال در **بخش ترجمه** یعنی $T_{Translation}$، سه عنصر T_{TLB}، T_{Cache} و T_{Mem} برای ترجمه وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

برای محاسبه $T_{Translation}$ ابتدا میانگین T_{Mem} و T_{Cache} را محاسبه می‌کنیم به شکل $T_{Cache-Mem}$، در نهایت هم، میانگین $T_{Cache-Mem}$ و T_{TLB} را محاسبه می‌کنیم که همان $T_{Translation}$ است. به صورت زیر:

برای بدست آوردن $T_{Cache-Mem}$ درخت زیر را در نظر بگیرید:



khalilifar.ir

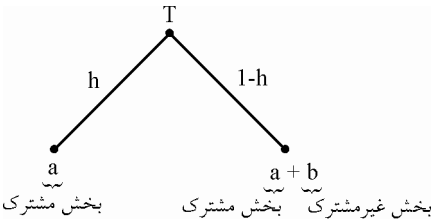
توجه: PAGE FRAME یا در Cache قرار دارد یا ندارد، اگر PAGE FRAME در Cache نباشد، پس باید به Cache آورده شود، که این زمان به Cache آوردن برابر $T_{Cache\text{-}FRAME\text{-}MISS}$ است. در واقع سیستم از Cache می‌خواند. بنابراین در درخت فوق به جای استفاده از T_{Mem} از $T_{Cache\text{-}FRAME\text{-}MISS}$ استفاده کردیم که شامل رفتن به Memory و آوردن به Cache است.

پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Cache\text{-}Mem}$ خواهیم داشت.

$$T_{Cache\text{-}mem} = T_{Cache\text{-}FRAME} + (1- H_{Cache}) \times T_{Cache\text{-}FRAME\text{-}MISS}$$

$$T_{Cache\text{-}mem} = 10+(1\text{-}0.9)\times 100=20\ ns$$

توجه: جهت ساده‌سازی رابطه درخت فوق، همواره می‌توان از **اتحاد درخت احتمال** به صورت زیر، استفاده نمود:



$$T = \text{بخش مشترک} + (1-h)\times \text{بخش غیرمشترک}$$

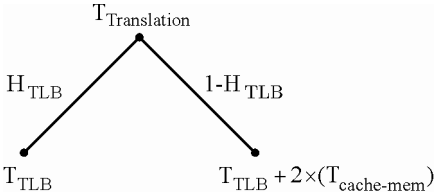
$$T = a + (1-h)\times b$$

پس از ساده‌سازی درخت مطرح شده براساس **اتحاد درخت احتمال**، رابطه زیر را برای محاسبه $T_{Cache-Mem}$ خواهیم داشت:

$$T_{Cache\text{-}mem} = T_{Cache\text{-}FRAME} + (1- H_{Cache}) \times T_{Cache\text{-}FRAME\text{-}MISS}$$

$$T_{Cache\text{-}mem} = 10+(1\text{-}0.9)\times 100=20\ ns$$

در انتها برای بدست آوردن $T_{Translation}$ درخت زیر را در نظر بگیرید:



توجه: وجود ضریب 2 در پشت $T_{Cache\text{-}mem}$ به این دلیل است که برای تولید آدرس فیزیکی، به دو آدرس DIR در جدول صفحه جزئی سطح اول، یعنی PAGE DIRECTORY و آدرس PAGE در جدول صفحه جزئی سطح دوم، یعنی PAGE TABLE نیاز داریم. بنابراین به 2 بار $T_{Cache\text{-}mem}$ و

khalilifar.ir

مراجعه به Cache نیاز است. یعنی یک $T_{Cache-mem}$ برای مراجعه به جدول صفحه جزئی سطح اول و یک $T_{Cache-mem}$ دیگر برای مراجعه به جدول صفحه جزئی سطح دوم.

توجه: در سیستم جداول صفحه چندسطحی درایه‌های موجودی که در TLB قرار دارند، شامل اطلاعات مرتبط همه سطوح است، نمی‌شود سطحی باشد و سطحی نباشد، در سؤال مذکور یا تعدادی از درایه‌های مرتبط دو جدول PAGE DIRECTORY و PAGE TABLE به طور همزمان داخل TLB هستند و یا به طور همزمان داخل TLB نیستند.

پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Translation}$ خواهیم داشت:

$$T_{Translation} = T_{TLB} + (1 - H_{TLB}) \times 2 \times (T_{Cache-mem})$$

$$T_{Translation} = 2 + (1 - 0.98) \times 2 \times (20) = 2.8 \text{ ns}$$

$T_{Destination}$: میانگین زمان دسترسی به مقصد موردنظر

توجه: مطابق فرض سوال در **بخش مقصد** یعنی $T_{Destination}$، سه عنصر T_{Cache}، T_{Mem} و T_{Disk} برای دسترسی به مقصد مورد نظر وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

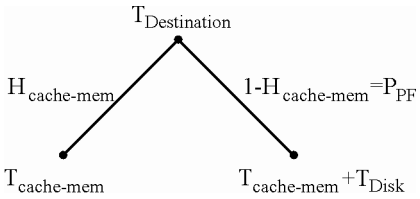
برای محاسبه $T_{Destination}$ ابتدا میانگین T_{Cache} و T_{Mem} را محاسبه می‌کنیم به شکل $T_{Cache-Mem}$، در نهایت هم، میانگین $T_{Cache-Mem}$ و T_{Disk} را محاسبه می‌کنیم که همان $T_{Destination}$ است. به صورت زیر:

مطابق آنچه پیشتر گفتیم، $T_{Cache-Mem}$ به صورت زیر محاسبه شد:

$$T_{Cache-mem} = T_{Cache-FRAME} + (1 - H_{Cache}) \times T_{Cache-FRAME-MISS}$$

$$T_{Cache-mem} = 10 + (1-0.9) \times 100 = 20 \text{ ns}$$

در انتها برای بدست آوردن $T_{Destination}$ درخت زیر را در نظر بگیرید:



پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Destination}$ خواهیم داشت:

$$T_{Destination} = T_{Cache-Mem} + (1 - H_{Cache-Mem}) \times T_{Disk}$$

$$T_{Destination} = T_{Cache-Mem} + (P_{PF}) \times T_{Disk}$$

$$T_{Destination} = 20 + (10^{-6}) \times (5 \times 10^{6}) = 20 + 5 = 25 \text{ ns}$$

و در نهایت براساس رابطه کل زمان دسترسی سیستم خواهیم داشت:

$$T_{Access} = T_{Translation} + T_{Destination}$$

khalilifar.ir

$T_{Access} = 2.8+25=27.8\ ns \cong 28\ ns$

۸- گزینه (۲) صحیح است.

تکنیک‌های برنامه‌نویسی و ساختارهای داده‌ای که اصل مراجعات محلی را رعایت می‌کنند، برای صفحه‌بندی بر پایه تقاضا مناسب هستند.

مانند پشته، جستجوی ترتیبی، کد خالص (محض)، آرایه، بردار (عملیات برداری)، اما تکنیک‌های برنامه‌نویسی و ساختارهای داده‌ای که اصل مراجعات محلی را رعایت نمی‌کنند، برای صفحه بندی بر پایه تقاضا مناسب نیستند. در واقع این ساختارها به محل‌های مختلف پرش می‌کنند.

مانند جدول سمبل hashed، جستجوی دودویی و آدرس‌دهی غیرمستقیم (کارکرد غیرمستقیم)

در حافظه مجازی با صفحه‌بندی درخواستی (بر پایه تقاضا)، صفحات فقط زمانی بارگذاری می‌شوند که در حین اجرای برنامه درخواست شوند و به جای اینکه کل فرآیند به حافظه آورده شود، فقط صفحات مورد نیاز وارد حافظه می‌شوند. بر این اساس تکنیک‌های برنامه‌نویسی و ساختارهایی برای این محیط مناسب هستند که اصل مراجعات محلی در آن‌ها رعایت شود.

انتخاب دقیق تکنیک‌های برنامه‌نویسی و ساختمان داده‌ها می‌تواند، اصل محلی بودن مراجعات را افزایش و نرخ خطای صفحه و تعداد صفحات موجود در مجموعه کاری را کاهش دهد. از این رو پشته ویژگی مراجعات محلی را به خوبی دارد، زیرا دستیابی از بالای آن صورت می‌گیرد و بعدی و بعدی به هم نزدیک هستند. همچنین در جستجوی ترتیبی، کد خالص و عملیات برداری اصلی مراجعات محلی رعایت می‌شود.

اما جدول درهم‌سازی (hashed) از جمله ساختارهایی است که مراجعات را متفرق می‌کند و به محلی بودن آسیب می‌رساند. همچنین جستجوی دودویی و کارکرد غیرمستقیم باعث تضعیف محلی بودن می‌شوند. محلی بودن را مانند شعاع یک دایره در نظر بگیرید، هر چه قدر این شعاع کوچکتر شود محلی بودن بیشتر و بیشتر می‌شود و هر چه قدر این شعاع بزرگتر شود محلی بودن کمتر و کمتر می‌شود. شما با هم محله‌های خود محلی‌تر هستید، اما هر چه قدر از محله خود دور می‌شوید، اهالی محله‌های دیگر با شما کمتر محلی هستند.

۹- گزینه (۱) صحیح است.

در تکنیک صفحه‌بندی، برای هر فرآیند یک جدول صفحه معمولی تشکیل می‌شود که در آن به ازای همه صفحه‌های یک فرآیند، درایه وجود دارد و هر درایه مشخص می‌کند که کدام قاب فیزیکی به این صفحه اختصاص یافته است. در این روش وقتی فرآیندها بزرگ باشند، هزینه نگهداری جداول صفحه بسیار زیاد می‌شود، در ضمن به ازای هر فرآیند نیز باید یک جدول صفحه داشته باشیم. به عبارتی وقتی تعداد و اندازه فرآیندها بزرگ شود، این روش برای ترجمه آدرس مجازی به فیزیکی مقرون به صرفه نیست.

khalilifar.ir

برای حل این مشکل **جدول صفحه چندسطحی** و **جدول صفحه معکوس** ابداع شده است. جدول صفحه معکوس از نظر سربار حافظه بهتر از روش جدول صفحه چندسطحی است و حالت حداقلی است. اغلب سیستم‌های کامپیوتری، فضای آدرس منطقی(مجازی) بزرگی را پشتیبانی می‌کنند، مانند کامپیوترهایی که آدرس‌های 32 یا 64 بیتی را پشتیبانی می‌کنند، در چنین محیط‌هایی جدول صفحه معمولی بسیار بزرگ خواهد شد. برای مثال یک سیستم را با 32 بیت فضای آدرس منطقی(مجازی) در نظر بگیرید. اگر اندازه صفحه در این سیستم برابر با 4KB (2^{12}) باشد، در اینصورت جدول صفحه شامل بیش از یک میلیون درایه خواهد بود. $\left(2^{32} / 2^{12} = 2^{20}\right)$. فرض کنید هر درایه جدول صفحه معمولی، شامل 4 بایت باشد، در اینصورت هر فرآیند به $\left(4 \times 2^{20}\right)$ 4MB فضای آدرس فیزیکی فقط برای نگهداری جدول صفحه معمولی نیاز دارد. همچنین مشخص است که با توجه به محدودیت اندازه قاب (4KB) نمی‌توان این جدول صفحه معمولی را بصورت پیوسته درون یک قاب جا داد. 4MB اندازه جدول صفحه معمولی در فضای 4KB مربوط به یک قاب جا نمی‌شود. یک راه حل تقسیم جدول صفحه معمولی به جداول صفحه جزئی و ایجاد جدول صفحه چندسطحی است، در این حالت جدول صفحه معمولی نیز همانند فرآیند صفحه‌بندی می‌شود. راه حل دیگر استفاده از جدول صفحه معکوس است. در راه حل جدول صفحه معکوس به جای اینکه در جداول صفحه مربوط به هر فرآیند، به ازای هر صفحه مجازی یک درایه داشته باشیم، به ازای هر قاب در حافظه فیزیکی یک درایه در جدول صفحه معکوس نگهداری می‌کنیم. در واقع به ازای هر قاب حافظه اصلی اینکه در حال حاضر کدام صفحه مربوط به کدام فرآیند در این قاب ذخیره شده است نگهداری می‌شود. بنابراین تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی(اصلی) است.

مثال: در یک سیستم صفحه‌بندی که دارای 34 بیت آدرس است 23 بیت اول برای شماره صفحه و 11 بیت بعدی برای آدرس‌دهی درون صفحه است. در یک سیستم با صفحه‌بندی معکوس (Inverted Page Table) با 128 مگابایت حافظه، جدول صفحه دارای چند خانه (درایه یا مدخل) است؟

پاسخ: تعداد درایه‌های جدول صفحه معکوس، مطابق رابطه زیر محاسبه می‌گردد:

تعداد قاب‌های حافظه فیزیکی = تعداد درایه‌های جدول صفحه معکوس

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

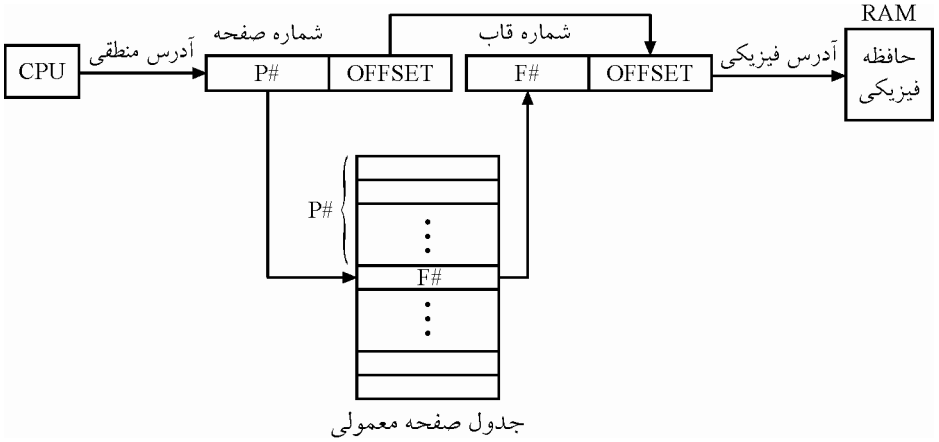
همانطور که گفتیم تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی است، بنابراین تعداد درایه‌های جدول صفحه معکوس به صورت زیر خواهد بود:

$$\text{تعداد درایه‌های جدول صفحه معکوس} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

khalilifar.ir

در یک عبارت ساده، هدف از استفاده از جدول صفحه معکوس، **کاهش** میزان حافظه فیزیکی مورد نیاز برای ترجمه آدرس مجازی به فیزیکی است. با استفاده از تکنیک جدول صفحه معکوس، مقدار زیادی در حافظه فیزیکی صرفه‌جویی می‌شود اما تبدیل آدرس مجازی به فیزیکی کندتر و وقت‌گیرتر می‌شود. در واقع **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم. زیرا جدول صفحه معکوس بر اساس #F اندیس شده است و نه مانند جدول صفحه معمولی که بر اساس #P اندیس شده است. در واقع وقتی یک فرآیند با PID مختص به خود به صفحه مجازی #P مراجعه می‌کند، سخت‌افزار دیگر نمی‌تواند از #P به عنوان اندیس جدول صفحه استفاده کند و صفحه فیزیکی را بیابد و از این جهت باید سرتاسر جدول صفحه معکوس (وارونه) را برای یافتن درایه (PID,P#) از آدرس مجازی (PID,P#,OFFSET) جستجو کند. اگر یک تطبیق پیدا شود، در اینصورت آدرس فیزیکی (F#,OFFSET) تولید خواهد شد و اگر هیچ تطبیقی یافته نشود، در اینصورت یک دسترسی آدرس غیر مجاز می‌باشد. ضمن اینکه این جستجو باید به ازای هربار دسترسی به حافظه انجام شود. همانطور که گفتیم در جدول صفحه معکوس **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم، یعنی سرعت نگاشت آدرس منطقی به فیزیکی **کاهش** می‌یابد. درواقع جدول صفحه معکوس **زمان** نگاشت آدرس منطقی به آدرس فیزیکی را **افزایش** می‌دهد. هرچند موجب صرفه‌جویی در مصرف حافظه فیزیکی می‌شود. **بنابراین گزینه اول نادرست و گزینه دوم درست است.**

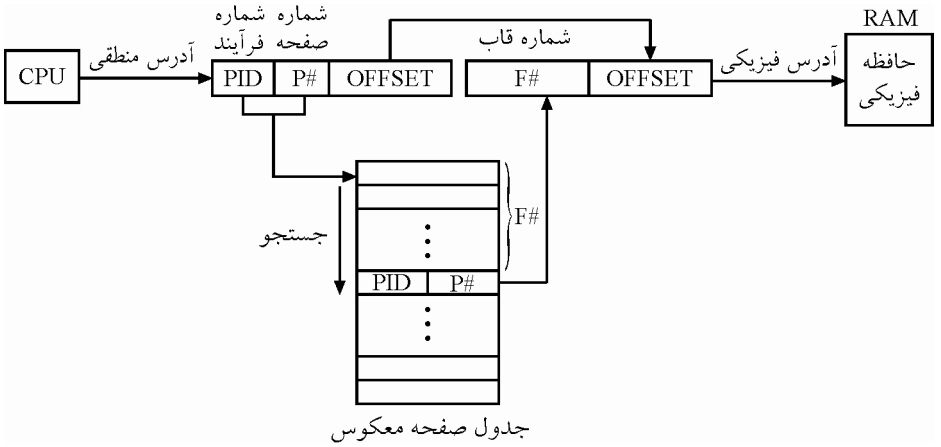
نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معمولی** به صورت زیر است:



جدول صفحه معمولی

نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معکوس** به صورت زیر است:

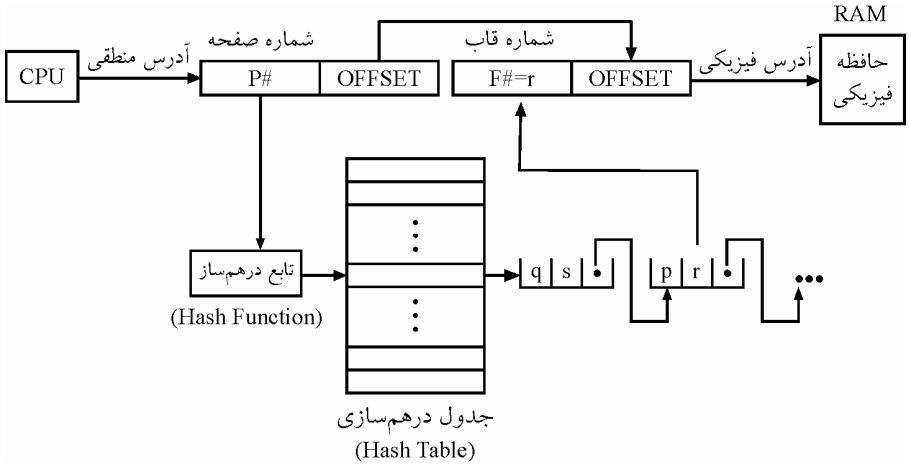
khalilifar.ir



جدول صفحه درهم‌سازی شده

یک راه حل عمومی برای نگاشت فضای آدرس بزرگ به کوچک، مثلا نگاشت فضای آدرس 64 بیتی به 32 بیتی، در سیستمی که فضای آدرس 64 بیتی را پشتیبانی نمی‌کند و فضای 32 بیتی را پشتیبانی می‌کند، استفاده از جدول صفحه درهم‌سازی شده است. تابع درهم‌ساز حجم زیادی از داده را به یک عدد طبیعی تبدیل می‌کند. عدد طبیعی حاصل از تابع درهم‌سازی معمولاً به عنوان اندیس یک آرایه مورد استفاده قرار می‌گیرد. به مقادیر حاصل از تابع درهم‌ساز مقدار درهم(Hashed Value) گفته می‌شود. به عبارت دیگر در اینجا تابع درهم‌ساز مقدار #P از یک آدرس مجازی بزرگ مربوط به یک فرآیند را می‌گیرد و پس از تقسیم بر عدد تابع درهم‌ساز، باقی‌مانده حاصل درایه مشخصی را در جدول درهم‌سازی نشان می‌دهد. هر درایه در جدول درهم‌سازی شامل یک لیست پیوندی از باقی‌مانده‌های یکسان حاصل از تابع درهم‌ساز است. به عبارت دیگر همه صفحات مجازی یک فرآیند که مقدار درهم یکسانی دارند تشکیل یک لیست پیوندی می‌دهند در جلوی یک درایه مشخص از جدول درهم‌سازی. هر گره لیست پیوندی شامل سه فیلد (۱) شماره صفحه مجازی، (۲) شماره قاب و (۳) یک اشاره‌گر به گره بعدی لیست پیوندی است. نحوه کار بدین صورت است که پس از آنکه درایه مرتبط با #P مورد نظر توسط تابع درهم‌ساز، در جدول درهم‌سازی مشخص شد، جستجو در جهت کشف شماره قاب مورد انتظار آغاز می‌شود، بدین صورت که #P مورد نظر با فید اول در گره اول لیست پیوندی مقایسه می‌شود، اگر مطابقت داشت، شماره قاب مورد نظر از فیلد دوم در گره اول جهت ساخت آدرس فیزیکی استخراج می‌شود، در غیر اینصورت گره‌های بعدی جهت کشف شماره قاب مورد انتظار در لیست پیوندی مورد جستجو قرار می‌گیرد. شکل زیر گویای مطلب است:

khalilifar.ir



توجه: برای افزایش سرعت تبدیل آدرس مجازی به فیزیکی در جدول صفحه معکوس می‌توان از تکنیک TLB و یا تفکر جدول درهم‌سازی(Hashe Table) با اندکی تغییر استفاده نمود. بنابراین هر ترجمه آدرس مجازی به فیزیکی نیازمند دوبار مراجعه به حافظه فیزیکی می‌باشد، یکبار برای دسترسی به جدول درهم‌سازی و یکبار دیگر برای دسترسی به جدول صفحه معکوس.

توجه: جدول صفحه معکوس، اطلاعات کاملی در مورد فضای آدرس منطقی یک فرآیند را ندارد. در حالی که سیستم صفحه‌بندی مبتنی بر تقاضا به این اطلاعات کامل جهت پردازش نقص‌های صفحه نیازمند است. برای دسترسی به این اطلاعات کامل، یک **جدول صفحه خارجی**، یکی به ازای هر فرآیند، باید نگهداری شود. جدول صفحه خارجی شامل فیلدهای Valid، Present و آدرس صفحه مجازی دچار نقص صفحه شده بر روی رسانه ذخیره‌سازی است. **بنابراین گزینه چهارم درست است.**

توجه: هنگامی که یک نقص صفحه رخ می‌دهد، ممکن است یک **نقص صفحه دیگر** برای بارگذاری **جدول صفحه خارجی** از رسانه ذخیره‌سازی به حافظه فیزیکی رخ دهد. بنابراین در جدول صفحه معکوس، زمان سرویس نقص صفحه(page fault) به دلیل ایجاد یک نقص صفحه دیگر، **افزایش** می‌یابد. **بنابراین گزینه سوم درست است.**

توجه: در سیستم‌هایی که از جدول صفحه معکوس‌شده استفاده می‌کنند، استفاده اشتراکی از صفحات میان فرآیندها در حالت عادی امکان‌پذیر نیست، در حالت کلی اشتراک به شکل چند به یک است، یعنی صفحه مشترک میان فرآیندها در یک قاب مشخص قرار می‌گیرد. که پیاده‌سازی حالت چند به یک در جدول صفحه معکوس درحالت ابتدایی امکان‌پذیر نیست. زیرا در جدول صفحه معکوس برای هر درایه آن فقط می‌توان شماره یک صفحه از یک فرآیند را قرار داد. البته توسط مکانیزم‌هایی می‌توان اشتراک‌گذاری صفحات میان فرآیندها را در جدول صفحه معکوس نیز پیاده‌سازی نمود. برای مثال به جدول صفحه معکوس اجازه دهیم برای هر درایه، بیش از یک شماره صفحه را آدرس‌دهی کند.

khalilifar.ir

۱۰- گزینه (۳) صحیح است.

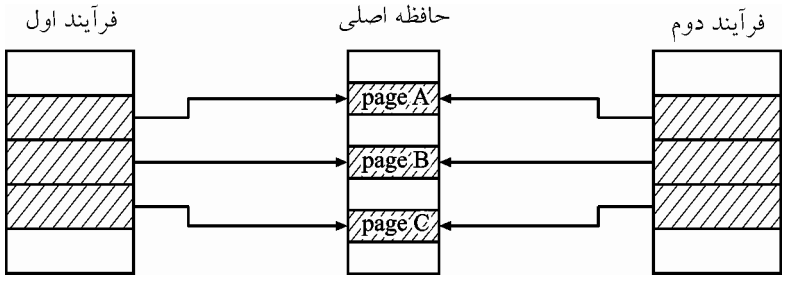
تکنیک صفحه‌بندی اصلاً تکه‌تکه شدن خارجی ندارد. بنابراین گزینه سوم نادرست است. در این تکنیک کاهش اندازه‌ی صفحه، سبب کاهش تکه‌تکه شدن داخلی (در صفحه آخر هر فرآیند) می‌شود، بنابراین گزینه چهارم درست است. کاهش اندازه صفحه سبب کاهش زمان سرویس نقص صفحه می‌شود، چون کاهش اندازه صفحه منجر به کاهش عملیات مبادله صفحه موردنظر مابین حافظه اصلی و هارد دیسک می‌شود. بنابراین گزینه اول درست است. کاهش اندازه صفحه سبب افزایش بهره‌وری حافظه می‌گردد، چون کاهش اندازه صفحه سبب کاهش تکه تکه شدن داخلی می‌شود که این امر منجر به افزایش بهره‌وری حافظه می‌گردد. همچنین کاهش اندازه صفحه سبب افزایش زمان I/O می‌شود چون هر چند کاهش اندازه صفحه منجر به کاهش عملیات مبادله صفحه موردنظر مابین حافظه اصلی و هارد دیسک می‌شود، اما در مجموع تعداد و تعدد این مبادله‌ها مابین حافظه اصلی و هارد دیسک زیاد می‌شود به دلیل کوچک بودن اندازه صفحات و به تبع ازدیاد تعداد صفحات و این یعنی افزایش زمان I/O برای کل صفحات مبادله شده. بنابراین گزینه دوم درست است.

۱۱- گزینه (۲) صحیح است.

در تکنیک copy-on-write با ایجاد فرآیند جدید توسط دستور fork، صفحات فرآیند پدر برای فرزند کپی نمی‌شوند، بلکه این صفحات بین فرآیند پدر و فرزند به اشتراک گذاشته می‌شوند. هر زمان که یکی از دو فرآیند پدر یا فرزند بخواهد صفحه‌ای را تغییر دهد، یک کپی جداگانه از آن صفحه برای آن فرآیند ساخته می‌شود و فرآیند از آن به بعد از آن صفحه استفاده می‌کند (به جای صفحه مشترک) و فرآیندهای دیگر از صفحه اصلی (مشترک) استفاده می‌کنند.

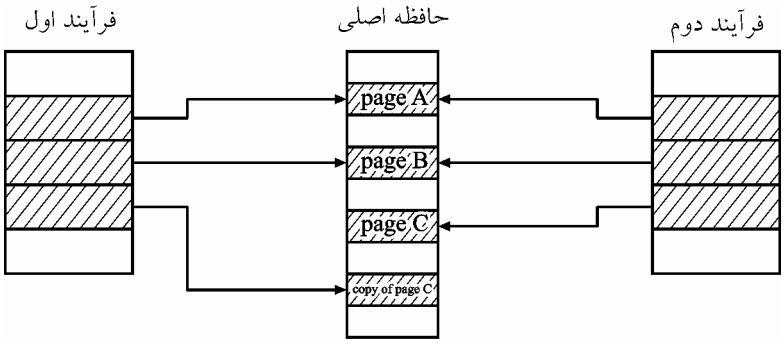
در این تکنیک زمانی فرآیند پدر می‌تواند خاتمه یابد که یک کپی از صفحات آن برای هر یک از فرزندانش ایجاد شده باشد (یعنی صفحات تمامی فرآیندهای فرزند تغییر کرده باشند، به عبارت دیگر همه فرزندان همه صفحه‌های مربوط به خود را تغییر داده باشند.) و دیگر نیازی به صفحات فرآیند پدر نباشد.

شکل زیر گویای مطالب می‌باشد:



قبل از اینکه فرآیند اول صفحه C را تغییر دهد.

khalilifar.ir



بعد از اینکه فرآیند اول صفحه C را تغییر دهد.

۱۲- گزینه (۳) صحیح است.

به طور کلی میانگین زمان دسترسی سیستم به مقصد نهایی برای هر آدرس سیستم از رابطه زیر محاسبه می‌گردد:

$$T_{Access} = T_{Translation} + T_{Destination}$$

که پارامترهای آن شامل $T_{Translation}$ یعنی **بخش ترجمه** و $T_{Destination}$ یعنی **بخش مقصد** می‌باشد.

$T_{Translation}$: زمان ترجمه

توجه: مطابق فرض سوال در **بخش ترجمه** یعنی $T_{Translation}$، یک عنصر T_{Mem} برای ترجمه وجود دارد که مقدار آن بیان نشده است، بنابراین مقدار آنرا صفر در نظر می‌گیریم. به صورت زیر:

$$T_{Mem} = 0$$

بنابراین زمان **ترجمه** به صورت زیر است:

$$T_{Translation} = 0$$

$T_{Destination}$: میانگین زمان دسترسی به مقصد موردنظر

توجه: مطابق فرض سوال در **بخش مقصد** یعنی $T_{Destination}$، دو عنصر T_{Mem} و T_{Disk} برای دسترسی به مقصد مورد نظر وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

برای محاسبه $T_{Destination}$ میانگین T_{Mem} و T_{Disk} را محاسبه می‌کنیم که همان $T_{Destination}$ است.

مطابق مفروضات مطرح شده در صورت سوال سه وضعیت زیر برقرار است:

الف) صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه منتقل شود، اما قاب خالی در حافظه موجود نیست، بنابراین یک صفحه باید از حافظه خارج گردد، تا فضای کافی برای ورود صفحه مورد درخواست فراهم گردد. ولی صفحه‌ای که باید از حافظه خارج گردد، **تمیز** است، یعنی کپی صفحه موجود در حافظه با کپی موجود در دیسک یکی است، یعنی صفحه تغییر نکرده

khalilifar.ir

است، بنابراین نیاز به دوباره نویسی ندارد. یعنی صفحه جدید بر روی صفحه کاندید برای خروج، جایگزین می‌گردد، بدون آنکه صفحه کاندید برای خروج، اطلاعتش بر روی دیسک ذخیره گردد.
مطابق فرض مسأله $\frac{1}{2}$ صفحات **تمیز** یعنی تغییر نیافته‌اند به شکلی که قاب خالی هم در حافظه موجود نیست یا مستقل از **تمیز** و **کثیف** بودن صفحات، قاب خالی در حافظه موجود است که مطابق آنچه گفتیم رابطه زیر برقرار است:

$$\frac{1}{2}\times T_{Disk}$$

ب) صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه منتقل شود، اما قاب خالی در حافظه موجود است. بنابراین صفحه مورد درخواست در قاب خالی قرار می‌گیرد.
مطابق فرض مسأله $\frac{1}{2}$ صفحات **تمیز** یعنی تغییر نیافته‌اند به شکلی که قاب خالی هم در حافظه موجود نیست یا مستقل از **تمیز** و **کثیف** بودن صفحات، قاب خالی در حافظه موجود است که مطابق آنچه گفتیم رابطه زیر برقرار است:

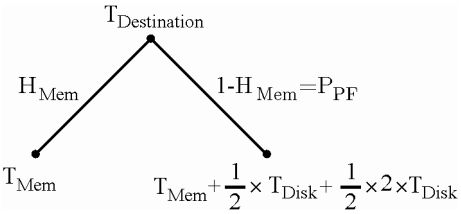
$$\frac{1}{2}\times T_{Disk}$$

ج) صفحه مورد درخواست در دیسک قرار دارد که باید به حافظه منتقل شود، اما قاب خالی در حافظه موجود نیست، بنابراین یک صفحه باید از حافظه خارج گردد تا فضای کافی برای ورود صفحه مورد درخواست فراهم گردد. ولی صفحه‌ای که باید از حافظه خارج گردد، **کثیف** است، یعنی کپی صفحه‌ی موجود در حافظه با کپی صفحه موجود در دیسک یکی نیست، یعنی صفحه تغییر کرده‌است، بنابراین نیاز به دوباره‌نویسی دارد. یعنی قبل از آنکه صفحه جدید بر روی صفحه‌ی کاندید برای خروج جایگزین گردد، ابتدا اطلاعات صفحه کاندید برای خروج بر روی دیسک ذخیره می‌گردد که یک $T_{RW}=T_{Disk}$ زمان نیاز دارد، سپس صفحه جدید بر روی صفحه کاندید برای خروج، جایگزین می‌گردد، که مجددا یک T_{Disk} زمان نیاز دارد.
مطابق فرض مسأله $\frac{1}{2}$ صفحات **کثیف** یعنی تغییر یافته‌اند به شکلی که قاب خالی هم در حافظه موجود نیست که مطابق آنچه گفتیم رابطه زیر برقرار است:

$$\frac{1}{2}\times\left[T_{RW}+T_{Disk}\right]=\frac{1}{2}\times 2\times T_{Disk}$$

بنابراین صفحات تغییریافته یا **کثیف** نیاز به دوبار جابه‌جایی بین حافظه و دیسک دارند، یک بار برای قرار دادن صفحه کثیف از حافظه به دیسک و یک بار دیگر برای قرار دادن صفحه مورد درخواست از دیسک به حافظه. البته اگر قاب خالی در حافظه نباشد.
برای بدست آوردن $T_{Destination}$ درخت زیر را در نظر بگیرید:

khalilifar.ir



پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Destination}$ خواهیم داشت:

$$T_{Destination} = T_{Mem} + \left(1-H_{Mem}\right) \times\left[\frac{1}{2}\times T_{Disk} + \frac{1}{2}\times 2\times T_{Disk}\right]$$

$$T_{Destination} = T_{Mem} + \left(P_{PF}\right) \times\left[\frac{1}{2}\times T_{Disk} + \frac{1}{2}\times 2\times T_{Disk}\right]$$

$$T_{Destination} = 0 + \left(P\right) \times\left[\frac{1}{2}\times d + \frac{1}{2}\times 2\times d\right]$$

$$T_{Destination} = \left(P\right) \times\left[\frac{1}{2}\times d + \frac{1}{2}\times 2\times d\right] = \frac{1}{2}\times p\times d + \frac{1}{2}\times p\times 2d = \frac{p\times d}{2} + \frac{2\times p\times d}{2} = \frac{3pd}{2}$$

و در نهایت براساس رابطه کل زمان دسترسی سیستم خواهیم داشت:

$$T_{Access(old)} = T_{Translation} + T_{Destination}$$

$$T_{Access(old)} = 0 + \frac{3pd}{2} = \frac{3pd}{2}$$

حال اگر سرعت حافظه جانبی ۲ برابر شود، یعنی زمان d به $\frac{d}{2}$ کاهش یابد، آنگاه رابطه زیر برقرار است:

$$T_{Access(new)} = T_{Translation} + T_{Destination}$$

$$T_{Access(new)} = 0 + \frac{3p\frac{d}{2}}{2} = \frac{3pd}{4}$$

میزان کاهش برابر تفاضل دو حالت **قدیم** و **جدید** می‌باشد، به صورت زیر:

$$T_{Access(old)} - T_{Access(new)} = \frac{3pd}{2} - \frac{3pd}{4} = \frac{6pd}{4} - \frac{3pd}{4} = \frac{3pd}{4} = \frac{3}{4}pd$$

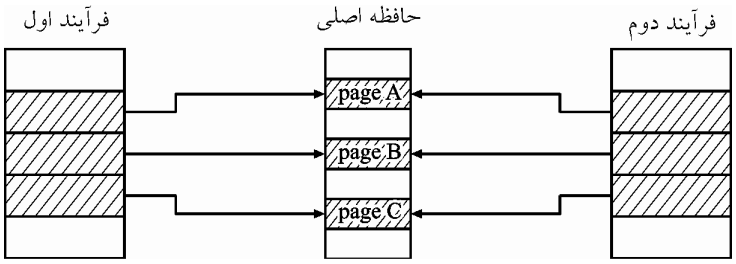
بنابراین گزینه سوم پاسخ سؤال خواهد بود.

۱۳- گزینه (۱ و ۳) صحیح است.

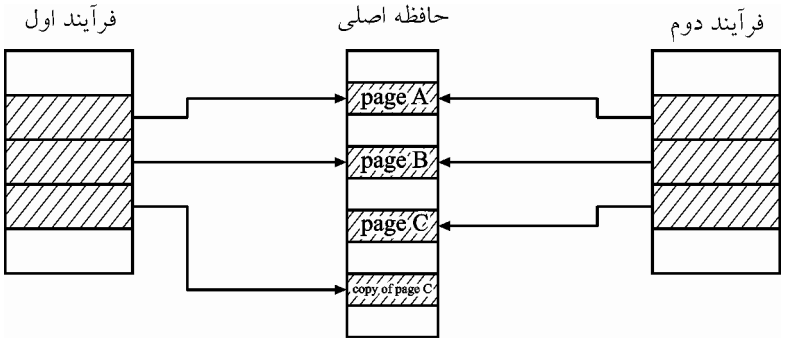
در unix برای ایجاد یک فرآیند جدید از دستور ( )fork استفاده می‌شود و این کار را با ایجاد یک فرآیند تکراری و دقیقاً یکسان با داده‌های مشترک با فرآیند پدر انجام می‌دهد. در واقع در ابتدای کار داده‌های Heap و Code بین فرآیند پدر و فرزند به اشتراک گذاشته می‌شود. اما در ادامه اگر

khalilifar.ir

پدر و فرزند قصد اعمال تغییراتی را در داده‌های مشترک خود داشته باشند، در اینصورت هـر یـک حافظه‌های مختص به خود را در اختیار می‌گیرند.



قبل از اینکه فرآیند اول صفحه C را تغییر دهد.



بعد از اینکه فرآیند اول صفحه C را تغییر دهد.

فراخوانی fork یک مقدار برمی‌گرداند که برای فرآیند فرزند برابر صفر و برای فرآیند پدر برابـر بـا شناسه فرآیند (PID) فرزند خواهد بود. بنابراین با استفاده از عدد PID حاصل از بازگشت، می‌توان متوجه شد که بین دو فرآیند، کدامیک پدر و کدامیک فرزند است.

توجه: هر فرآیند Process id، Stack و رجیسترهای مختص به خود را دارد.

توجه: سازمان سنجش آموزش کشور، ابتدا در کلید اولیه خود گزینه اول را به عنوان پاسـخ اعـلام نمود، اما در کلید نهایی گزینه‌های اول و سوم را به عنوان پاسخ نهایی اعلام نمود، که کـار درسـتی بوده است.

۱۴- گزینه (۴) صحیح است.

یک راه مقابله با Trashing کنترل تعداد وقفه‌های نقص صـفحه بـا اسـتفاده از الگـوریتم فرکـانس نقص صفحه یا PFF (Page Fault Frequency) است. این الگوریتم زمان کاهش یا افـزایش تعـداد قاب صفحه تخصیص یافته به یک فرآیند را بیان می‌کند، اما کنترلی در مورد اینکه کدام صفحه باید

khalilifar.ir

در هنگام نقص صفحه جایگزین شود، انجام نمی‌دهد. این الگوریتم فقط اندازه مجموعه تخصیص را کنترل می‌کند. که بهتر است، این مجموعه تخصیص برابر مجموعه کاری (working set) باشد.
روال کار بدین صورت است که اگر تعداد نقص‌های صفحه برای یک فرآیند افزایش یافت، باید تعدادی قاب حافظه به آن اختصاص یابد و اگر تعداد نقص‌های صفحه برای یک فرآیند از یک حد پایین کمتر شد، باید تعدادی قاب از آن فرآیند پس گرفته شود. نکته مهم اینجاست که اگر تعداد نقص‌های صفحه یک فرآیند بالا رفت، ولی قاب آزاد در حافظه وجود نداشت، باید درجه چند برنامگی سیستم را کاهش داد و یک یا چند فرآیند را به حالت معلق در آورد تا قاب‌هایی که در اختیار دارد، آزاد شود.

۱۵- گزینه (۴) صحیح است.

یک راه حل مقابله با Trashing کنترل تعداد وقفه‌های نقص صفحه با استفاده از الگوریتم فرکانس نقص صفحه یا PFF (Page Fault Frequency) است. این الگوریتم زمان کاهش یا افزایش تعداد قاب صفحه تخصیص یافته به یک فرآیند را بیان می‌کند، اما کنترلی در مورد این که کدام صفحه باید در هنگام نقص صفحه جایگزین شود، انجام نمی‌دهد. این الگوریتم فقط اندازه مجموعه تخصیص را کنترل می‌کند. که بهتر است، این مجموعه تخصیص برابر مجموعه کاری (working set) باشد. روال کار بدین صورت است که اگر تعداد نقص‌های صفحه برای یک فرآیند افزایش یافت، باید تعدادی قاب حافظه به آن اختصاص یابد و اگر تعداد نقص‌های صفحه برای یک فرآیند از یک حد پایین کم‌تر شد، باید تعدادی قاب از آن فرآیند پس گرفته شود. نکته مهم اینجاست که اگر تعداد نقص‌های صفحه یک فرآیند بالا رفت، ولی قاب آزاد حافظه وجود نداشت، باید درجه چند برنامگی سیستم را کاهش داد و یک یا چند فرآیند را به حالت معلق در آورد تا قاب‌هایی که در اختیار دارد، آزاد شود.

۱۶- گزینه (۳) صحیح است.

الگوریتم LRU (Least Recently Used)

ایده اصلی این الگوریتم این است که اگر صفحه‌ای در چند دستور اخیر مراجعات زیادی داشته است، به احتمال قوی در دستورات بعدی هم ارجاعات زیادی خواهد داشت، همچنین اگر یک صفحه، اخیرا هیچ مراجعه‌ای نداشته، احتمالاً در آینده نزدیک هم ارجاعی نخواهد داشت. در واقع این الگوریتم بیان می‌کند هنگام وقوع خطای نقص صفحه، صفحه‌ای را حذف کنید که طولانی‌ترین زمان عدم استفاده را دارد.
می‌توان گفت LRU تقریبی از الگوریتم بهینه می‌باشد که در آن به جای توجه به آینده، به گذشته توجه می‌شود.

khalilifar.ir

توجه: الگوریتم LRU ناهنجاری بی‌لیدی ندارد، بنابراین با افزایش تعداد قاب، همواره، نقص صفحه کاهش می‌یابد.

توجه: بهتر بود طراح محترم، تعداد قاب‌های صفحه موجود در RAM را بیان می‌نمود، اما به واسطه آزمون و خطا می‌توان به پاسخ احتمالی طراح رسید.

اگر 1 قاب در نظر گرفته شود:

واضح است که در این حالت، به ازای تمام در خواست‌ها، نقص صفحه رخ خواهد داد، که این مقدار برابر 20 نقض صفحه می‌باشد.

اگر 2 قاب در نظر گرفته شود:

نگاه به گذشته ←

رشته مراجعات	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
قاب 1	7	7	1	1	0	0	0	0	2	2	0	0	2	2	2	2	1	1	0	0
قاب 2		0	0	2	2	3	3	4	4	3	3	3	3	1	1	0	0	7	7	1
نقص صفحه	*	*	*	*	*	*		*	*	*	*		*	*		*	*	*	*	*

در این حالت 17 نقص صفحه به وقوع پیوست.

اگر 3 قاب در نظر گرفته شود:

نگاه به گذشته ←

رشته مراجعات	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
قاب 1	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
قاب 2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
قاب 3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
نقص صفحه	*	*	*	*		*		*	*	*	*			*		*		*		

در این حالت 12 نقص صفحه به وقوع پیوست، که برابر گزینه سوم است.

اگر 4 قاب در نظر گرفته شود:

نگاه به گذشته ←

رشته مراجعات	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
قاب 1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
قاب 2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
قاب 3			1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1
قاب 4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
نقص صفحه	*	*	*	*		*		*						*				*		

khalilifar.ir

در این حالت 8 نقص صفحه به وقوع پیوست.

توجه: اگر این روال را ادامه دهیم، یعنی تعداد قاب‌ها را افزایش دهیم، مطابق خاصیت الگوریتم LRU، واضح است که نقص صفحه کاهش می‌یابد. بنابراین واضح است که گزینه سوم درست است.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه سوم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

سخنی با طراح: طراح محترم، سوال استاندارد باید مستقل از گزینه‌ها قابل حل باشد و وابسته به گزینه‌ها نباشد. این را دیگه همه می‌دانند، همه ...

۱۷- گزینه (۲) صحیح است.

به طور کلی میانگین زمان دسترسی سیستم به مقصد نهایی برای هر آدرس سیستم از رابطه زیر محاسبه می‌گردد:

$$T_{Access} = T_{Translation} + T_{Destination}$$

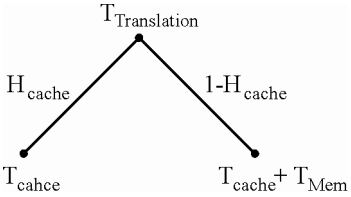
که پارامترهای آن شامل $T_{Translation}$ یعنی **بخش ترجمه** و $T_{Destination}$ یعنی **بخش مقصد** می‌باشد.

$\mathbf{T_{Translation}}$: میانگین زمان ترجمه

توجه: مطابق فرض سوال در **بخش ترجمه** یعنی $T_{Translation}$، دو عنصر T_{Cache} و T_{Mem} برای ترجمه وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

برای محاسبه $T_{Translation}$ میانگین T_{Cache} و T_{Mem} را محاسبه می‌کنیم که همان $T_{Translation}$ است. به صورت زیر:

برای بدست آوردن $T_{Translation}$ درخت زیر را در نظر بگیرید:



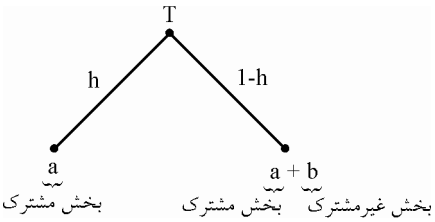
پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Translation}$ خواهیم داشت:

$$T_{Translation} = T_{Cache} + (1-H_{Cache}) \times T_{Mem}$$

$$T_{Translation} = 10 + (1-H_{Cache}) \times 200$$

توجه: جهت ساده‌سازی رابطه درخت فوق، همواره می‌توان از **اتحاد درخت احتمال** به صورت زیر، استفاده نمود:

khalilifar.ir



$T =$ بخش مشترک $+ (1-h) \times$ بخش غیرمشترک

$$T = a + (1-h) \times b$$

پس از ساده‌سازی درخت مطرح شده براساس **اتحاد درخت احتمال**، رابطه زیر را برای محاسبه $T_{Translation}$ خواهیم داشت:

$$T_{Translation} = T_{Cache} + (1\text{-}H_{Cache}) \times T_{Mem}$$

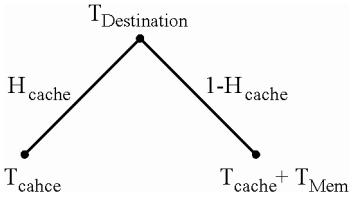
$$T_{Translation} = 10 + (1\text{-}H_{Cache}) \times 200$$

$T_{Destination}$: میانگین زمان دسترسی به مقصد موردنظر

توجه: مطابق فرض سوال در **بخش مقصد** یعنی $T_{Destination}$، دو عنصر T_{Cache} و T_{Mem} برای دسترسی به مقصد مورد نظر وجود دارد، بنابراین باید میانگین آنها محاسبه گردد.

برای محاسبه $T_{Destination}$ میانگین T_{Cache} و T_{Mem} را محاسبه می‌کنیم که همان $T_{Destination}$ است. به صورت زیر:

برای بدست آوردن $T_{Destination}$ درخت زیر را در نظر بگیرید:



پس از ساده‌سازی درخت فوق رابطه زیر را برای محاسبه $T_{Destination}$ خواهیم داشت:

$$T_{Destination} = T_{Cache} + (1\text{-}H_{Cache}) \times T_{Mem}$$

$$T_{Destination} = 10 + (1\text{-}H_{Cache}) \times 200$$

و در نهایت براساس رابطه کل زمان دسترسی سیستم خواهیم داشت:

$$T_{Access} = T_{Translation} + T_{Destination}$$

$$T_{Access} = \left[10 + (1-H_{Cache}) \times 200\right] + \left[10 + (1-H_{Cache}) \times 200\right]$$

مطابق فرض سوال، زمان دسترسی موثر (T_{Access}) در این سیستم، 10 درصد بیشتر از زمان دسترسی به کش (T_{cache}) است، بنابراین داریم:

khalilifar.ir

$$T_{Access} = T_{cache} + \frac{1}{10}T_{cache} = 10 + \frac{1}{10} \times 10 = 11 \text{ ns}$$

همچنین براساس رابطه کل زمان دسترسی سیستم خواهیم داشت:

$$T_{Access} = T_{Translation} + T_{Destination}$$

$$T_{Access} = \left[T_{cache} + (1 - H_{cache})T_{Mem}\right] + \left[T_{cache} + (1 - H_{cache})T_{Mem}\right]$$

$$T_{Access} = \left[10 + \left(1 - H_{Cache}\right) \times 200\right] + \left[10 + \left(1 - H_{Cache}\right) \times 200\right]$$

$$11 = \left[10 + \left(1 - H_{Cache}\right) \times 200\right] + \left[10 + \left(1 - H_{Cache}\right) \times 200\right]$$

$$11 = 20 + \left(1 - H_{Cache}\right) \times 400$$

رابطه فوق نشان می‌دهد که حاصل $400 \times \left(1 - H_{Cache}\right)$ باید برابر مقدار 9- باشد که معنا ندارد.
نتیجه اینکه منظور طراح محترم از زمان دسترسی موثر سیستم همان فقط زمان ترجمه سیستم است، و نه کل زمان دسترسی موثر سیستم که البته فرض نادرستی هم هست، در ادامه بر اساس فرض طراح، مساله را حل می‌کنیم.
همانطور که پیشتر گفتیم رابطه زیر برای محاسبه $T_{Translation}$ برقرار است:

$$T_{Translation} = T_{Cache} + (1\text{-}H_{Cache}) \times T_{Mem}$$

$$T_{Translation} = 10 + (1\text{-}H_{Cache}) \times 200$$

مطابق فرض سوال، زمان دسترسی موثر در این سیستم، 10 درصد بیشتر از زمان دسترسی به کش (T_{cache}) است، بنابراین داریم:

$$T_{Translation} = T_{cache} + \frac{1}{10}T_{cache} = 10 + \frac{1}{10} \times 10 = 11 \text{ ns}$$

همچنین براساس رابطه زمان ترجمه سیستم داریم:

$$T_{Translation} = T_{cache} + (1 - H_{cache}) \times T_{Mem}$$

$$11 = 10 + (1 - H_{cache}) \times 200$$

$$1 = (1 - H_{cache}) \times 200$$

$$1 - H_{cache} = \frac{1}{200}$$

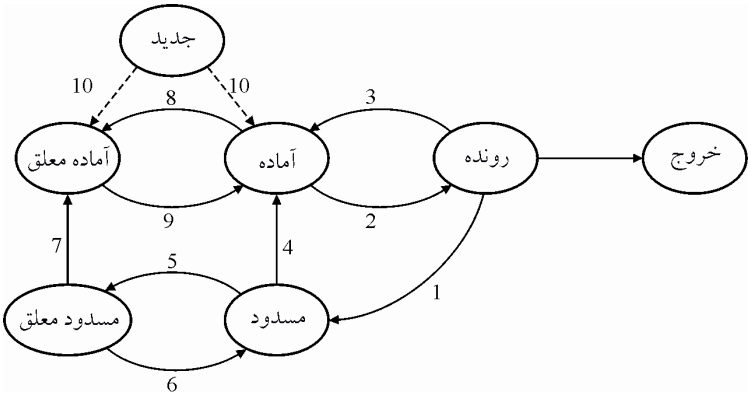
$$H_{cache} = 1 - \frac{1}{200} = \frac{199}{200}$$

بنابراین گزینه دوم پاسخ مد نظر طراح سوال بوده است. سازمان سنجش آموزش کشور نیز گزینه دوم را به عنوان پاسخ نهایی سوال اعلام کرده بود.

khalilifar.ir

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

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



۱) 10,7,4
۲) 2,4,1
۳) 8,5,2
۴) 9,6

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

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

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

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

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

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

۱) 141، 108
۲) 135، 102

khalilifar.ir

۳) 133، 99

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

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

۱) امکان‌پذیر است ولی می‌تواند به بن‌بست بیانجامد.

۲) امکان‌پذیر است ولی نتیجه غلط خواهد داد.

۳) امکان‌پذیر است و مشکلی ندارد.

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

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

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

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

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

khalilifar.ir

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

کدام گزینه صحیح است؟ (مهندسی IT- دولتی۹۰)

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

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

khalilifar.ir

(مهندسی کامپیوتر – دولتی۹۱)

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

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

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

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

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

(مهندسی IT- دولتی۹۱)

Fig A

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

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

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

Fig B

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

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

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

Fig C

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

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

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

Fig D

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

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

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

khalilifar.ir

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

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

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

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

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

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

کدام یک از گزینه‌های زیر درست نیست؟

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

khalilifar.ir

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

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

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

کدام یک از گزینه‌های زیر درست نیست؟

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

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

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

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

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

---

khalilifar.ir

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

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

۱) 4,7,6,-3
۲) 4,7,6,13,-3
۳) 4,1,6,-3,14
۴) 4,7,6,13,-3,14

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

۱) اگر دو تا فیلسوف چپ دست یا دو فیلسوف راست دست کنار هم باشند بن بست رخ می‌دهد.
۲) مستقل از نحوه‌ی نشستن فیلسوفان چپ دست و راست دست، هیچگاه بن بست رخ نمی‌دهد.
۳) اگر از یک نوع فیلسوف، دو تا و از نوع دیگر سه تا داشته باشیم بن بست رخ می‌دهد.
۴) اگر همگی فیلسوف‌ها با هم همزمان اولین چنگال‌ها را بردارند، بن بست رخ می‌دهد.

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

(مهندسی IT – دولتی ۹۴)

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

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

۱) دقیقاً دو بار
۲) حداقل دو بار
۳) دقیقاً سه بار
۴) حداقل سه بار

---

khalilifar.ir

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

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

۱) 4 ۲) 5 ۳) 9 ۴) 10

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

(مهندسی IT- دولتی ۹۶)

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

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

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

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

---

khalilifar.ir

پاسخ تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۱- گزینه (۴) صحیح است.

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

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

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

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

اول اینکه، پردازنده آزاد می‌گردد.

دوم اینکه، بلوک‌هایی از حافظه آزاد می‌گردد.

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

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

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

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

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

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

khalilifar.ir

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

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

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

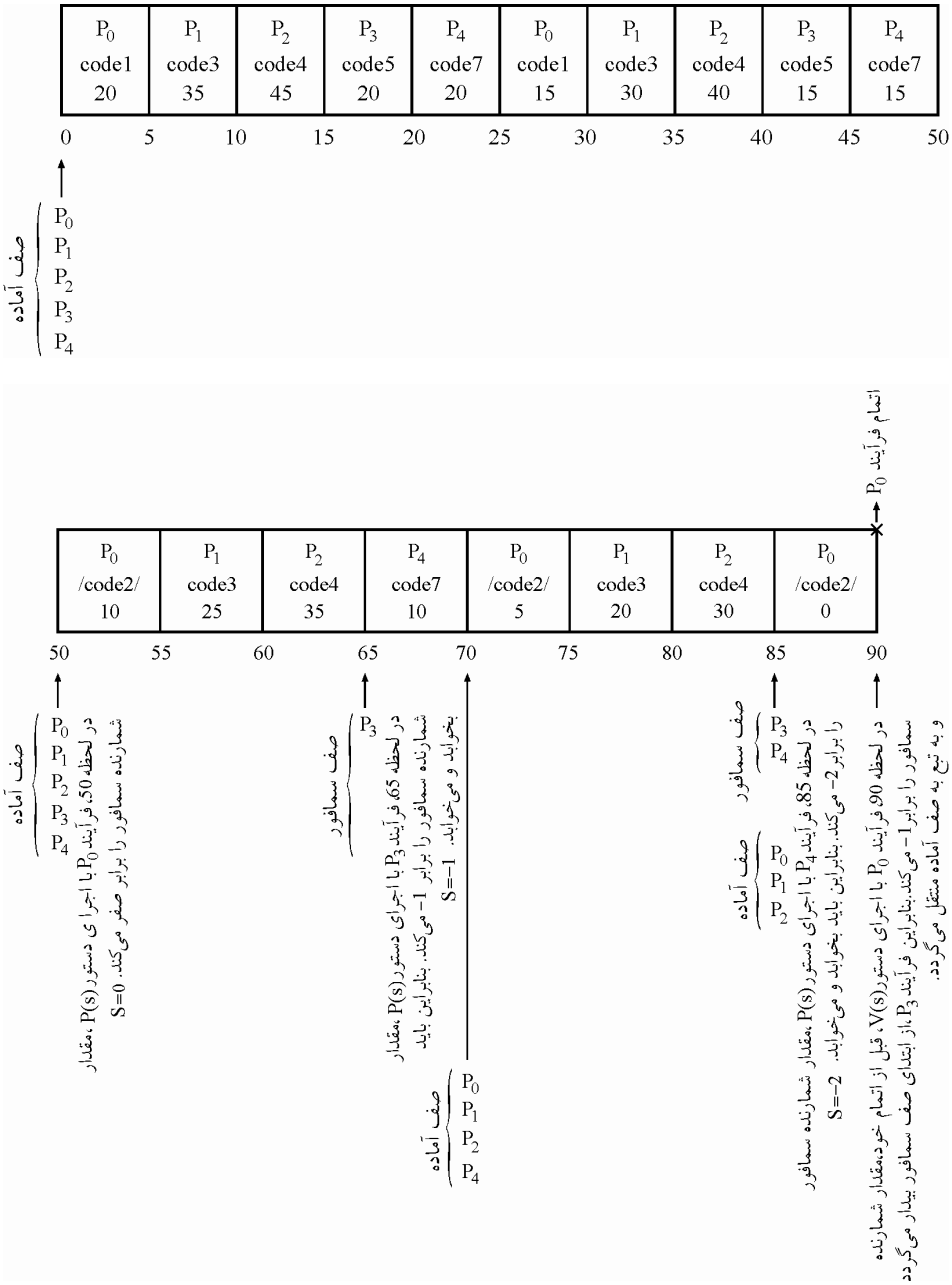
۲- گزینه ( ) صحیح است.

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

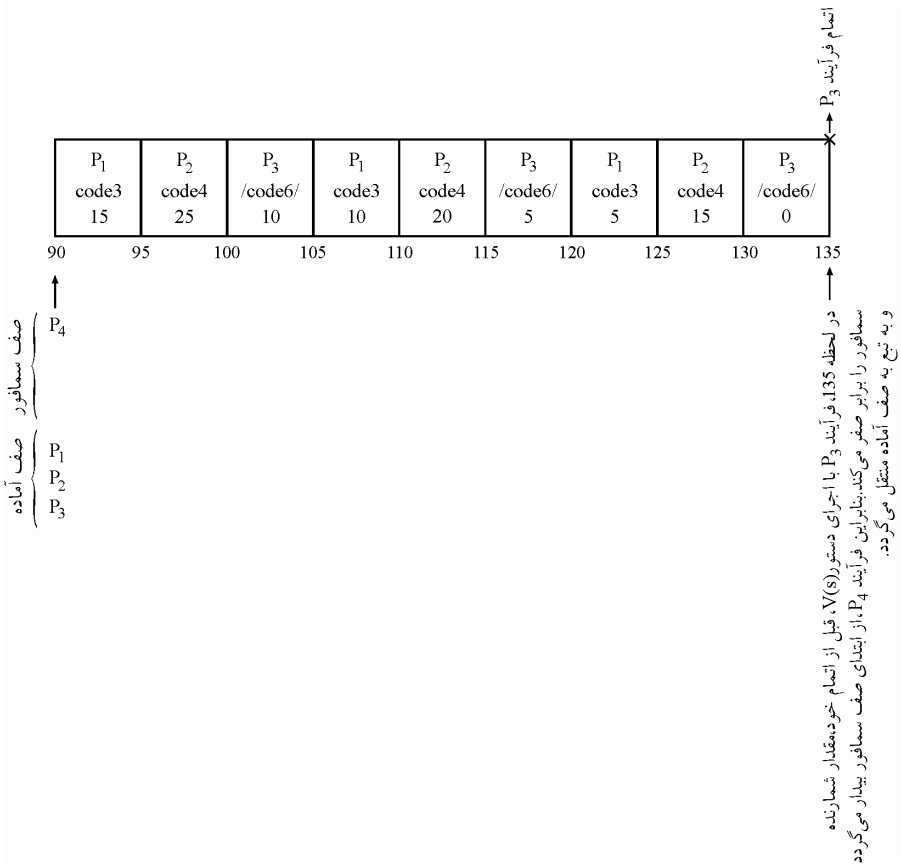
khalilifar.ir

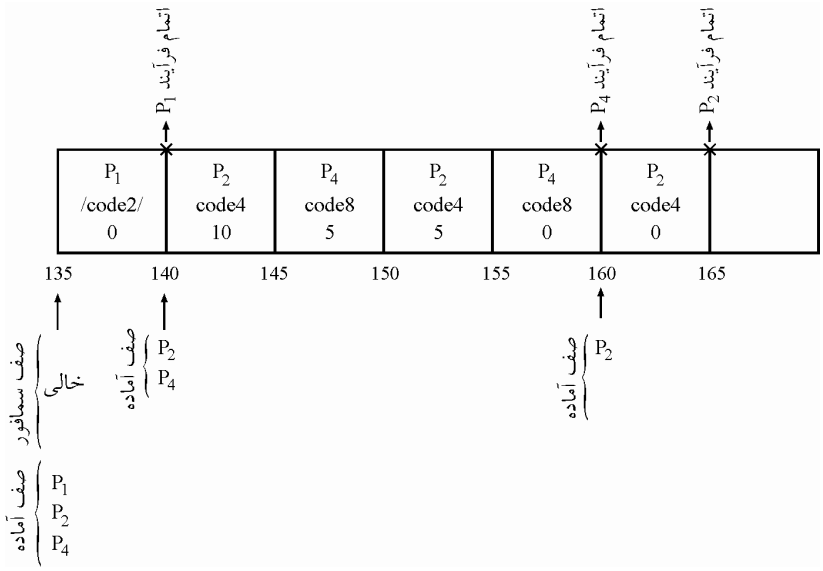
راه حل اول:

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



khalilifar.ir

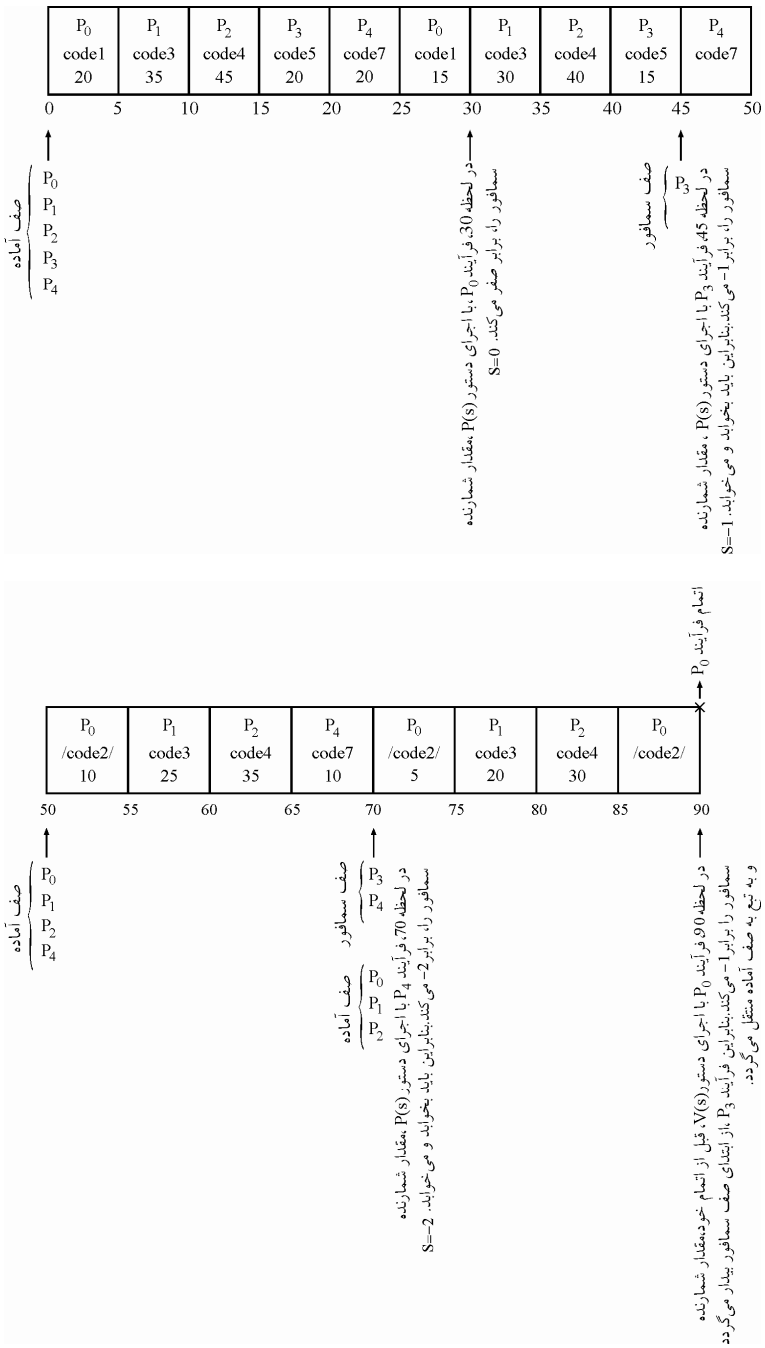




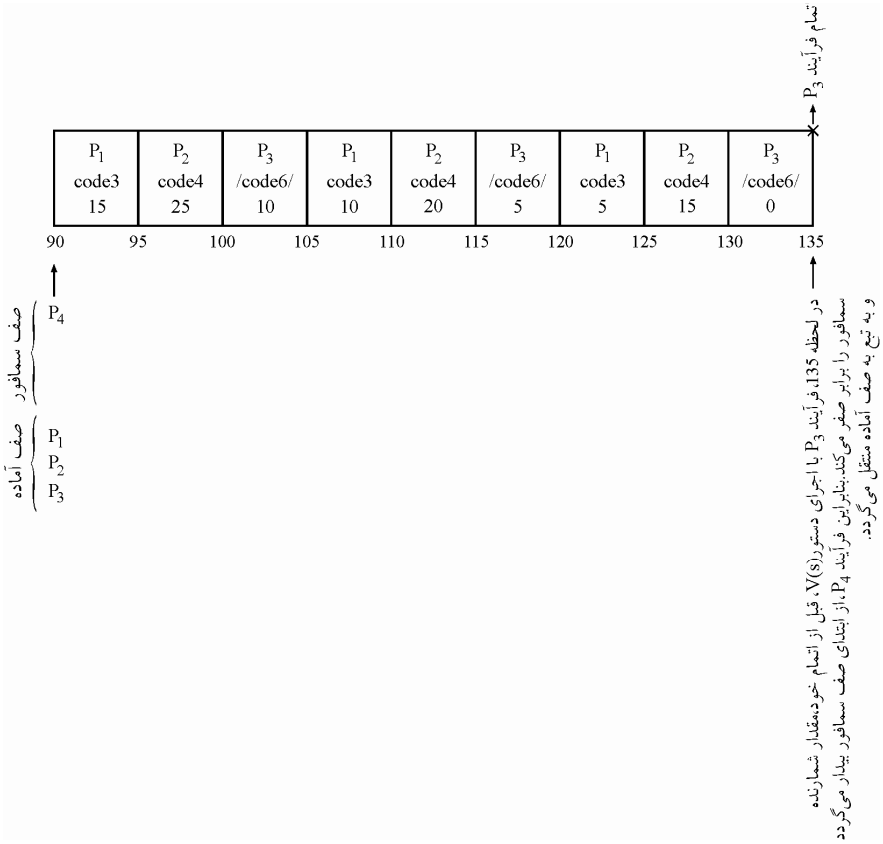
khalilifar.ir

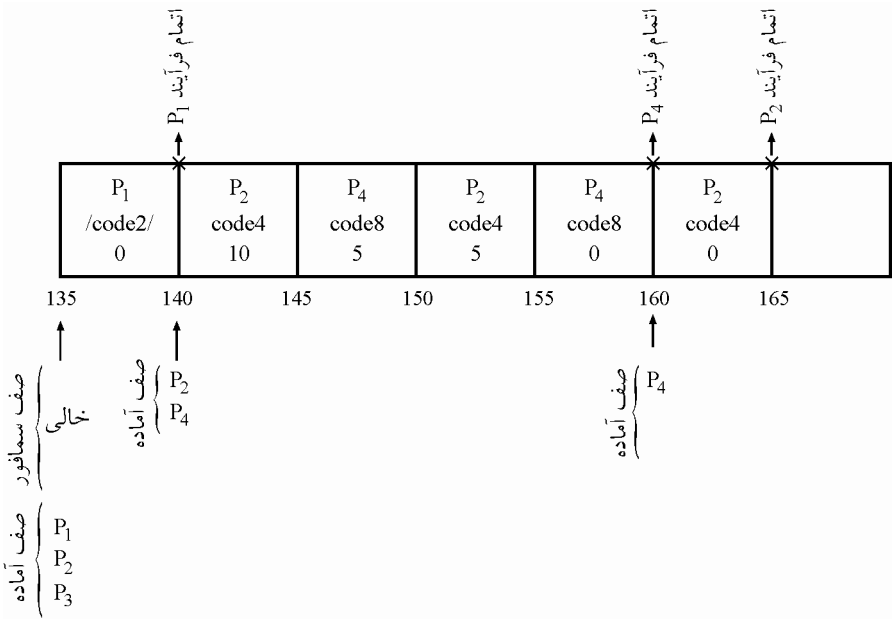
راه حل دوم:

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



khalilifar.ir





khalilifar.ir

در هر دو صورت حل، داریم:

زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_0 زمان بازگشت = $90 - 0 = 90$

P_1 زمان بازگشت = $140 - 0 = 140$

P_2 زمان بازگشت = $165 - 0 = 165$

P_3 زمان بازگشت = $135 - 0 = 135$

P_4 زمان بازگشت = $160 - 0 = 160$

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

زمان اجرای فرآیند– زمان بازگشت فرآیند = زمان انتظار فرآیند

P_0 زمان انتظار = $90 - 25 = 65$

P_1 زمان انتظار = $140 - 40 = 100$

P_2 زمان انتظار = $165 - 50 = 115$

P_3 زمان انتظار = $135 - 25 = 110$

P_4 زمان انتظار = $160 - 25 = 135$

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

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

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

138=33+105

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

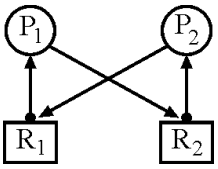
توجه: برای مجموعه فرآیندهای P_0، P_3 و P_4 نیز هیچگاه بن بست رخ نمی‌دهد، برای این مجموعه فرآیندها زمانی بن بست رخ می‌دهد که هر سه فرآیند بخوابند و دیگر کسی نباشد که آنها را بیدار کند، یعنی به خواب ابدی بروند، اما مطابق آنچه در اجرای فرآیندها دیدیم، همه فرآیندها پس از اجرای کامل، از سیستم خارج شدند و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

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

khalilifar.ir

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

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



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

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

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

khalilifar.ir

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

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

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

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

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

حال اگر در این لحظه فرآیند جدیدی به نام P_5 درخواست ورود به ناحیه بحرانی را دهد، با اجرای دستور wait، مقدار سمافور را 2- کرده و به صف سمافور اضافه می‌شود. از آنجایی که با اجرای $\text{block}(-(s \rightarrow \text{value}))$ (+2, -2) فرآیند P_5 رتبه 2+ را دارد، جلوتر از فرآیند P_4 (که رتبه 3 را داشت) قرار می‌گیرد و اولویت اجرایش بالاتر از P_4 می‌شود. $(\overrightarrow{P_4, P_5})$ سناریوی فوق می‌تواند بارها و بارها تکرار شده و تخصیص ناحیه بحرانی به فرآیندهای جدید مانند P_6، P_7 و غیره، با رتبه بالاتری نسبت به P_4 انجام شود.

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

khalilifar.ir

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

این سناریو می‌تواند بارها و بارها تکرار شود.

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

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

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

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

۵- گزینه (۴) صحیح است.

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

khalilifar.ir

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

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

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

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

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

khalilifar.ir

۶- گزینه (۱) صحیح است.

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

توجه: فرض می‌شود مقدار شمارنده سمافور برابر یک است.

Fig A:

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

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

Fig B:

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

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

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

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

Fig C:

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

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

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

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

khalilifar.ir

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

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

:Fig D

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

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

۷- گزینه (۳) صحیح است.

شرایط رقابتی (مسابقه)

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

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

khalilifar.ir

۱- شرط انحصار متقابل

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

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

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

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

khalilifar.ir

۲- شرط پیشرفت

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

۳- شرط انتظار محدود

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

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

قحطی (گرسنگی)

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

بن‌بست

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

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

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

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

khalilifar.ir

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

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

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

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

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

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

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

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

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

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

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

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

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

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

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

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

khalilifar.ir

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

P1:

/*critical_section*/

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

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

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

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

P2:

① key[2]= TRUE;

② while (key[2]) swap (key[2],lock);

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

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

swap (key[2],lock);

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

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

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

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

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

khalilifar.ir

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

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

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

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

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

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

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

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

```
P1:
① key[1]=TRUE;
```

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

```
P2:
① key[2]=TRUE;
```

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

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

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

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

khalilifar.ir

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

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

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

swap (key[1],lock);

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

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

P1:
/*critical_section*/

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

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

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

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

swap (key[1],lock);

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

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

khalilifar.ir

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

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

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

قانون سوم ارسطو (آزمون شرط پیشرفت)

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

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

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

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

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

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

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

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

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

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

khalilifar.ir

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

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

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

P1:

/*critical_section*/

③ lock=FALSE;

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

P1:

/*remainder_section*/

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

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

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

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

P2:

① key[2]= TRUE;

② while (key[2]) swap (key[2],lock);

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

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

swap (key[2],lock);

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

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

P2:

/*critical_section*/

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

khalilifar.ir

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

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

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

P2:
/*remainder_section*/

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

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

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

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

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

swap (key[2],lock);

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

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

P2:
/*critical_section*/

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

khalilifar.ir

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)

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

قانون چهارم ارسطو (آزمون گرسنگی)

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

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

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

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

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

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

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

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

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

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

khalilifar.ir

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

khalilifar.ir

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

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

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

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

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

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

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

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

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

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

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

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

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

khalilifar.ir

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

قانون دوم ارسطو(آزمون بن بست)

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

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

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

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

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

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

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

```
P1:
① key[1]=TRUE;
```

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

```
P2:
① key[2]=TRUE;
```

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

khalilifar.ir

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

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

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

P1:

② while (key[1]) swap (key[1],lock);

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

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

swap (key[1],lock);

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

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

P1:

/*critical_section*/

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

P2:

② while (key[2]) swap (key[2],lock);

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

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

swap (key[1],lock);

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

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

khalilifar.ir

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

فرم ساده قانون دوم ارسطو (آزمون بن بست)

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

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

صورت سوال به این شکل است:

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

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

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

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

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

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

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

۴) راه حل صحیح است.

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

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

۸- گزینه ( ) صحیح است.

شرایط رقابتی (مسابقه)

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

khalilifar.ir

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

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

۱- شرط انحصار متقابل

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

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

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

khalilifar.ir

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

۲- شرط پیشرفت

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

۳- شرط انتظار محدود

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

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

قحطی (گرسنگی)

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

بن‌بست

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

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

khalilifar.ir

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

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

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

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

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

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

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

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

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

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

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

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

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

توجه: هم اکنون C[1] = TRUE است.

khalilifar.ir

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

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

خط اول فرآیند P1 را اجرا کنید.

```
P1:
① C[1] =FALSE;
```

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

خط دوم فرآیند P0 را مجددا اجرا کنید.

```
P0:
② while(C[1]) do;
```

توجه: هم اکنون C[1] = FALSE است.

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

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

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

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

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

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

```
P1:
② while(C[0]) do;
```

توجه: هم اکنون C[0] = FALSE است.

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

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

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

khalilifar.ir

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

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

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

قانون دوم ارسطو (آزمون دوم شرط انحصار متقابل)

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

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

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

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

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

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

```
P0:
① C[0]=FALSE;
```

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

```
P1:
① C[1]=FALSE;
```

khalilifar.ir

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

P_0:
② while (C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

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

P0:
/*critical_section*/

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

P_1:
② while (C[0]) do;

توجه: هم اکنون C[0] = FALSE است.

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

P1:
/*critical_section*/

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

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

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

khalilifar.ir

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

قانون سوم ارسطو (آزمون شرط پیشرفت)

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

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

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

P_0:
① C[0]= FALSE;
② while(C[1]) do;

توجه: هم اکنون C[1] = TRUE است.

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

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

خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =FALSE;

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

خط دوم فرآیند P0 را مجددا اجرا کنید.

P0:
② while(C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

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

P0:
/*critical_section*/

khalilifar.ir

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

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

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

P0:

/*critical_section*/

③ C[0]=TRUE;

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

P0:

/*remainder_section*/

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

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

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

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

P_1:

② while (C[0]) do;

توجه: هم اکنون C[0] = TRUE است.

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

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

khalilifar.ir

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)

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

قانون چهارم ارسطو (آزمون گرسنگی)

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

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

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

P_0:

① C[0]= FALSE;

② while(C[1]) do;

توجه: هم اکنون C[1] = TRUE است.

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

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

خط اول فرآیند P1 را اجرا کنید.

P1:

① C[1] =FALSE;

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

خط دوم فرآیند P0 را مجددا اجرا کنید.

P0:

② while(C[1]) do;

khalilifar.ir

توجه: هم اکنون C[1] = FALSE است.

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

P0:

/*critical_section*/

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

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

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

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

P1:

② while(C[0]) do;

توجه: هم اکنون C[0] = FALSE است.

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

P1:

/*critical_section*/

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

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

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

khalilifar.ir

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

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

قانون دوم ارسطو(آزمون بن بست)

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

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

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

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

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

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

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

```
P0:
① C[0]=FALSE;
```

khalilifar.ir

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

P1:
① C[1]=FALSE;

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

P_0:
② while (C[1]) do;

توجه: هم اکنون C[1] = FALSE است.

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

P0:
/*critical_section*/

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

P_1:
② while (C[0]) do;

توجه: هم اکنون C[0] = FALSE است.

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

P1:
/*critical_section*/

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

فرم ساده قانون دوم ارسطو (آزمون بن بست)

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

khalilifar.ir

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

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

صورت سوال به این شکل است:

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

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

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

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

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

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

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

۴) این راه حل شرط پیشرفت را برآورده می‌کند.

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

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

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

۹- گزینه ( ) صحیح است.

شرایط رقابتی (مسابقه)

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

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

khalilifar.ir

۱- شرط انحصار متقابل

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

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

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

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

khalilifar.ir

۲- شرط پیشرفت

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

۳- شرط انتظار محدود

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

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

قحطی (گرسنگی)

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

بن‌بست

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

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

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

khalilifar.ir

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

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

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

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

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

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

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

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

قانون اول ارسطو(آزمون اول شرط انحصار متقابل)

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

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

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

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

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

khalilifar.ir

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

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

دو خط اول فرآیند P1 را اجرا کنید.

```
P1:
① C[1] =TRUE;
② turn= 0;
```

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

خط سوم فرآیند P0 را مجددا اجرا کنید.

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

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

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

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

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

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

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

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

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

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

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

khalilifar.ir

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

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

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

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

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

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

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

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

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

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

```
P0:
① C[0]= TRUE;
```

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

```
P1:
① C[1]= TRUE;
```

khalilifar.ir

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود بـه ناحیـه‌ی بحرانـی را دارنـد. بنـابراین تابلوی وضعیت فرآیندها هر دو C[0] = TRUE و C[1] = TRUE می‌شود، زیرا هر دو فرآینـد بـه شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. اما متغیر نوبت turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآینـد، شـماره‌ی فرآینـد خـود را در متغیر نوبت turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیـره کنـد، فرآینـدی اسـت کـه شماره‌اش در متغیر نوبت turn باقی می‌ماند و دیگری اثرش در متغیر نوبت turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغییر turn گره خورده است، بنـابراین فرآینـدی که دیرتر متغیر نوبت turn را مقداردهی کرده است، باید صـبر پیشـه کنـد و متغیـر نوبـت turn را نگه‌داری کند و در حلقه‌ی انتظار بچرخد. و فرآیندی که زودتر متغیر نوبت turn را مقداردهی کرده است، وارد ناحیه بحرانی می‌شود.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنـابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتـی کـه دو فرآینـد بـه دسـتور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود، اما فرآینـد P1 بایـد در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P_0:
② turn= 1;

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

P_1:
② turn= 0;

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

P_0:
③ while(C[0] && turn = 1) do;

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

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

P0:
/*critical_section*/

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

P_0:
③ while(C[1] && turn = 0) do;

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

khalilifar.ir

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

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

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

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هـر دو تونستن به طور همزمان وارد باجه تلفن همگانی بشنن اونوقت شرط دوم انحصار متقابل نقـض شده. اخلاق می‌گه دو نفر نباید همزمان باهم داخل باجه تلفن همگانی باشـن، یعنـی اگـه یـه نفـر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگـه بشـه شـرط دوم انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

توجه: برای برقرار بودن شرط انحصار متقابل باید **قانون اول ارسطو** (آزمـون اول شـرط انحصـار متقابل) و **قانون دوم ارسطو** (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشـند. بنـابراین شرط انحصار متقابل در سوال مطرح شده برقرار است.

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سـپس همـان فرآینـد اول را داخل ناحیه باقی‌مانده خودش قرار بـده، **(گـام ۳)** در ادامـه فرآینـد دوم را داخـل ناحیـه بحرانـی خودش قرار بده، **(گام ۴)** سپس همان فرآیند دوم را داخل ناحیـه بـاقی‌مانـده خـودش قـرار بـده، **(گام ۵)** در نهایت همان فرآیند دوم به ابتدای برنامه برگردد و مجـددا تصـمیم بگیـرد وارد ناحیـه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانـی خـودش شـود، آنگـاه در ایـن حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست.

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

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

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

khalilifar.ir

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

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

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

دو خط اول فرآیند P1 را اجرا کنید.

```
P1:
① C[1] =TRUE;
② turn= 0;
```

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

خط سوم فرآیند P0 را مجددا اجرا کنید.

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

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

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

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

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

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

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

```
P0:
/*critical_section*/
④ C[0]= FALSE;
```

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

```
P0:
/*remainder_section*/
```

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

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

khalilifar.ir

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

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

P1:
③ while(C[1] && turn = 0) do;

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

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

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

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)

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

قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، **(گام ۳)** در ادامه فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۴)** در نهایت همان فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه

khalilifar.ir

بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.

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

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

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

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

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

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

دو خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =TRUE;
② turn= 0;

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

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:
③ while(C[0] && turn = 1) do;

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

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

P0:
/*critical_section*/

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

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

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

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

P1:
③ while(C[1] && turn = 0) do;

khalilifar.ir

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

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

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۳) می‌شویم. هم اکنون پردازنده پشت ناحیه بحرانی فرآیند P1 در یک حلقه انتظار، دچار انتظار مشغول است.

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

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

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

P0:
/*critical_section*/
④ C[0]=FALSE;

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

P0:
/*remainder_section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۴) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

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

فرض کنید فرآیند P0 قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

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

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

khalilifar.ir

توجه: شاید در این مرحله پیش خودتان فکر کنید که خب مثل سابق دوباره سراغ فرآیند P1 می‌رویم و با turn= 0 در خط ② راه را برای ورود فرآیند P0 باز می‌کنیم، اول اینکه خب نادرست فکر کردید، دقت کنید که شما اصلا نمی‌توانید مجددا سراغ خط ② از فرآیند P1 بروید چون قبلا از آن خط عبور کرده بودید و اگر هم سراغ فرآیند P1 بروید در ادامه همان خط ③ و حلقه while را مشاهده خواهید کرد. مانند انداختن یک سکه که بعد انداختن به سمت پایین حرکت می‌کند، مگر اینکه دوباره آنرا به سمت بالا پرتاپ کنیم. دوم اینکه از قوانین تبعیت کنید و به هیچ عنوان به مراحل آن دست نزنید و فقط و فقط مطابق قوانین حرکت کنید. اینطوری موفق می‌شوید.

همانطور که در (گام ۴) گفتیم قرار شد که فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب نشد، فرآیند اول نتوانست مجددا وارد ناحیه بحرانی خودش بشود. بنابراین گرسنگی رخ نداده است.

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

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

قانون دوم ارسطو(آزمون بن بست)

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

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند که در این حالت بن‌بست رخ داده است.

khalilifar.ir

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

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

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

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

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

P0:
① C[0]= TRUE;

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

P1:
① C[1]= TRUE;

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

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتی که دو فرآیند به دستور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود اما فرآیند P1 باید در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

khalilifar.ir

P_0:
② turn= 1;

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

P_1:
② turn= 0;

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

P_0:
③ while(C[0] && turn = 1) do;

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

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

P_0:
/*critical_section*/

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

P_0:
③ while(C[1] && turn = 0) do;

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

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

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

فرم ساده قانون دوم ارسطو (آزمون بن بست)

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

khalilifar.ir

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

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

صورت سوال به این شکل است:

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

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

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

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

گزینه دوم **نادرست** است، زیرا شرط انحصار متقابل هم برقرار است و برآورده می‌کند.

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

گزینه سوم **نادرست** است، زیرا شرط انتظار محدود هم برقرار است و برآورده می‌کند.

۴) این راه حل تنها شرط پیشرفت را برآورده می‌کند.

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

توجه: همانطور که واضح و مشخص هست، همه گزینه‌ها نادرست هستند و هر چهار گزینه می‌تواند پاسخ سوال باشد.

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

۱۰- گزینه (۴) صحیح است.

در راه حل سمافور اجرای دو عمل wait و signal باید به صورت اتمیک و تجزیه‌ناپذیر باشند. چرا که در این اعمال نیز باید **انحصار متقابل** رعایت شود و به عبارت دیگر، در هر زمان فقط یک فرآیند حق دستکاری شمارنده سمافور را با یکی از دو عمل مزبور داشته باشد. راه حل معمولی این است که wait و signal را به صورت فراخوان سیستمی پیاده‌سازی کنیم. آن وقت هسته یک سیستم عامل تک پردازنده می‌تواند به راحتی تمام وقفه‌ها را در ابتدای رویه‌های wait و signal از کار بیندازد و پس از انجام رویه دوباره فعال سازد. از آن‌جا که تعداد دستورالعمل‌های هر یک از دو فراخوان اندک‌اند، از کار انداختن وقفه‌ها هیچ اشکالی را به وجود نخواهد آورد. اما این راه حل در سیستم‌های تک پردازنده‌ای کارآمد خواهد بود، زیرا هر فرآیند قادر به از کار انداختن وقفه‌های پردازنده جاری خود خواهد بود، و ممکن است فرآیند دیگری، توسط پردازنده دیگری، همزمان با فرآیند اول پس از عبور تقریباً همزمان از خطوط تابع wait وارد ناحیه بحرانی گردد. بنابراین در سیستم‌های چند پردازنده‌ای، به دنبال راه حل دیگری باید بود. یک راه حل این است که شمارنده

khalilifar.ir

سمافور، یا در وسعت بزرگ‌تر کل الگوریتم تابع wait به عنوان یک ناحیه بحرانی، در بخش ناحیه بحرانی راه حل TSL قرار گیرد. از آن‌جا که راه حل TSL گذرگاه حافظه را مسدود می‌کند، بنابراین مطابق شرح احوال الگوریتم TSL، این راه حل گارانتی می‌کند که در هر لحظه فقط یک پردازنده به شمارنده سمافور دسترسی داشته باشد. در راه حل TSL مشکل انتظار مشغول وجود دارد.

همانطور که گفتیم در راه حل سمافور دو عمل wait و signal باید به صورت اتمیک و تجزیه ناپذیر باشند. چراکه آنچه اهمیت دارد، برقراری شرط انحصار متقابل در پیاده‌سازی دو عمل wait و signal است.

ساختار کلی این راه‌حل به صورت زیر می‌باشد:

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

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

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

راه‌حل سمافور بر دو دسته‌ی کلی (۱) سمافور عمومی و (۲) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور عمومی می‌پردازیم:

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
Int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

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

khalilifar.ir

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله‌ی انتظار مشغول و حل مسأله‌ی اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این‌گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

بر روی سمافور عمومی s دو تابع wait(s) و signal(s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
```

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

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

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

ساختار این تابع به صورت زیر است:

khalilifar.ir

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

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

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

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

ساختار کلی از کار انداختن وقفه در سیستم **تک پردازنده‌ای**، به صورت زیر می‌باشد:

```
Disable_Interupts ();
Critical _section ();
Enable_ Interupts ();
remainder_section ();
```

ساده‌ترین راه‌حل، آن است که هر فرآیند بلافاصله پس از ورود به ناحیه‌ی بحرانی‌اش، تمام وقفه‌ها را از کار بیندازد و دقیقاً قبل از خروج از ناحیه‌ی بحرانی همه‌ی آن‌ها را مجدداً فعال سازد. با متوقف کردن وقفه‌ها، دیگر وقفه‌های ساعت نیز رخ نخواهد داد. پردازنده فقط در نتیجه‌ی وقوع وقفه‌های ساعت و یا وقفه‌های ورودی و خروجی است که می‌تواند از یک فرآیند به فرآیندی دیگر تعویض متن کند. پس با از کار انداختن وقفه‌ها، پردازنده دیگر تحت هیچ شرایطی قادر

khalilifar.ir

نخواهد بود از فرآیندی به فرآیند دیگر تعویض متن کند. بنابراین هنگامی که یک فرآیند وقفه‌ها را غیر فعال می‌کند، می‌تواند بدون هیچ مشکلی و بدون ترس از مداخله‌ی دیگر فرآیندها به خواندن و نوشتن در حافظه‌ی مشترک بپردازد. پس برای رعایت شرط انحصار متقابل **در سیستم‌های تک‌پردازنده‌ای** کافی است، وقفه‌ها متوقف شوند.

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

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

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک پردازنده‌ای** به کمک از کار انداختن وقفه به صورت زیر می‌باشد:

```
Disable_Interupts ();
wait (semaphore)
    {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
Enable_ Interupts ();
critical_section ();
Disable_Interupts ();
signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
    if (s.count < = 0)
    {
     remove a process from queue;
     wake up ();
      }
}
Enable_ Interupts ();
remainder_section ();
```

khalilifar.ir

توجه: از کار انداختن وقفه‌ها، در سیستم‌های تک‌پردازنده‌ای، اغلب در داخل خود سیستم عامل برای مدیریت نواحی بحرانی همچون اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal، تکنیک مفیدی است، اما برای مدیریت نواحی بحرانی فرآیندهای کاربر، به دلیل فراموش‌کار بودن کاربر، تکنیک مناسبی نیست.

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

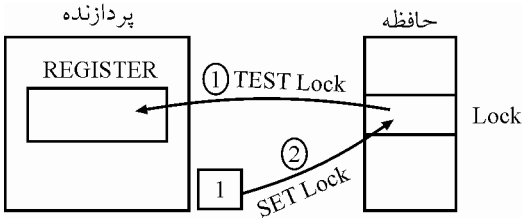
ساختار کلی دستورالعمل TSL در سیستم **تک و چند پردازنده‌ای**، به صورت زیر می‌باشد:

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

بسیاری از پردازنده‌ها، دستورالعمل دو بخشی اما اتمیک خاصی دارند به نام TSL (TEST and SET Lock)، بدین نحو که این دستورالعمل به شکل تجزیه ناپذیر، عملیات زیر را انجام می‌دهد:

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

به شکل زیر توجه کنید:



سخت‌افزار تضمین می‌کند که دو عمل خواندن و مقداردهی متغیر قفل به صورت اتمیک (تجزیه ناپذیر) انجام شود. بدین شکل که هیچ فرآیند و حتی پردازنده‌ی دیگری نتواند به این متغیر قفل دسترسی پیدا کند تا وقتی که اجرای دستورالعمل به پایان برسد. پردازنده‌ای که دستورالعمل TSL

khalilifar.ir

را اجرا می‌کند، گذرگاه حافظه را قفل می‌کند تا از دسترسی دیگر پردازنده‌ها به حافظه جلوگیری کند تا اینکه این دستورالعمل به پایان برسد.

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

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

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

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

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

khalilifar.ir

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک و چند پردازنده‌ای** به کمک دستورالعمل TSL به صورت زیر می‌باشد:

```
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Wait (semaphore)
    {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
Add this process to s.queue;
block ();
  }
}
exit_section
MOVE LOCK, 0
RET
critical_section ();
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
    if (s.count< = 0)
    {
     Remove a process from queue;
     Wake up ();
      }
}
exit_section
MOVE LOCK, 0
```

khalilifar.ir

RET
remainder_section ();

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

توجه: این راه‌حل، فقط در پردازنده‌هایی قابل اجراست که دستورالعمل TSL را پشتیبانی می‌کنند.

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

۱۱- گزینه (۱) صحیح است.

در راه حل سمافور اجرای دو عمل wait و signal باید به صورت اتمیک و تجزیه‌ناپذیر باشند. چرا که در این اعمال نیز باید **انحصار متقابل** رعایت شود و به عبارت دیگر، در هر زمان فقط یک فرآیند حق دستکاری شمارنده سمافور را با یکی از دو عمل مزبور داشته باشد. راه حل معمولی این است که wait و signal را به صورت فراخوان سیستمی پیاده‌سازی کنیم. آن وقت هسته یک سیستم عامل تک پردازنده می‌تواند به راحتی تمام وقفه‌ها را در ابتدای رویه‌های wait و signal از کار بیندازد و پس از انجام رویه دوباره فعال سازد. از آن‌جا که تعداد دستورالعمل‌های هر یک از دو فراخوان اندک‌اند، از کار انداختن وقفه‌ها هیچ اشکالی را به وجود نخواهد آورد. اما این راه حل در سیستم‌های تک پردازنده‌ای کارآمد خواهد بود، زیرا هر فرآیند قادر به از کار انداختن وقفه‌های پردازنده جاری خود خواهد بود، و ممکن است فرآیند دیگری، توسط پردازنده دیگری، همزمان با فرآیند اول پس از عبور تقریباً همزمان از خطوط تابع wait وارد ناحیه بحرانی گردد. بنابراین در سیستم‌های چند پردازنده‌ای، به دنبال راه حل دیگری باید بود. یک راه حل این است که شمارنده سمافور، یا در وسعت بزرگ‌تر کل الگوریتم تابع wait به عنوان یک ناحیه بحرانی، در بخش ناحیه بحرانی راه حل TSL قرار گیرد. از آن‌جا که راه حل TSL گذرگاه حافظه را مسدود می‌کند، بنابراین مطابق شرح احوال الگوریتم TSL، این راه حل گارانتی می‌کند که در هر لحظه فقط یک پردازنده به شمارنده سمافور دسترسی داشته باشد. در راه حل TSL مشکل انتظار مشغول وجود دارد.

همانطور که گفتیم در راه حل سمافور دو عمل wait و signal باید به صورت اتمیک و تجزیه ناپذیر باشند. چراکه آنچه اهمیت دارد، برقراری شرط انحصار متقابل در پیاده‌سازی دو عمل wait و signal است.

ساختار کلی این راه‌حل به صورت زیر می‌باشد:

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

khalilifar.ir

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

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

راه‌حل سمافور بر دو دسته‌ی کلی(۱) سمافور عمومی و (۲) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور عمومی می‌پردازیم:

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
Int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

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

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله‌ی انتظار مشغول و حل مسأله‌ی اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این‌گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

بر روی سمافور عمومی s دو تابع wait(s) و signal(s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

khalilifar.ir

```
wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
```

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

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

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

ساختار این تابع به صورت زیر است:

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

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

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند علاقه‌مند به خروج از ناحیه‌ی بحرانی، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به ناحیه‌ی بحرانی که در حال حاضر در صف سمافور قرار

khalilifar.ir

دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up() بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

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

ساختار کلی از کار انداختن وقفه در سیستم **تک پردازنده‌ای**، به صورت زیر می‌باشد:

```
Disable_Interupts ();
Critical _section ();
Enable_ Interupts ();
remainder_section ();
```

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

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

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

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

khalilifar.ir

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک پردازنده‌ای** به کمک از کار انداختن وقفه به صورت زیر می‌باشد:

```
Disable_Interupts ();
wait (semaphore)
   {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
add this process to s.queue;
block ();
  }
}
Enable_ Interupts ();
critical_section ();
Disable_Interupts ();
signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
   if (s.count < = 0)
   {
    remove a process from queue;
    wake up ();
     }
}
Enable_ Interupts ();
remainder_section ();
```

توجه: از کار انداختن وقفه‌ها، در سیستم‌های تک‌پردازنده‌ای، اغلب در داخل خود سیستم عامل برای مدیریت نواحی بحرانی همچون اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal، تکنیک مفیدی است، اما برای مدیریت نواحی بحرانی فرآیندهای کاربر، به دلیل فراموش‌کار بودن کاربر، تکنیک مناسبی نیست.

توجه: در سیستم‌های چندپردازنده‌ای، غیرفعال کردن وقفه‌ها، فقط در پردازنده‌ای اثر دارد که دستورالعمل از کار انداختن وقفه را اجرا می‌کند. پس پردازنده‌های دیگر به کارشان ادامه می‌دهند. از آنجایی که ممکن است بیش از یک فرآیند در هر لحظه در حال اجرا باشد، ممکن است فرآیندهای دیگر که روی پردازنده‌های دیگر در حال اجرا هستند نیز وارد ناحیه‌ی بحرانی شوند. به

khalilifar.ir

عبارت دیگر، در سیستم‌های چندپردازنده‌ای، این ذات توازی است که باعث ورود همزمان فرآیندها به ناحیه‌ی بحرانی می‌شود و نه وقوع وقفه، که با جلوگیری از آن، بخواهیم مشکل را حل کنیم. پس در سیستم‌های چند پردازنده‌ای ممکن است شرط انحصار متقابل برقرار نباشد.

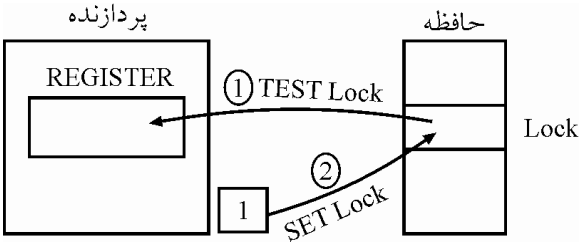
ساختار کلی دستورالعمل TSL در سیستم **تک و چند پردازنده‌ای**، به صورت زیر می‌باشد:

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

بسیاری از پردازنده‌ها، دستورالعمل دو بخشی اما اتمیک خاصی دارند به نام TSL (TEST and SET Lock)، بدین نحو که این دستورالعمل به شکل تجزیه ناپذیر، عملیات زیر را انجام می‌دهد:

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

به شکل زیر توجه کنید:



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

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

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

khalilifar.ir

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

ساختار کلی برقراری شرط انحصار متقابل توسط توابع wait و signal، به صورت زیر می‌باشد:

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

توجه: برای اینکه توابع wait و signal درب مناسبی برای برقراری شرط انحصار متقابل ناحیه بحرانی که در بر دارند باشند، خود باید درب کاملی باشند، یعنی باید بشوند که بتوانند بشوند، پس اول باید شرط انحصار متقابل برای خود توابع wait و signal برقرار باشد، تا بتوانند شرط انحصار متقابل را برای ناحیه بحرانی که در بر دارند را برقرار کنند.

ساختار کلی اتمیک و تجزیه‌ناپذیرسازی توابع wait و signal در سیستم **تک و چند پردازنده‌ای** به کمک دستورالعمل TSL به صورت زیر می‌باشد:

khalilifar.ir

```
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Wait (semaphore)
    {
s.count = s.count – 1; / * critical_section ()*/
if (s.count< 0)
{
Add this process to s.queue;
block ();
  }
}
exit_section
MOVE LOCK, 0
RET
critical_section ();
enter_section
TSL REGISTER, LOCK
CMP REGISTER, 0
JNE enter_section
RET
Signal (semaphore s)
{
s.count = s.count + 1; / * critical_section ()*/
    if (s.count< = 0)
    {
     Remove a process from queue;
     Wake up ();
      }
}
exit_section
MOVE LOCK, 0
RET
remainder_section ();
```

khalilifar.ir

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

توجه: این راه‌حل، فقط در پردازنده‌هایی قابل اجراست که دستورالعمل TSL را پشتیبانی می‌کنند.

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

۱۲- گزینه ( ) صحیح است.

	Thread 1		Thread 2
(1)	if $(a<b)$ then	(4)	$b=10$
(2)	$c=b-a$	(5)	$c=-3$
(3)	$c=b+a$		

مطابق فرض سؤال مقادیر اولیه a برابر 4، b برابر صفر و c برابر صفر است. بنابراین ترتیب‌های زیر را داریم:

ترتیب اجرا	مقدار c در انتها	گزینه‌ی نادرست
$(4)\rightarrow(5)\rightarrow(1)\rightarrow(2)$	6	-
$(1)\rightarrow(3)\rightarrow(4)\rightarrow(5)$	-3	-
$(1)\rightarrow(4)\rightarrow(5)\rightarrow(3)$	14	1 و 2
$(4)\rightarrow(1)\rightarrow(5)\rightarrow(2)$	6	-

بنابراین، گزینه‌ی سوم یا چهارم درست است، سایر مقادیر قابل تولید نیست!

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه چهارم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

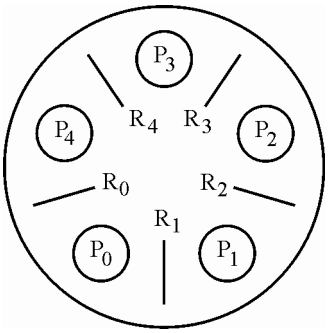
۱۳- گزینه (۲) صحیح است.

حل مسأله فیلسوفان خورنده توسط سمافور

5 فیلسوف زندگی خود را صرف فکر کردن و خوردن کرده‌اند. آن‌ها دور یک میز دایره‌ای با 5 بشقاب و 5 عدد چنگال نشسته‌اند. هر فیلسوف برای غذا خوردن حتماً باید دو چنگال در دست داشته باشد. بین هر جفت بشقاب روی میز، یک چنگال وجود دارد. هنگامی که فیلسوفی در حال فکر کردن است با بقیه هیچ ارتباطی ندارد. هر از گاهی فیلسوف احساس گرسنگی کرده و سعی می‌کند، چنگال‌های سمت راست و چپش را یکی یکی و با هر ترتیب ممکن بردارد، اگر موفق به برداشتن هر دو چنگال شود برای مدتی غذا خورده و دوباره چنگال‌ها را پایین گذاشته و به فکر

khalilifar.ir

کردن ادامه می‌دهد. فیلسوف مجاز است که در هر بار فقط یک چنگال را بردارد و همچنین نمی‌تواند چنگالی که دست فیلسوف دیگری است را به زور بگیرد.



راه حل اول

از آنجایی که هر فیلسوفی که در حال خوردن است، نمی‌تواند چنگال‌های در اختیارش را به فیلسوف مجاور بدهد، لازم است برای هر چنگال شرط انحصار متقابل برقرار باشد. لذا برای هر چنگال یک شمارنده سمافور تعریف می‌شود. مسأله فیلسوفان خورنده در راه حل اول بر دو نوع (1) چپگرد و (2) راستگرد می‌باشد.

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

ساختار کلی این راه حل به صورت زیر می‌باشد:

توجه: قطعه کد زیر را شبه کد فرض کنید.

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

khalilifar.ir

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

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

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

شرایط بن‌بست را به یاد آورید:

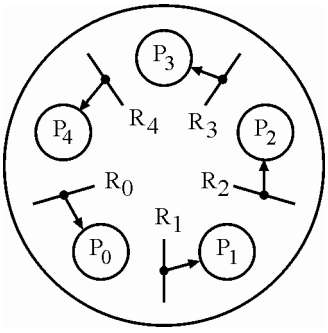
۱- انحصار متقابل (برقرار است، توسط تعریف شمارنده سمافور برای هر چنگال (منبع بحرانی))

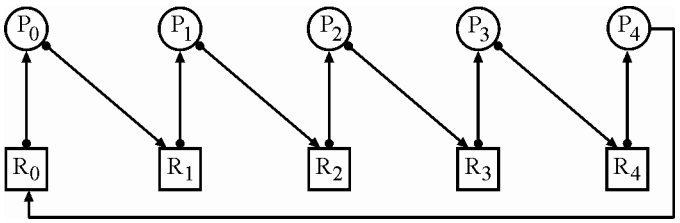
۲- انحصاری بودن (برقرار است، نمی‌توان چنگال (منبع بحرانی) را به زور پس گرفت).

۳- نگهداری و انتظار (برقرار است.)

۴- سیکل انتظار چرخشی (برقرار است.)

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





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

ساختار کلی این راه حل به صورت زیر می‌باشد:

khalilifar.ir

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

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

توجه: حال اگر در راه حل چپگرد حداقل یک فیلسوف راست دست و یا در راه حل راستگرد حداقل یک فیلسوف چپ دست وجود داشته باشد، شرط سیکل نقض می‌گردد، بنابراین هیچ‌گاه بن‌بست رخ نمی‌دهد.

توجه: مطابق فرض سوال، با وجود حداقل یک فیلسوف چپ دست و یک فیلسوف راست دست، بنابراین هر نوع راه حلی، اعم از چپگرد یا راستگرد پیش رود، از آن‌جا که حداقل یک فیلسوف در جهت مقابل قرار دارد، بنابراین شرایط نقض شرط سیکل، ایجاد می‌گردد، که همین عامل باعث می‌شود، هیچ‌گاه بن‌بست رخ ندهد.

۱۴- گزینه (۲) صحیح است.

راه حل سمافور در سال 1965 توسط Dijkstra پیشنهاد شده است.

ساختار کلی این راه حل به صورت زیر می‌باشد:

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

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

توجه: دو تابع wait (mutex) و signal (mutex)، باید به صورت اتمیک (تجریه ناپذیر) انجام گیرند. اتمیک بودن، تضمین می‌کند که از لحظه‌ای که یک عملیات بر روی شمارنده سمافور شروع

khalilifar.ir

می‌شود، هیچ فرآیند دیگری نتواند به سمافور دسترسی پیدا کند تا زمانی که آن عملیات به پایان برسد. اتمیک بودن این عملیات برای حل مسایل همگام‌سازی و کنترل شرایط رقابتی و به تبع برقراری شرط انحصار متقابل کاملاً لازم و ضروری است.

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

راه حل سمافور بر دو دسته کلی (1) سمافور عمومی و (2) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور دودویی می‌پردازیم:

سمافور دودویی

تنها تفاوت سمافور دودویی و سمافور عمومی در نحوه‌ی تعریف توابع wait و singal است.

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

ساختار این تابع به صورت زیر است:

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

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

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

khalilifar.ir

توجه: در سمافور دودویی، شمارنده سمافور، هیچگاه منفی نمی‌شود و در هر شرایطی فقط می‌تواند دو مقدار باینری یا دودویی 0 یا 1 را داشته باشد، برای اثبات، مجدداً به تعاریف توابع wait و signal در سمافور دودویی دقت کنید.

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

توجه: در این سوال تابع release معادل تابع signal در نظر گرفته شده است.

با توجه به مقادیر شمارنده‌های سمافور مطرح شده در سوال، $S_0 = 1$، $S_1 = 0$ و $S_2 = 0$، فرآیند P_0 ابتدا اجرا می‌شود.

توجه: فرآیندهای P_1 و P_2 فاقد حلقه هستند، و توسط شرایطی که فرآیند P_0 برای فرآیندهای P_1 و P_2 فراهم می‌کند، فرآیندهای P_1 و P_2 فقط و فقط می‌توانند **یکبار** اجرا شوند و تمام شوند.

سناریوی اول: (دو بار چاپ '0')

ابتدا فرآیند P_0 اجرا می‌شود و روی شمارنده سمافور S_0، عمل $wait(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $release(S_1)$ و $release(S_2)$ مقدار شمارنده سمافورهای S_1 و S_2 را برابر یک می‌کند.

حال فرآیندهای P_1 و P_2 می‌توانند به هر ترتیبی (اول P_1 بعد P_2 و یا اول P_2 بعد P_1)، با توجه به مقدار شمارنده سمافورهای S_1 و S_2، اجرا شوند. ولی از آنجا که شمارنده سمافور دودویی است، در انتهای اجرای دو فرآیند P_1 و P_2، حداکثر مقدار شمارنده سمافور دودویی S_0 به واسطه دو بار اجرای دستور $release(s_0)$ برابر یک خواهد بود. اگر سمافور S عمومی بود، مقدار شمارنده سمافور S_0 برابر 2 می‌بود، اما فرض سوال سمافور دودویی است!

در این لحظه فرآیندهای P_1 و P_2 تمام شده‌اند و دیگر اجرا نمی‌شوند، از آنجا که در انتهای کار فرآیندهای P_1 و P_2 مقدار شمارنده سمافور S_0، برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یک بار دیگر اجرا شود و دومین '0' را نیز چاپ کند. در ادامه فرآیند P_0، با اجرای دستور $wait(S_0)$، با سرنوشتی که دارد، برای همیشه می‌خوابد.

دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ دومین '0'، تغییری در روند اجرای کار ندارند، چون فرآیندهای P_1 و P_2 دیگر نیستند و تمام شده‌اند.

khalilifar.ir

سناریوی دوم: (سه بار چاپ '0')

ابتدا فرآیند P_0 اجرا می‌شود و روی شمارنده سمافور S_0، عمل $wait(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $release(S_1)$ و $release(S_2)$ مقدار شمارنده سمافورهای S_1 و S_2 را برابر یک می‌کند.

حال اگر فرآیند P_1، با توجه به مقدار شمارنده سمافور S_1 اجرا شود، در انتهای اجرای فرآیند P_1، مقدار شمارنده سمافور دودویی S_0 به واسطه $release(S_0)$ برابر یک خواهد شد. در این لحظه فرآیند P_1 تمام شده است و دیگر اجرا نمی‌شود. از آنجا که در انتهای کار فرآیند P_1، مقدار شمارنده سمافور S_0 برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یکبار دیگر اجرا شود و دومین '0' را چاپ کند. دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ دومین '0'، تغییری در روند اجرای کار ندارد، چون فرآیند P_1، دیگر نیست و تمام شده است و برای فرآیند P_2 نیز مقدار شمارنده سمافور S_2 قبلاً و بعد از چاپ اولین '0' برابر یک شده است و چون شمارنده سمافور دودویی است، انجام دستور $release(S_2)$ بعد از چاپ دومین '0' تأثیری در مقدار شمارنده سمافور S_2 نخواهد داشت و در همان مقدار یک باقی خواهد ماند.

توجه: دقت کنید که حداکثر مقدار شمارنده سمافور دودویی برابر یک می‌باشد.

حال اگر فرآیند P_2، با توجه به مقدار شمارنده سمافور S_2، اجرا شود، در انتهای اجرای فرآیند P_2، مقدار شمارنده سمافور دودویی S_0 به واسطه $release(S_0)$ برابر یک خواهد شد. در این لحظه فرآیند P_2 تمام شده است و دیگر اجرا نمی‌شود. از آنجا که در انتهای کار فرآیند P_2 مقدار شمارنده سمافور S_0 برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یکبار دیگر اجرا شود و سومین '0' را چاپ کند. دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ سومین '0'، تغییری در روند اجرای کار ندارد، چون فرآیندهای P_1 و P_2 تمام شده‌اند و دیگر اجرا نمی‌شوند.

در ادامه، فرآیند P_0، با اجرای دستور $wait(S_0)$، با سرنوشتی که دارد، برای همیشه می‌خوابد.

بنابراین گزینه دوم درست خواهد بود.

۱۵- گزینه (۴) صحیح است.

راه‌حل سمافور در سال ۱۹۶۵ توسط Dijkstra پیشنهاد شده است.

ساختار کلی این راه‌حل به صورت زیر می‌باشد:

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

khalilifar.ir

توجه: mutex از عبارت Mutual Exclusion گرفته شده است.

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

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

راه‌حل سمافور بر دو دسته‌ی کلی (۱) سمافور عمومی و (۲) سمافور دودویی می‌باشد، که در ادامه به بررسی سمافور عمومی می‌پردازیم:

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
Int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

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

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله‌ی انتظار مشغول و حل مسأله‌ی اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این‌گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

khalilifar.ir

بر روی سمافور عمومی s دو تابع wait(s) و signal(s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
add  this process to s.queue;
block ();
  }
}
```

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

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

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

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

ساختار این تابع به صورت زیر است:

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

khalilifar.ir

توجه: راه‌حل سمافور و تابع اتمیک signal باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه سازی کرد.

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

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

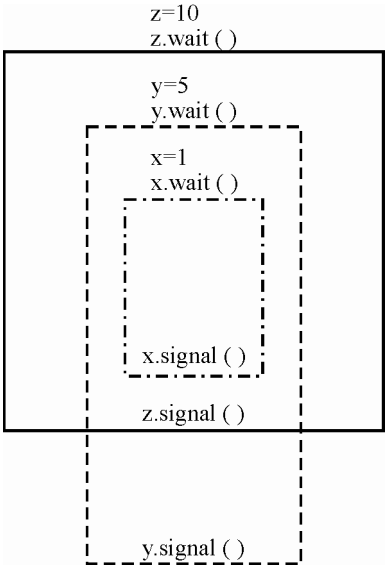
در صورت سوال مطرح شده است که سه سمافور با مقدار اولیه x=1، y=5 و z=10 در نظر گرفته شود. همچنین گفته شده است که قطعه کد زیر توسط 20 پردازه (process) اجرا می‌شود.

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

در ادامه خواسته سوال این است که، حداکثر طول صفی که برای سمافور y تشکیل می‌شود، چقدر است؟

شکل زیر را در نظر بگیرید:

khalilifar.ir



دقت کنید که در صورت سوال مطرح شده است که قطعه کد فوق توسط 20 پردازه (process) اجرا می‌شود. پس می‌بایست قطعه کد فوق توسط 20 فرآیند اجرا گردد و از آن عبور کنند.

20 فرآیند P_1، P_2، P_3، P_4، P_5، P_6، P_7، P_8، P_9، P_{10}، P_{11}، P_{12}، P_{13} ، P_{14}، P_{15}، P_{16}، P_{17}، P_{18}، P_{19} و P_{20} را در نظر بگیرید. از این تعداد با توجه به شمارنده سمافور z که برابر مقدار 10 است، ابتدا فرآیندهای P_1، P_2، P_3، P_4، P_5، P_6، P_7، P_8، P_9 و P_{10} به ترتیب اجرا می‌شوند و از سمافور z عبور می‌کنند. و فرآیندهای P_{11}، P_{12}، P_{13} ، P_{14}، P_{15}، P_{16}، P_{17}، P_{18}، P_{19} و P_{20} پشت سمافور z در صف سمافور z آرام می‌گیرند و می‌خوابند. در ادامه از بین فرآیندهای عبور کرده از سمافور z یعنی فرآیندهای P_1، P_2، P_3، P_4، P_5، P_6، P_7، P_8، P_9 و P_{10} با توجه به شمارنده سمافور y که برابر مقدار 5 است، فقط فرآیندهای P_1، P_2، P_3، P_4 و P_5 این شانس را پیدا می‌کنند که از سمافورy عبور کنند و عبور می‌کنند، و فرآیندهای P_6، P_7، P_8، P_9 و P_{10} پشت سمافور y در صف سمافور y آرام می‌گیرند و می‌خوابند. پس درحال حاضر 5 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار5 است. در ادامه از بین فرآیندهای عبور کرده از سمافور y یعنی فرآیندهای P_1، P_2، P_3، P_4 وP_5 با توجه به شمارنده سمافور x که برابر مقدار 1 است، فقط فرآیند P_1 این شانس را پیدا می‌کند که از سمافورx عبور کند و عبور می‌کند، و فرآیندهای P_2، P_3، P_4 وP_5 پشت سمافور x در صف سمافور x آرام می‌گیرند و می‌خوابند. پس درحال حاضر 4 فرآیند در صف سمافور x به خواب رفته‌اند، بنابراین طول صفی که برای سمافور x تشکیل شده است، هم اکنون برابر مقدار4 است. در ادامه فرآیند P_1 پس از عبور از ناحیه سمافور x تابع x.signal() را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_2 می‌گردد، در این لحظه فرآیند P_2 در خط بعد از تابع x.wait() و قبل از تابع x.signal() قرار می‌گیرد، همچنین در ادامه فرآیند P_1 پس از عبور از ناحیه سمافور z تابع z.signal() را اجرا

khalilifar.ir

می‌کند، این امر منجر به بیدار شدن فرآیند P_{11} می‌گردد، در این لحظه فرآیند P_{11} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{11} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{11} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد. پس درحال حاضر 6 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 6 است. در ادامه فرآیند P_2 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_3 می‌گردد، در این لحظه فرآیند P_3 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_2 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{12} می‌گردد، در این لحظه فرآیند P_{12} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{12} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{12} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد.

پس درحال حاضر 7 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 7 است. در ادامه فرآیند P_3 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_4 می‌گردد، در این لحظه فرآیند P_4 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_3 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{13} می‌گردد، در این لحظه فرآیند P_{13} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{13} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{13} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد.

پس درحال حاضر 8 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 8 است. در ادامه فرآیند P_4 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_5 می‌گردد، در این لحظه فرآیند P_5 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_4 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{14} می‌گردد، در این لحظه فرآیند P_{14} در خط بعد از تابع ()z.wait و قبل از تابع ()y.wait قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{14} اجرا گردد و تابع ()y.wait را اجرا کند، فرآیند P_{14} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد.

پس درحال حاضر 9 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 9 است. در ادامه فرآیند P_5 پس از عبور از ناحیه سمافور x تابع ()x.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_6 می‌گردد، در این لحظه فرآیند P_6 در خط بعد از تابع ()x.wait و قبل از تابع ()x.signal قرار می‌گیرد، همچنین در ادامه فرآیند P_5 پس از عبور از ناحیه سمافور z تابع ()z.signal را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{15} می‌گردد، در این لحظه فرآیند P_{15} در خط بعد از تابع ()z.wait

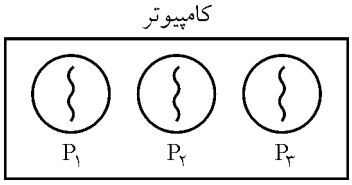
khalilifar.ir

و قبل از تابع y.wait() قرار می‌گیرد، حال اگر در همین لحظه فرآیند P_{15} اجرا گردد و تابع y.wait() را اجرا کند، فرآیند P_{15} نیز پشت سمافور y در صف سمافور y آرام می‌گیرد و می‌خوابد.
پس درحال حاضر 10 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 10 است. این حداکثر طول صفی است که می‌تواند برای سمافورy ایجاد شود. در ادامه این سناریو وجود دارد که فرآیند P_1 پس از عبور از ناحیه سمافور y تابع y.signal() را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_6 می‌گردد، در این لحظه فرآیند P_6 در خط بعد از تابع y.wait() و قبل از تابع x.wait() قرار می‌گیرد، درحال حاضر 9 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار 9 است. این روند تا اتمام ملاقات هر سه ناحیه x، y و z برای همه فرآیندها ادامه پیدا می‌کند.

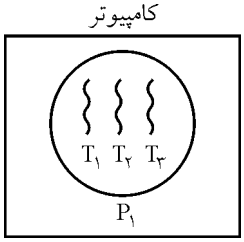
۱۶- گزینه (۲) صحیح است.

نخ یا ریسمان (Thread)

در سیستم‌های قدیمی‌تر، به ازای هر فرآیند یک رشته نخ یا رشته اجرایی و به تبع یک شمارنده برنامه (PC) وجود داشت اما در سیستم عامل‌های امروزی به ازای هر فرآیند می‌توان چند نخ یا رشته اجرایی داشت.
شکل زیر سه فرآیند معمولی را نشان می‌دهد که هریک برای خودشان یک رشته اجرایی و یک حافظه مختص به خود را دارند.

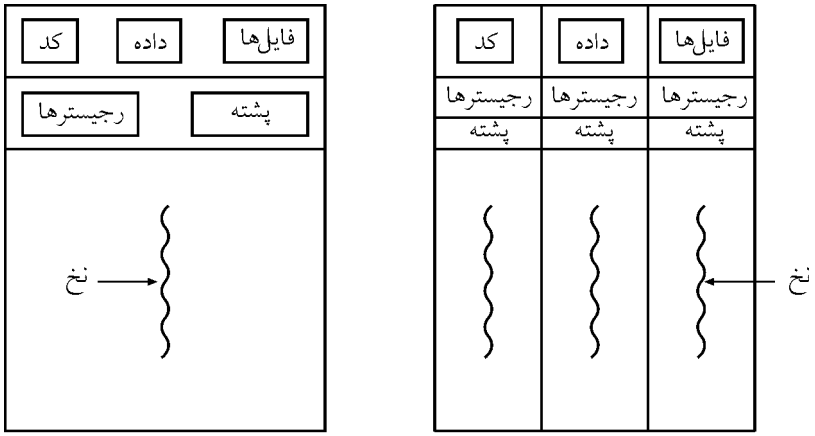


ولی در شکل زیر یک فرآیند، سه رشته اجرایی دارد که هر یک رجیستر، پشته و شمارنده برنامه (PC) مجزای خود را دارند و مانند فرآیندها می‌توانند همروند (در سیستم‌های تک‌پردازنده‌ای) و موازی (در سیستم‌های چندپردازنده‌ای یا چندهسته‌ای) اجرا شوند.



khalilifar.ir

توجه: نخ‌های همتا که در یک فرآیند قرار دارند، از کد، داده و منابع مشترک استفاده می‌کنند اما هر نخ، شمارنده برنامه، مجموعه رجیستر و فضای پشته جداگانه‌ای در اختیار دارد. در واقع هر نخ، TCB مجزایی دارد.



توجه: از آن‌جا که نخ‌های همتا در یک فرآیند قرار داشته و اشتراکات زیادی با هم دارند، عمل تعویض متن بین آنها به راحتی و با هزینه کمتری صورت می‌گیرد، در واقع TCB مربوط به نخ‌ها، محتوی کمتری نسبت به PCB فرآیندها دارد، مثلاً لیست فایل‌های باز مربوط به فرآیندها است، بنابراین این لیست به هنگام تعویض متن فرآیندها باید داخل PCB مربوط به فرآیند ذخیره گردد، در حالی که به هنگام تعویض متن بین نخ‌ها نیازی به ذخیره‌سازی لیست فایل‌های باز مربوط به یک فرآیند در TCB یک نخ نیست. بنابراین تعویض متن بین نخ‌ها نسبت به فرآیندها ارزان‌تر است.

توجه: گاهی از نخ به عنوان Light Weight Process (فرآیند سبک وزن) نیز یاد می‌کنند و به کل یک فرآیند، Heavy Weight Process (فرآیند سنگین وزن) نیز می‌گویند.

توجه: نخ‌ها هم مانند فرآیندها می‌توانند حالت‌های مختلفی را تجربه کنند مانند آماده، در حال اجرا یا منتظر. در واقع پردازنده می‌تواند بین نخ‌ها به اشتراک گذاشته شود.

چند نخی در زبان #C

#C پیاده‌سازی چند نخی را پشتیبانی می‌کند. در زبان #C، هر برنامه به طور پیش فرض از یک نخ تشکیل شده است و درصورت ایجاد نخ‌های دیگر، مفهوم چندنخی پیاده‌سازی می‌گردد.

توجه: نخ اول به صورت پیش‌فرض وجود دارد و برنامه با نخ اول شروع به اجرا می‌کند.

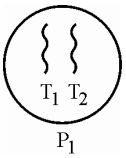
مثال: در قطعه کد زیر نخ T1 به طور پیش‌فرض وجود دارد و نخ T2 ایجاد می‌گردد.

khalilifar.ir

```
static void main ()
    {
Thread     T2=new     Thread (Go);
   →   T2.Start( )                    ↓
   →   Go ( );                  در این بخش نخ T2 ایجاد
        }                       می‌گردد و فعالیت Go
static void Go( )               داخل این نخ قرار می‌گیرد.
{
for (int i = 0 ; i<5 ; i++)
        console.write("*");
}
```

نخ T2 اجرا می‌گردد.

فعالیت Go داخل نخ T1 قرار داده شده است. می‌شود فعالیت دیگری تعریف شود و در این بخش فراخوانی شود.



توجه: فعالیت Go داخل نخ T1 قرار دارد، اما فعالیت Go در نخ T2 هم قرار داده شده است. در دو نخ T1 و T2 که فعالیت Go داخل آن قرار دارد، متغیر محلی i در داخل پشته مربوط به هر نخ ایجاد می‌گردد.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

****

چاپ 10 عدد ستاره به دلیل اجرای همروند (سیستم تک‌پردازنده‌ای) یا موازی (سیستم چندپردازنده‌ای یا چند هسته‌ای) دو نخ T1 و T2 است!

مثال: در قطعه کد زیر نخ T1 به طور پیش‌فرض وجود دارد و نخ T2 ایجاد می‌گردد.

```
static void main ()
    {
  Thread     T2=new     Thread (func);
   →   T2.Start( ) ;                  ↓
   →   console.write("X");      در این بخش نخ T2 ایجاد
        }                       می‌گردد و فعالیت func
static void func( )             داخل این نخ قرار می‌گیرد.
{
console.write("Y");
}
```

نخ T2 اجرا می‌گردد.

فعالیت چاپ "X" داخل نخ T1 قرار داده شده است.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

XY

khalilifar.ir

مزایای فرآیندهای چندنخی

۱- ساختار بسیاری از برنامه‌های کاربری ذاتاً از بخش‌های کاملاً مستقل تشکیل می‌شوند که جدا نکردن آن‌ها باعث پیچیدگی بالا و کاهش خوانایی در برنامه می‌گردد. مهندسی نرم‌افزار نیز بر ساخت برنامه‌های کاربردی توسط پیمانه‌های مستقل نیز تأکید دارد. برای مثال، خطاست اگر بیندیشید که برنامه شبیه‌سازی 11 بازیکن یک تیم فوتبال در یک نخ قرار گیرد. به عنوان مثالی دیگر یک برنامه واژه‌پرداز می‌تواند از نخ‌های مستقلی مانند کنترل املا و گرامر، صفحه‌آرایی، مدیریت ورودی‌های کاربر و غیره تشکیل شده باشد.

۲- در فرآیند تک‌نخی، هرگاه فراخوان سیستمی مسدود کننده‌ای اجرا شود، کل فرآیند مسدود می‌گردد. در حالی‌که در فرآیندهای چندنخی در صورتی که سیستم عامل زمان‌بندی چند نخی را پشتیبانی کند، فقط نخی که فراخوان سیستمی مسدودکننده دارد، مسدود می‌گردد و مابقی نخ‌های یک فرآیند می‌توانند پس از در اختیار گرفتن پردازنده، اجرا گردند. فرآیند تک‌نخی مانند قانونی می‌باشد که اگر یک نفر در خانواده خطا کند، همه خانواده محکوم می‌گردند و فرآیند چندنخی مانند قانونی می‌باشد که اگر یک نفر در خانواده خطا کند، فقط همان یک نفر محکوم می‌گردد و بقیه خانواده می‌توانند به زندگی طبیعی خود ادامه دهند.

۳- ایجاد هم‌روندی (در سیستم تک‌پردازنده‌ای) و توازی (در سیستم چندپردازنده‌ای یا چند هسته‌ای) در نخ‌های یک فرآیند و فرآیندهای دیگر.

مثال: کاربرد چندنخی در فرآیند سمت سرویس‌دهنده.

در این مدل، فرآیند سمت سرویس‌دهنده از چندین نخ جهت پاسخ به درخواست‌های کاربر یعنی سرویس‌گیرنده تشکیل شده است. پاسخ هر کاربر می‌تواند توسط یک نخ از سمت سرویس‌دهنده داده شود. چنانچه نخی در فرآیند سرویس‌دهنده جهت تبادل داده از روی دیسک به سمت سرویس‌گیرنده مسدود گردد، نخ‌های دیگر فرآیند سرویس‌دهنده می‌توانند به درخواست‌های دیگر، پاسخ دهند. زیرا کارکرد آن‌ها وابسته به نخ مسدود شده نیست.

توجه: شاید بگویید به جای قرار دادن کارهای مختلف یک سرویس‌دهنده در داخل نخ‌های یک فرآیند، می‌شد هریک از کارها را در داخل یک فرآیند قرار داد و چند فرآیندی را در مقابل چندنخی ابداع کرد. اما به دلایل زیر استفاده از چندنخی معقولانه‌تر به‌نظر می‌رسد:

- هزینه زمانی ایجاد (بارگذاری TCB) و پایان دادن (ذخیره‌سازی TCB) یک نخ در یک فرآیند به مراتب کمتر از ایجاد (بارگذاری PCB) و پایان دادن (ذخیره‌سازی PCB) یک فرآیند است. نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند، در حالی که فرآیندها، منابع مختص به خود را در اختیار می‌گیرند.
- نخ‌های همتا در یک فرآیند، اشتراکات زیادی باهم دارند، بنابراین عمل تعویض متن بین آنها با هزینه کمتری انجام می‌گردد. در حالی که تعویض متن بین فرآیندها به دلیل عدم اشتراکات با هزینه بیشتری انجام می‌گردد.

khalilifar.ir

توجه: وجه اشتراک نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، فایل‌های باز و وجه اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد.

مدیریت نخ‌های هم‌روند(Thread Safety)

در اغلب سیستم‌های امروزی، تعدادی از فرآیندها یا نخ‌ها به صورت همروند بر روی یک پردازنده و یا به صورت موازی بر روی چندین پردازنده یا چندین هسته اجرا می‌شوند. در سیستم‌های چندبرنامگی، چندپردازنده‌ای و چندهسته‌ای، هم‌روندی و توازی فرآیندها و نخ‌ها، یک پدیده‌ی عادی به شمار می‌آید.

فرآیندها و نخ‌های هم‌روند و همکار، به ارتباط با یکدیگر نیاز دارند. آن‌ها برای دستیابی به یک هدف مشترک، نیازمند همکاری، هماهنگی، تبادل داده و استفاده از داده‌ها و سایر منابع مشترک هستند. بنابراین مدیریت اجرای همروند چند فرآیند یا چند نخ بر روی یک پردازنده و اجرای موازی چند فرآیند یا چند نخ بر روی چندین پردازنده یا چندین هسته، حائز اهمیت فراوان است. این مدیریت باید به گونه‌ای باشد که اجرای یک فرآیند یا نخ آسیبی به اجرای فرآیندها یا نخ‌های همکار و هم‌روند دیگر نرساند.

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

الف) تبادل داده

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

- **حافظه مشترک**
- **فایل مشترک**

توجه: تبادل داده برای نخ‌های همتا و هم‌خانواده درون یک فرآیند ساده می‌باشد، زیرا نخ‌ها یک فضای آدرس مشترک دارند. اما نخ‌های متعلق به فرآیندهای جداگانه یعنی غیرهمتا که در فضای آدرس متفاوت قرار دارند، در صورت نیاز به ارتباط باید از مکانیسم‌های ارتباط فرآیندها، استفاده کنند.

khalilifar.ir

ناحیه بحرانی

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

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

منبع بحرانی

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

ب) شرایط رقابتی (مسابقه)

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

مثال: شرایط رقابتی

دو نخ هم‌روند Thread1 و Thread2 در یک سیستم اشتراک زمانی که از متغیر مشترک سراسری S، در بخشی از کد خود استفاده می‌کنند در نظر بگیرید، بعد از اجرای کامل دو نخ، مقدار نهایی S چه خواهد شد؟ (مقدار اولیه متغیر سراسری S برابر صفر است).

Thread1 :	Thread2 :
S=S+1	S=S-1

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

Thread1:	Thread2:
MOVE REGISTER, S INC REGISTER MOVE S, REGISTER	MOVE REGISTER, S DEC REGISTER MOVE S, REGISTER

حالت اول:

فرض کنید Thread1 کامل اجرا شود و سپس Thread2 کامل اجرا شود (یا برعکس).

Thread1:	Thread2:
① REGISTER ← 0 ② REGISTER ← 1 ③ S ← 1	④ REGISTER ← 1 ⑤ REGISTER ← 0 ⑥ S ← 0

khalilifar.ir

بنابراین مقدار نهایی متغیر S برابر صفر خواهد بود.(S=0)

حالت دوم:

فرض کنید ابتدا ترتیب زیر اجرا شود.

Thread1:

① REGISTER ← 0
INC REGISTER
MOVE S, REGISTER

Thread2:

② REGISTER ← 0
③ REGISTER ← -1
④ S ←-1

سپس پردازنده در اثر تعویض متن به نخ Thread1 باز گردد.

توجه: هر نخ محتویات رجیسترهای خودش را قبل از تعویض متن در TCB ذخیره می‌کند.

بنابراین در این لحظه مقدار رجیستر در TCB فرآیند Thread1، برابر صفر است.

حال در ادامه دستورات باقی مانده نخ Thread1 اجرا می‌شوند.

Thread1:

.
.
.

⑤ REGISTER ←1
⑥ S ←1

بنابراین مقدار نهایی متغیر S برابر مثبت یک خواهد بود.(S= +1)

حالت سوم:

فرض کنید ابتدا ترتیب زیر اجرا شود.

Thread1:

② REGISTER ← 0
③ REGISTER ← 1
④ S ←1

Thread2:

① REGISTER ← 0
DEC REGISTER
MOVE S, REGISTER

سپس پردازنده در اثر تعویض متن به نخ Thread2 باز گردد.

توجه: هر نخ محتویات رجیسترهای خودش را قبل از تعویض متن در TCB ذخیره می‌کند.

بنابراین در این لحظه مقدار رجیستر در TCB نخ Thread2، برابر صفر است.

حال در ادامه دستورات باقی مانده نخ Thread2 اجرا می‌شوند.

Thread2:

.
.
.

⑤ REGISTER ← -1
⑥ S ←-1

khalilifar.ir

بنابراین مقدار نهایی متغیر S برابر منفی یک خواهد بود. (S= -1)

توجه: اما مشکل نهایی اینجاست که مقدار نهایی متغیر S، به نحوه‌ی **تعویض متن پردازنده** یا به عبارتی، به ترتیب اجرای دستورالعمل‌ها، بستگی دارد و می‌تواند مقادیر 0، 1، 1- را داشته باشد. این پدیده، حاصل رقابت بر سر تصاحب یک عامل مشترک (متغیر مشترک S) است.

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

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

۱- شرط انحصار متقابل

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

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

khalilifar.ir

```
Thread ()  {
while (TRUE)  {
entry_section () ;  // تلاش برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی
critical_ selection (); // ناحیه‌ی بحرانی
exit_ selection () ; // اعلام خروج از ناحیه‌ی بحرانی
remainder_ section () ; // ناحیه‌ی باقی مانده
  }
}
```

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

۲- شرط پیشرفت

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

۳- شرط انتظار محدود

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

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

قحطی (گرسنگی)

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

khalilifar.ir

بن‌بست

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

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

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

ج) همگام سازی

گاهی اوقات خواسته‌ی ما این است که نخ‌ها به یک ترتیب مشخص و از قبل تعیین شده بر روی یک عامل مشترک (داده مشترک) عملیاتی را انجام دهند. واضح است که در این حالت تبادل داده به روش استفاده از حافظه مشترک است. همچنین از آنجا که پای یک عامل مشترک در میان است، پس رقابت بر سر تصاحب این عامل مشترک هم در میان است. بنابراین راه‌حل همگام‌سازی باید به گونه‌ای باشد که نخ‌های همکار دچار شرایط رقابتی نشوند. به بیان دیگر باید مانع اثر مخرب **تعویض متن پردازنده** بین نخ‌های همکار که عامل ایجاد شرایط رقابتی است، شد.

برای مثال اگر نخ Thread1 داده‌ای را باید تولید کند و سپس نخ Thread2 آن را مصرف کند، نخ Thread2 باید منتظر باشد تا نخ Thread1، داده‌ی مورد نیازش را آماده کند و بعد شروع به مصرف نماید.

مثال: همگام سازی دو نخ

دو نخ Thread1 وThread2 را در نظر بگیرید که در یک سیستم تک‌پردازنده‌ای اشتراک زمانی به صورت هم‌روند اجرا می‌شوند. فرض کنید نخ Thread1 در بخشی از کد خود مقدار متغیر x را می‌خواند و نخ Thread2 نیز در بخشی از برنامه‌اش باید مقدار متغیر x خوانده شده توسط Thread1 را چاپ کند. هدف مسأله، نوشتن این دو نخ به صورتی است که Thread2 در چاپ متغیر x بر Thread1 پیشی نگیرد و اگر زودتر به بخش چاپ متغیر x رسید و هنوز مقدار متغیر x خوانده نشده بود، صبر کند تا Thread1 متغیر x را مقداردهی کند. بنابراین باید یک مسأله‌ی همگام‌سازی حل شود. واضح است که در این حالت تبادل داده به روش استفاده از حافظه مشترک است. یکی از راه‌حل‌های مناسب برای همگام‌سازی نخ‌های همکار، استفاده از کلاس سمافور و یا کلاس مانیتور است.

راه‌حل‌های کنترل شرایط رقابتی نخ‌های همکار

۱- راه‌حل‌های نرم‌افزاری (برعهده‌ی برنامه‌نویس)

۲- راه‌حل‌های سخت‌افزاری

۳- راه‌حل کلاس سمافور

khalilifar.ir

۴- راه‌حل کلاس مانیتور

راه‌حل‌های همگام‌سازی نخ‌های همکار

۱- راه‌حل کلاس سمافور

۲- راه‌حل کلاس مانیتور

توجه: راه‌حل‌های فوق، در سیستم‌های تک‌پردازنده‌ای، چندپردازنده‌ای و چندهسته‌ای با حافظـه‌ی اشتراکی قابل استفاده‌اند.

خواسته سوال این است که چه وقت یک تابع در سطح نخ **ایمن** است؟

جهت پاسخ به خواسته سوال شبه کد زیر را در نظر بگیرید:

```
#include <pthread.h>
  static void main ( )
     {
      Thread T2=new Thread (increment_counter);
         T2.Start ( )
         increment_counter ( );
     }
int increment_counter ( )
{
static int counter = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//only allow one thread to increment at a time
pthread_mutex_lock (&mutex);
counter= counter+1;
//store value before any other threads increment it further
int result = counter;
pthread_mutex_unlock (&mutex);
return result;
}
```

توجـــه: فعالیـــت increment_counter داخـــل نـــخ Thread1 قـــرار دارد، همچنـــین فعالیـــت increment_counter در نخ Thread2 هم قرار داده شده است. هر نخ پشته و رجیستر مخـتص بـه خود را دارد و متغیرهای محلی یک نخ در داخل پشته مربوط به هر نخ ایجاد می‌گـردد. امـا در دو نخ Thread1 و Thread2 که فعالیت increment_counter داخـل آن قـرار دارد، متغیـر counter از نوع static است که داخل سگمنت داده نگهداری می‌شود که به تبع در دو نخ به اشـتراک گذاشـته می‌شود، بنابراین متغیر counter منبع بحرانی و دستور counter= counter+1 ناحیه بحرانـی مـابین دو نخ محسوب می‌گردد که باید محافظت گردد. **بنابراین یک تابع همچون increment_counter در سطح نخ ایمن است اگر و فقط اگر از نخ‌های همروند فراخوانی شود، همیشه نتیجه درست را برگرداند.** دقت کنید که این رابطه دو طرفه صـادق اسـت، بنـابراین شـرط لازم و کـافی در هـر طرف برقرار است و طرف دیگر را نتیجه می دهد. بنابراین گزینه دوم پاسخ سوال است.

khalilifar.ir

تست‌های فصل هفتم: مدیریت بن‌بست

۱- فرض کنید سیستمی با 3 فرآیند با مشخصات زیر داشته باشیم:

e: execution time (زمان اجرا)

r: release time (زمان رسیدن)

R: number of required resources (تعداد منابع مورد نیاز)

$P_1 := r = 0 \ , e = 4 \ , R = 2$

$P_2 : r = 1 \ , \ e = 2.5 \ , R = 3$

$P_3 : r = 3 \ , e = 4 \ , R = 4$

با فرض وجود 4 منبع همسان در سیستم و این فرض که هر فرآیند در لحظات $\varepsilon, \varepsilon+1$ و ... پس از اجرا (ε یک عدد بسیار کوچک است) منابع انحصاری (Non Preemptive) خود را یکی‌یکی درخواست می‌کند و زمان‌بندی براساس الگوریتم Preemptive LCFS (LCFS غیرانحصاری یا قبضه‌ای) انجام می‌شود، کدام عبارت صحیح است؟ (مهندسی کامپیوتر – دولتی ۸۹)

(LCFS: Last – Come First Served)

۱) سیستم حدود لحظه 4 دچار بن‌بست (deadlock) می‌شود.

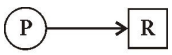
۲) سیستم دچار بن‌بست (deadlock) نمی‌شود و متوسط زمان کامل (Turnaround Time) برای آن 7 است.

۳) سیستم حدود لحظه 5 دچار بن‌بست (deadlock) می‌شود.

۴) سیستم دچار بن‌بست (deadlock) نمی‌شود و متوسط زمان کامل (Turnaround Time) برای آن 6.17 است.

۲- در یک گراف مربوط به درخواست و تخصیص منابع یال‌های زیر را تعریف می‌کنیم.

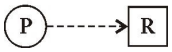
(نوع 1) فرآیند P متقاضی منبع R است:



(نوع 2) منبع R در اختیار فرآیند P است:



(نوع 3) فرآیند P مدعی استفاده از منبع R در آینده است:



با فرض وجود یک نمونه (instance) از هر منبع کدام عبارت درست است؟ (مهندسی IT- دولتی۸۹)

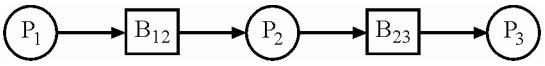
۱) وجود حلقه در گراف شامل یال‌های نوع 1 و 2 به منزله احتمال وجود بن‌بست و وجود حلقه در گراف شامل یال‌های نوع 1، 2 و 3 به منزله وجود بن‌بست است.

۲) وجود حلقه در گراف شامل یال‌های نوع 1 و 2 به منزله احتمال وجود بن‌بست و وجود حلقه در گراف شامل یال‌های نوع 1، 2 و 3 به منزله احتمال وجود بن‌بست است.

khalilifar.ir

۳) وجود حلقه در گراف شامل یال‌های 1 و 2 به منزله وجود بن‌بست و وجود حلقه در گراف شامل یال‌های نوع 1، 2 و 3 به منزله احتمال وجود بن‌بست است.
۴) وجود حلقه در گراف شامل یال‌های نوع 1 و 2 به منزله وجود بن‌بست و وجود حلقه در گراف شامل یال‌های 1، 2 و 3 به منزله وجود بن‌بست است.

۳- سیستمی شامل سه فرآیند P_1، P_2 و P_3 که به صورت همروند اجرا می‌شوند و دو بافر B_{12} و B_{23} مطابق شکل زیر طرح شده است:



فرآیند P_2 خروجی فرآیند P_1 را مصرف می‌کند و فرآیند P_3 خروجی فرآیند P_2 را مصرف می‌کند. بافرها در دیسک پیاده‌سازی شده و بلوک‌های داده‌ای که فرآیندها تبادل می‌کنند، دارای اندازه برابر هستند. فضای مشترکی در دیسک برای بافرهای B_{12} و B_{23} در نظر گرفته شده است و مرز بین دو بافر شناور است و هر بافر تا زمانی که فضا روی دیسک موجود باشد می‌تواند رشد کند، یعنی: $C_{12} + C_{23} \le C$

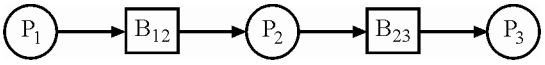
C: تعداد کل بلوک‌های در نظر گرفته شده برای B_{12} و B_{23}

C_{12}: تعداد بلوک‌های بافر B_{12}

C_{23}: تعداد بلوک‌های بافر B_{23} **(مهندسی IT- دولتی۹۰)**

۱) این سیستم می‌تواند در بن‌بست قرار گیرد ولی گرسنگی (starvation) ندارد.
۲) این سیستم در بن‌بست قرار نمی‌گیرد و گرسنگی (starvation) ندارد.
۳) این سیستم در بن‌بست قرار نمی‌گیرد ولی می‌تواند دچار گرسنگی (starvation) شود.
۴) این سیستم هم می‌تواند در بن‌بست قرار گیرد و هم می‌تواند دچار گرسنگی (starvation) شود.

۴- سیستمی شامل سه فرآیند P_1، P_2 و P_3 و دو بافر B_{12} و B_{23} مطابق شکل زیر طرح شده است:



فرآیند P_2 خروجی فرآیند P_1 را مصرف می‌کند و فرآیند P_3 خروجی فرآیند P_2 را مصرف می‌کند. بافرها در دیسک پیاده‌سازی شده و بلوک‌های داده‌ای که فرآیندها تبادل می‌کنند، دارای اندازه برابر هستند. فضای مشترکی در دیسک برای بافرهای B_{12} و B_{23} در نظر گرفته شده است و مرز بین دو بافر شناور است و هر بافر تا زمانی که فضا روی دیسک موجود باشد می‌تواند رشد کند، یعنی: $C_{12} + C_{23} \le C$

C: تعداد کل بلوک‌های در نظر گرفته شده برای B_{12} و B_{23}

C_{12}: تعداد بلوک‌های بافر B_{12}

khalilifar.ir

C_{23}: تعداد بلوک‌های بافر B_{23}

(مهندسی کامپیوتر – دولتی۹۱)

۱) این سیستم می‌تواند در بن‌بست قرار گیرد ولی گرسنگی (starvation) ندارد.
۲) این سیستم در بن‌بست قرار نمی‌گیرد و گرسنگی (starvation) ندارد.
۳) این سیستم در بن‌بست قرار نمی‌گیرد ولی می‌تواند دچار گرسنگی (starvation) شود.
۴) این سیستم هم می‌تواند در بن‌بست قرار گیرد و هم می‌تواند دچار گرسنگی (starvation) شود.

۵- کامپیوتری دارای m واحد از یک منبع است و n پردازه برای در اختیار گرفتن این منابع با هم رقابت می‌کنند. هر پردازه حداکثر به دو واحد از این منبع نیاز داشته و در هر زمان می‌تواند تنها یک واحد منبع را آزاد یا درخواست نماید. کدام یک از گزینه‌های زیر درباره حداکثر مقدار n که به ازای آن سیستم دچار بن‌بست نمی‌شود، درست است؟ (مهندسی کامپیوتر – دولتی۹۳)

۱) $n<m$ ۲) $n<2m-1$ ۳) $n<m+2$ ۴) $n<2m$

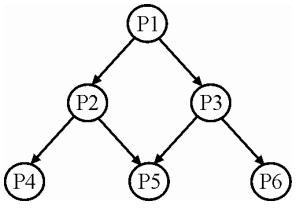
۶- دو فرآیند همروند P و Q را در نظر گیرید، کمترین اندازه ممکن برای حافظه اصلی (بدون در نظر گرفتن حافظه مورد نیاز هسته سیستم عامل) برای آنکه بن‌بست رخ ندهد، چند کیلوبایت است؟

(مهندسی IT- دولتی۹۳)

P:		Q:	
⋮		⋮	
Request	x_1k_B	Request	y_1k_B
⋮		⋮	
Request		Request	
⋮	x_2k_B	⋮	y_2k_B

۱) $x_1+y_1+\max(x_2,y_2)$
۲) $x_1+x_2+y_1+y_2$
۳) $x_1+y_1+\min(x_2,y_2)$
۴) x_1+x_2

۷- گراف تقدم – تأخر اجرای 6 پردازه (Process)، را در شکل زیر داریم. جدول نیازمندی پردازه‌ها به منابع داده شده است. حداقل چند واحد از هر یک از منابع B و A به ترتیب از راست به چپ داشته باشیم تا بن‌بست پیش نیاید؟ (مهندسی کامپیوتر- دولتی۹۵)



پردازه / منبع	P1	P2	P3	P4	P5	P6
A	4	2	3	4	3	2
B	2	3	2	1	2	1

۱) 5 و 5 ۲) 4 و 7 ۳) 5 و 9 ۴) 11 و 16

---

khalilifar.ir

۸- سامانه‌ای را در نظر بگیرید که در آن سه پردازه P_1, P_2, P_3 و چهار منبع R_1, R_2, R_3, R_4 وجـود دارند و حالت سامانه به صورت جدول زیر باشد، وضعیت این سامانه چگونه است؟

(مهندسی IT – دولتی ۹۵)

	تخصیص				نیاز				موجود			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	0	0	1	0	2	0	0	1	2	1	0	0
P_2	2	0	0	1	1	0	1	0				
P_3	0	1	2	0	2	1	0	0				

۱) در حالت مطمئن است.

۲) در حالت غیرمطمئن است.

۳) در حالت بن‌بست است.

۴) در حالت بن‌بست نیست اما حالت سامانه مطمئن نیست.

۹- در سیستمی با پنج پردازه و سه نوع منبع، وضعیت تخصیص منابع به صورت زیر اسـت. اگـر در این وضعیت، درخواستی برای یک واحد دیگر از منبع A توسط پردازه P_3 صادر شود، کدام مورد درست است؟

(مهندسی IT – دولتی ۹۶)

(تعداد منابع اولیه)

A	B	C
8	6	10

(حداکثر نیاز)

	A	B	C
P_0	3	6	8
P_1	7	3	6
P_2	5	3	3
P_3	4	5	9
P_4	2	3	3

(تخصیص‌یافته)

	A	B	C
P_0	0	1	2
P_1	2	0	3
P_2	3	2	0
P_3	1	0	2
P_4	1	1	0

۱) بعد از انجام درخواست فوق بن‌بست قطعی است.

۲) قبل از انجام درخواست فوق وقوع بن‌بست قطعی است.

۳) قبل از درخواست فوق احتمال وقوع بن‌بست وجود دارد.

۴) بعد از انجام درخواست فوق احتمال وجود بن‌بست وجود دارد.

---

khalilifar.ir

پاسخ تست‌های فصل هفتم: مدیریت بن‌بست

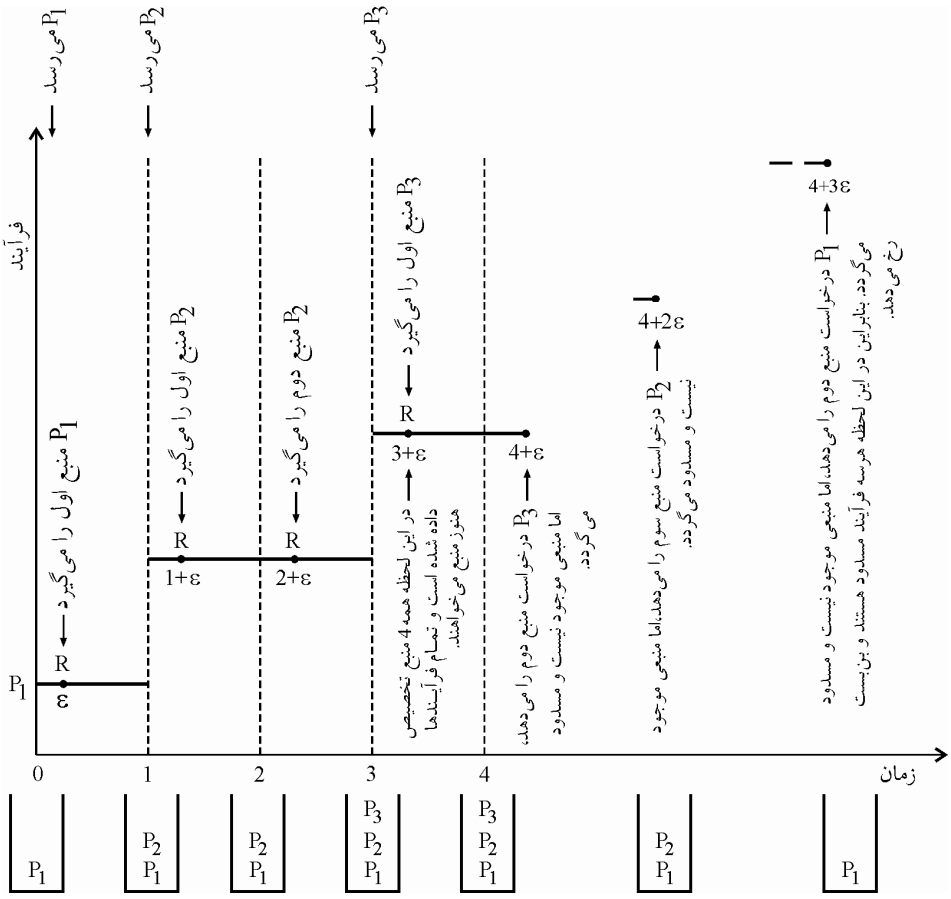
۱- گزینه (۱) صحیح است.

با توجه به اطلاعات مطرح شده در نظر بگیرید:

فرآیندها	زمان ورود	زمان اجرا	منابع مورد نیاز
P_1	0	4	2
P_2	1	2.5	3
P_3	3	4	4

راه حل اول:

با توجه به اینکه زمان‌بندی غیر انحصاری است و فرآیندی که از آخر وارد می‌شود، زودتر سرویس می‌گیرد می‌توان نمودار زیر را رسم کرد:



khalilifar.ir

توجه: با توجه به نوع زمان‌بندی الگوریتم Preemptive-LCFS، با ورود هر فرآیند جدید (آخرین فرآیند)، پردازنده به آن اختصاص می‌یابد تا اینکه آن فرآیند تمام شود یا فرآیند دیگری وارد شود.
توجه: هر فرآیند در لحظه ε پس از اجرا، منبع مورد نیاز خود را درخواست می‌کند.
توجه: در صورت سوال مطرح شده است که تخصیص منابع به صورت انحصاری است و این به این معنی است که سیستم عامل نمی‌تواند منبع تخصیص داده شده را قبل از آزادسازی و به زور پس بگیرد.
توجه: اینکه یک فرآیند یک منبع را به صورت انحصاری در اختیار دارد با اینکه یک فرآیند همه منابع خود را تا انتهای اجرا در اختیار می‌گیرد، تفاوت دارد. ممکن است فرآیندی یک منبع را به شکل انحصاری در اختیار بگیرد و سپس پس از انجام کار با آن منبع، آنرا رها سازد و سپس منبع دوم خود را در اختیار بگیرد. بنابراین انحصاری بودن منابع، به معنی آن نیست که یک فرآیند همه‌ی منابع خود را تا پایان اجرا نگه می‌دارد. بهتر بود که در صورت سوال این شرط قید می‌شد که فرآیندها تا پایان اجرا همه‌ی منابع خود را نگه می‌دارند، یعنی نگهداری و درخواست (انتظار).
با توجه به اینکه صحبتی از آزادسازی منابع نشده است، می‌توان حدس زد این شرطی که ما در مورد آن صحبت کردیم در دل سوال نهفته است و منظور طراح نیز همین بوده است، یعنی نگهداری و درخواست (انتظار) و نه رهاسازی و درخواست. البته مطابق فرض سؤال اینکه گفته شده است که هر فرآیند منابع انحصاری خود را یکی یکی درخواست می‌کند، این عبارت یکی یکی درخواست می‌کند همان به معنی درخواست و نگهداری (انتظار) است و نه رهاسازی و درخواست.
همانطور که در نمودار مشخص است، در لحظه $t=0$، فرآیند P_1 وارد سیستم شده و پردازنده را در اختیار می‌گیرد و در زمان ε اولین منبع خود را دریافت می‌کند، اما در لحظه‌ی $t=1$ و قبل از گرفتن منبع دوم توسط فرآیند P_1، پردازنده از فرآیند P_1 گرفته شده و به فرآیند P_2 (به دلیل ورود فرآیند P_2) داده می‌شود. (توجه کنید که الگوریتم LCFS غیر انحصاری است). فرآیند P_2 نیز پس از گذشت مدت زمان ε از شروعش یعنی در زمان $1+\varepsilon$ اولین منبع خود را دریافت می‌کند و همچنین در زمان $2+\varepsilon$ دومین منبع خود را نیز دریافت می‌کند. اما در لحظه‌ی $t=3$ و قبل از گرفتن منبع سوم توسط فرآیند P_2، پردازنده از فرآیند P_2، گرفته شده و به فرآیند P_3 (به دلیل ورود فرآیند P_3) داده می‌شود، در ادامه فرآیند P_3 نیز پس از گذشت مدت ε از شروعش یعنی در زمان $3+\varepsilon$ اولین منبع خود را دریافت می‌کند.
در این لحظه یعنی $3+\varepsilon$ همه‌ی منابع تخصیص داده شده است و تمام فرآیندها هنوز منبع می‌خواهند. همچنین از این لحظه به بعد فرآیند جدیدی نیز وارد سیستم نمی‌گردد، بنابراین همچنان فرآیند P_3 آخرین فرآیند موجود در سیستم است. بنابراین در لحظه $4+\varepsilon$ فرآیند P_3، درخواست دریافت دومین منبع خود را به سیستم ارائه می‌کند اما موفق به دریافت آن نمی‌گردد،

khalilifar.ir

زیرا دیگر منبع آزادی باقی نمانده است، بنابراین در لحظه‌ی 4+ε فرآیند P_3 مسدود می‌گردد. حال در ادامه پردازنده **بلافاصله** به فرآیند P_2 تعویض متن می‌کند. بنابراین در لحظه‌ی 4+2ε فرآیند P_2، (یک ε پس از اجرا، شروع 4+ε بوده است، پس درخواست در لحظه 4+2ε رخ می‌دهد) درخواست سومین منبع خود را به سیستم ارائه می‌کند، اما موفق به دریافت آن نمی‌گردد، زیرا دیگر منبع آزادی باقی نمانده است، بنابراین در لحظه‌ی 4+2ε فرآیند P_2 نیز مسدود می‌گردد. و در نهایت پردازنده مجدداً **بلافاصله** به فرآیند P_1 تعویض متن می‌کند. بنابراین در لحظه‌ی 4+3ε فرآیند P_1، (یک ε پس از اجرا، شروع 4+2ε بوده است، پس درخواست در لحظه 4+3ε رخ می‌دهد) درخواست دومین منبع خود را به سیستم ارائه می‌کند. اما موفق به دریافت آن نمی‌گردد، زیرا دیگر منبع آزادی باقی نمانده است. بنابراین در لحظه‌ی 4+3ε فرآیند P_1 نیز مسدود می‌گردد.
در نتیجه در لحظه‌ی 4+3ε یعنی حدود ثانیه چهارم هر سه فرآیند مسدود می‌گردند زیرا، نه راه پس دارند و نه راه پیش، پس بن بست به وقوع می‌پیوندد. بن بست در سرنوشت‌های به هم گره خورده رخ می‌دهد.

راه حل دوم:

مطابق فرض سؤال، گفته شده است که هر فرآیند در لحظات ε، 1+ε و ... پس از اجرا، منابع انحصاری (Non Preemptive) خود را یکی یکی درخواست می‌کند و زمان‌بندی براساس الگوریتم Preemptive LCFS (LCFS غیرانحصاری یا قبضه‌ای) انجام می‌شود، اینکه گفته شده است که هر فرآیند منابع انحصاری خود را یکی یکی درخواست می‌کند، این عبارت یکی یکی درخواست می‌کند همان به معنی درخواست و نگهداری (انتظار) است یعنی یک Request جدید پس از یک Request قدیم و نه رهاسازی و درخواست یعنی یک Release پس از یک Request قدیم و سپس یک Request جدید.

توجه: در حل مسائل بن‌بست، جهت کنترل بهتر، بهتر است، قطعه کد فرآیندها براساس فرض سؤال درنظر گرفته شود.

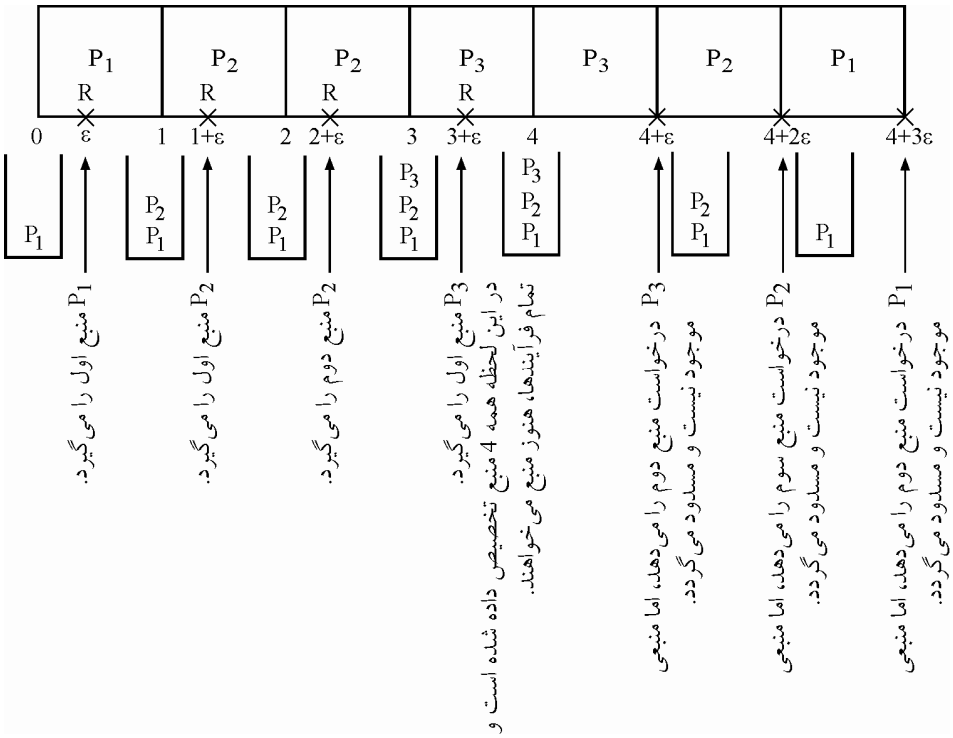
قطعه کد درخواست منابع همسان برای سه فرآیند P_1، P_2 و P_3 به صورت زیر است:

P_1	P_2	P_3
ε: Request (R)	1+ε: Request (R)	3+ε: Request (R)
4+3ε: Request (R) → می‌خوابد	2+ε: Request (R)	4+ε: Request (R)→ می‌خوابد
	4+2ε: Request (R) → می‌خوابد	Request (R)
		Request (R)

توجه: واضح است که مطابق فرض سؤال، فقط Request داریم و نه Release .

با توجه به مفروضات مسئله، نمودار گانت زیر را داریم:

khalilifar.ir

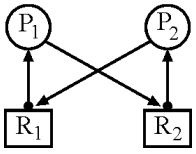


در نتیجه در لحظه $4+3\varepsilon$ یعنی حدود ثانیه چهارم هر سه فرآیند مسدود می‌گردند، زیرا نه راه پس دارند و نه راه پیش، پس بن‌بست به وقوع می‌پیوندد. بن‌بست در سرنوشت‌های به هم گره خورده رخ می‌دهد.

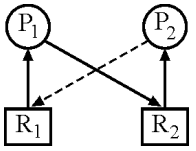
۲- گزینه (۳) صحیح است.

وجود حلقه در گراف شامل یال‌های نوع 1 و2 به منزله وجود قطعی بن بست است، زیرا بطور مثال فرآیندهای P_1 و P_2 به ترتیب منابع R_1 و R_2 را دارند و در یک لحظه بطور قطعی فرآیندهای P_1 و P_2 به ترتیب منابع R_2 و R_1 را نیز درخواست می‌کنند، یعنی در عین حال اینکه، فرآیندهای P_1 و P_2 به ترتیب منابع R_1 و R_2 را در اختیار دارند، برای پایان کار خود بطور قطعی درخواست داده‌اند که به ترتیب منابع R_2 و R_1 را نیز می‌خواهند. نگهداری و درخواست (انتظار) بنابراین هر چهار شرط کافمن برقرار است، (شرط انحصار متقابل به طور ذاتی برقرار است، شرط انحصاری بودن به طور ذاتی برقرار است، شرط نگهداری و درخواست (انتظار) برقرار شده است و همچنین شرط سیکل (انتظار چرخش) نیز برقرار شده است) پس وقوع بن بست حتمی است.

khalilifar.ir



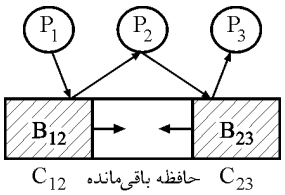
اما وجود حلقه درگراف شامل یال‌های نوع 1، 2 و3 به منزله احتمال وجود بن بست است.



زیرا بطور مثال فرآیندهای P_1 و P_2 به ترتیب منابع R_1 و R_2 را دارند و بطور قطعی فرآیند P_1، منبع R_2 را درخواست می‌کند، یعنی در عین حال اینکه فرآیند P_1 منبع R_1 را در اختیار دارد، برای پایان کار خود به طور قطعی درخواست داده است که منبع R_2 را نیز می‌خواهد. نگهداری و درخواست (انتظار). اما رفتار فرآیند P_2 به واسطه استفاده از نماد خط‌چین یعنی یال نوع 3 در آینده مشخص نیست، یعنی مشخص نیست که فرآیند P_2، در آینده رفتار نگهداری و درخواست (انتظار) را در پیش می‌گیرد، و یا رفتار رهاسازی و درخواست را. اگر فرآیند P_2، در آینده رفتار نگهداری و درخواست (انتظار) را در پیش بگیرد، یعنی در عین حال اینکه فرآیند P_2 منبع R_2 را در اختیار دارد، برای پایان کار خود به طور قطعی درخواست بدهد که منبع R_1 را نیز می‌خواهد، آنگاه در کنار رفتار نگهداری و درخواست (انتظار) مربوط به فرآیند P_1 بن‌بست ایجاد می‌گردد. اما اگر فرآیند P_2، در آینده رفتار رهاسازی و درخواست را در پیش بگیرد، یعنی فرآیند P_2 ابتدا منبع R_2 را رها کند و سپس برای پایان کار خود درخواست بدهد که منبع R_1 را نیز می‌خواهد، آنگاه در کنار رفتار نگهداری و درخواست (انتظار) مربوط به فرآیند P_1 بن‌بست ایجاد نمی‌گردد، زیرا شرایط چهارگانه کافمن برقرار نمی‌گردد. بنابراین وجود حلقه در گراف شامل یال‌های 1، 2 و 3 به منزله احتمال وجود بن‌بست است و نه به منزله وجود بن‌بست. بنابراین گزینه سوم پاسخ سؤال است.

۳- گزینه (۱) صحیح است.

شکل زیر را در نظر بگیرید:



khalilifar.ir

مجدداً صورت سؤال را با دقت بخوانید و توجه کنید که فرآیند P_2 خروجی فرآیند P_1 را مصرف می‌کند و فرآیند P_3 خروجی فرآیند P_2 را مصرف می‌کند. این سیستم می‌تواند دچار بن بست شود. سناریوی زیر را در نظر بگیرید:

۱- فرآیند P_1 تمام بلوک‌های دیسک را پُر نماید و در ادامه به دلیل عدم فضای خالی در بافر B_{12} مسدود گردد. در این شرایط $B_{23}=0$ و $B_{12}=C$ است.

۲- فرآیند P_2 اولین بلوک داده تولید شده توسط فرآیند P_1 را به عنوان ورودی مصرف نماید. (یک بلوک دیسک از بافر B_{12} خالی می‌شود)، بنابراین فرآیند P_1 به دلیل وجود فضای خالی به اندازه یک بلوک در بافر B_{12} از حالت مسدود خارج می‌گردد.

۳- فرآیند P_1 اجرا شده، بلوک خالی را پُر می‌نماید و مجدداً به دلیل عدم فضای خالی مسدود می‌گردد. در این شرایط $B_{23}=0$ و $B_{12}=C$ است.

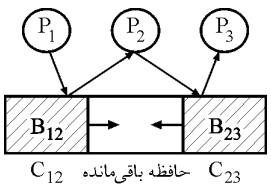
۴- هم اکنون فرآیند P_2 بخاطر پُر بودن تمام بلوک‌های دیسک، نمی‌تواند خروجی خود را در بافر خود یعنی B_{23} قرار دهد، زیرا تمام بافر در اختیار B_{12} است، بنابراین فرآیند P_2 نیز به دلیل عدم فضای خالی در بافر B_{23} مسدود می‌گردد.

۵- در ادامه، فرآیند P_3 نیز، منتظر بلوک خروجی مربوط به فرآیند P_2 است (به عنوان داده ورودی خود)، بنابراین فرآیند P_3 نیز به دلیل عدم وجود این داده ورودی، به دلیل عدم فضای خالی برای خروجی فرآیند P_2، مسدود می‌گردد.

در این شرایط هر سه فرآیند مسدود و بن بست رخ می‌دهد. در این سیستم گرسنگی وجود ندارد، گرسنگی یعنی فرآیندی مرتباً کار کند و از کار کردن فرآیند دیگری به طور مداوم جلوگیری کند. در حالی که بافر محدود است و یک فرآیند نمی تواند تا ابد کار کند. دیگر اینکه وابستگی فرآیندها به یکدیگر وجود دارد، یعنی کارکرد فرآیندها مطابق یک نظم و ترتیبی می‌باشد و نه شانسی و تصادفی، مطابق صورت سؤال فرآیند P_2 خروجی فرآیند P_1 را مصرف می‌کند و فرآیند P_3 خروجی فرآیند P_2 را مصرف می‌کند. بنابراین در این سیستم فرآیندها هیچگاه دچار گرسنگی نمی‌شوند.

۴- گزینه (۱) صحیح است.

شکل زیر را در نظر بگیرید:



khalilifar.ir

مجدداً صورت سوال را به دقت بخوانید و توجه کنید که فرآیند P_2 خروجی فرآیند P_1 را مصرف می‌کند و فرآیند P_3 خروجی فرآیند P_2 را مصرف می‌کند. این سیستم می‌تواند دچار بن بست شود. سناریوی زیر را در نظر بگیرید:

۱- فرآیند P_1 تمام بلوک‌های دیسک را پُر نماید و در ادامه به دلیل عدم فضای خالی در بافر B_{12} مسدود گردد. در این شرایط $B_{23}=0$ و $B_{12}=C$ است.

۲- فرآیند P_2 اولین بلوک داده تولید شده توسط فرآیند P_1 را به عنوان ورودی مصرف نماید. (یک بلوک دیسک از بافر B_{12} خالی می‌شود)، بنابراین فرآیند P_1 به دلیل وجود فضای خالی به اندازه یک بلوک در بافر B_{12} از حالت مسدود خارج می‌گردد.

۳- فرآیند P_1 اجرا شده، بلوک خالی را پُر می‌نماید و مجدداً به دلیل عدم فضای خالی مسدود می‌گردد. در این شرایط $B_{23}=0$ و $B_{12}=C$ است.

۴- هم اکنون فرآیند P_2 به خاطر پُر بودن تمام بلوک‌های دیسک، نمی‌تواند خروجی خود را در بافر خود یعنی B_{23} قرار دهد، زیرا تمام بافر در اختیار B_{12} است، بنابراین فرآیند P_2 نیز به دلیل عدم فضای خالی در بافر B_{23} مسدود می‌گردد.

۵- در ادامه، فرآیند P_3 نیز، منتظر بلوک خروجی مربوط به فرآیند P_2 است (به عنوان داده ورودی خود)، بنابراین فرآیند P_3 نیز به دلیل عدم وجود این داده ورودی، به دلیل عدم فضای خالی برای خروجی فرآیند P_2، مسدود می‌گردد.

در این شرایط هر سه فرآیند مسدود شده و بن بست رخ می‌دهد. در این سیستم گرسنگی وجود ندارد، گرسنگی یعنی فرآیندی مرتباً کار کند و از کار کردن فرآیند دیگری به طور مداوم جلوگیری کند. در حالی که بافر محدود است و یک فرآیند نمی‌تواند تا ابد کار کند. دیگر اینکه وابستگی فرآیندها به یکدیگر وجود دارد، یعنی کارکرد فرآیندها مطابق یک نظم و ترتیبی می‌باشد و نه شانسی و تصادفی، مطابق صورت سوال فرآیند P_2 خروجی فرآیند P_1 را مصرف می‌کند و فرآیند P_3 خروجی فرآیند P_2 را مصرف می‌کند. بنابراین در این سیستم فرآیندها هیچگاه دچار گرسنگی نمی‌شوند.

۵- گزینه (۱) صحیح است.

توجه: در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

مجموع درخواست‌های فرآیندها برای منابع: $\sum_{i=1}^{n} \text{Request}[i] < m + n$

$$\rightarrow 2n < m + n \rightarrow n < m$$

khalilifar.ir

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد بن‌بست به صورت زیر است:

$$n\times(r-1)=m$$

حال اگر مقدار m حداقل یک واحد بیشتر از $n\times(r-1)$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

$$n\times(r-1)<m$$

$$n\times r-n<m$$

$$n\times r<m+n$$

$$\sum_{i=1}^{n}\text{Request}[i]<m+n$$

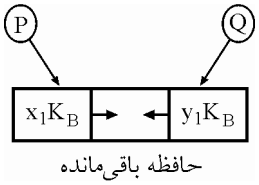
$$\longrightarrow 2n<m+n \;\rightarrow\; n<m$$

۶- گزینه (۳) صحیح است.

مطابق کد مطرح شده در صورت سئوال برای دو فرآیند Q و P داریم:

P:	Q:
⋮	⋮
Request x_1 K_B	Request y_1 K_B
⋮	⋮
Request x_2 K_B	Request y_2 K_B

khalilifar.ir



اگر دو فرآیند P و Q به ترتیب x_1 و y_1 کیلو بایت از حافظه را اشغال کنند و هر دو مطابق کد فرآیندها، درخواست x_2 و y_2 کیلوبایت حافظه جدید را داده باشند، باید حداقل به اندازه $\min(x_2, y_2)$ حافظه باقی‌مانده باشد تا یکی از دو فرآیند ادامه یافته و از حافظه خارج شود.

توجه: دقت کنید که شاید به ظاهر فکر کنید که حداقل باید به اندازه $\max(x_2,y_2)$ باشد، اما همان حداقل به اندازه $\min(x_2,y_2)$ برای آنکه بن‌بست رخ ندهد کافی است. فرض کنید مقدار y_2 بیشتر از x_2 باشد $(y_2 > x_2)$، بنابراین $\min(x_2,y_2)$ برابر مقدار x_2 می‌شود. حال اگر در ادامه، اول فرآیند P اجرا گردد، آنگاه x_2k_B حافظه جدید که کافی هم هست، به آن داده می‌شود و سپس فرآیند P تمام می‌گردد و حافظه‌های x_1k_B و x_2k_B آزاد می‌گردند. در ادامه نیز فرآیند Q اجرا می‌گردد و تمام می‌شود. و بن‌بست نمی‌شود. همچنین اگر در ادامه، اول فرآیند Q اجرا گردد از آنجا که نیاز به y_2k_B حافظه جدید دارد و موجود نیست، پس در ادامه فرآیند Q می‌خوابد، در ادامه نیز فرآیند P اجرا می‌گردد و x_2k_B حافظه جدید که کافی هم هست، به آن داده می‌شود و سپس فرآیند P تمام می‌گردد و حافظه‌های x_1k_B و x_2k_B آزاد می‌گردند که منجر به بیدار شدن فرآیند Q می‌شود. در ادامه نیز فرآیند Q اجرا می‌گردد و تمام می‌شود. و بن‌بست نمی‌شود. پس حداقل اندازه ممکن برای حافظه اصلی برای آنکه بن‌بست رخ ندهد، برابر $x_1+y_1+\min(x_2,y_2)$ کیلوبایت است. بنابراین گزینه سوم پاسخ سؤال است.

۷- گزینه (۳) صحیح است.

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$ منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

khalilifar.ir

اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$ منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد بن‌بست به صورت زیر است:

$$n\times(r-1)=m$$

حال اگر مقدار m حداقل یک واحد بیشتر از $n\times(r-1)$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

$$n\times(r-1)<m$$

$$n\times r-n<m$$

$$n\times r<m+n$$

$$\sum_{i=1}^{n}\text{Request}[i]<m+n$$

بنابراین در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

$\sum_{i=1}^{n}\text{Request}[i]<m+n$: مجموع درخواست‌های فرآیندها برای منابع

با توجه به گراف تقدم – تأخر، واضح است که P_2 و P_3 زمانی می‌توانند اجرا شوند که P_1 خاتمه یابد. همچنین P_5 زمانی می‌تواند اجرا شود که P_2 و P_3 خاتمه یابند، P_4 زمانی می‌تواند اجرا شود که P_2 خاتمه یابد و P_6 زمانی می‌تواند اجرا شود که P_3 خاتمه یابد.

با توجه به گراف داده شده فرآیندها یا به صورت (1) **غیرهمروند و تنها** اجرا می‌شوند مانند اجرای فرآیند P_1 که برای ایجاد بن‌بست معنا ندارد، چون یک فرآیند به تنهایی بن‌بست با خودش ایجاد نمی‌کند و یا به صورت (2) **همروند** اجرا می‌شوند، که حالت‌های زیر را داریم:

حالت‌ها	فرآیندها	نیاز به منبع A	MAX(A)	نیاز به منبع B	MAX(B)
1	P_2,P_3	2+3=5	--	3+2=⑤	⑤
2	P_2,P_6	2+2=4	--	3+1=4	--
3	P_3,P_4	3+4=7	--	2+1=3	--
4	P_4,P_5,P_6	4+3+2=⑨	⑨	1+2+1=4	--

همانطور که گفتیم، در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

$\sum_{i=1}^{n}\text{Request}[i]<m+n$: مجموع درخواست‌های فرآیندها برای منابع

مطابق رابطه فوق برای منبع A داریم:

khalilifar.ir

حالت اول: همروندی P_2,P_3

$$2+3<m+2 \rightarrow m>3$$

حالت دوم: همروندی P_2,P_6

$$2+2<m+2 \rightarrow m>2$$

حالت سوم: همروندی P_3,P_4

$$3+4<m+2 \rightarrow m>5$$

حالت چهارم: همروندی P_4,P_5,P_6

$$4+3+2<m+3 \rightarrow \boxed{m>6}$$

بنابراین حداقل منبع لازم از A، در بین این 6 فرآیند، برای اینکه بن‌بست پیش نیاید، به صورت زیر است:

$$m>6 \rightarrow m=7$$

نتیجه فوق واضح است، چون همانطور که پیش‌تر گفتیم، در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$ منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین شرایط ایجاد بن‌بست به صورت زیر است:

$$n\times(r-1)=m$$
$$(4-1)+(3-1)+(2-1)=m$$
$$3+2+1=m \rightarrow m=6$$

حال اگر حداقل نمونه دیگری از منبع A در سیستم موجود باشد، آن نمونه، به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند، سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.
بنابراین همانطور که گفتیم حداقل منبع لازم از A، در بین این 6 فرآیند، برای اینکه بن‌بست پیش نیاید، به صورت زیر است:

$$m>6 \rightarrow m=7$$

مطابق رابطه مطرح شده برای منبع B نیز داریم:

حالت اول: همروندی P_2,P_3

$$3+2<m+2 \rightarrow \boxed{m>3}$$

حالت دوم: همروندی P_2,P_6

$$3+1<m+2 \rightarrow m>2$$

حالت سوم: همروندی P_3,P_4

$$2+1<m+2 \rightarrow m>1$$

khalilifar.ir

حالت چهارم: همروندی P_4,P_5,P_6

$$1+2+1<m+3 \rightarrow m>1$$

بنابراین حداقل منبع لازم از B، در بین این 6 فرآیند، برای اینکه بن‌بست پیش نیاید، به صورت زیر است:

$$m>3 \rightarrow m=4$$

نتیجه فوق واضح است، چون همانطور که پیش‌تر گفتیم، در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$ منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین شرایط ایجاد بن‌بست به صورت زیر است:

$$n\times(r-1)=m$$

$$(3-1)+(2-1)=m$$

$$2+1=m \rightarrow m=3$$

حال اگر حداقل نمونه دیگری از منبع B در سیستم موجود باشد، آن نمونه، به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند، سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

بنابراین همانطور که گفتیم حداقل منبع لازم از B، در بین این 6 فرآیند، برای اینکه بن‌بست پیش نیاید، به صورت زیر است:

$$m>3 \rightarrow m=4$$

بنابراین در حالت تک منبعی حداقل منبع لازم از B و A به ترتیب از راست به چپ برای اینکه بن‌بست پیش نیاید برابر 4 و 7 است.

اما در حالت چند منبعی برای داشتن سیستم بدون بن‌بست، محافظه‌کارانه‌تر از سیستم تک منبعی برخورد می‌کنیم و می‌گوییم برای عاری بودن سیستم از بن‌بست باید به اندازه مجموع حداکثر نیاز همه فرآیندها از هر یک از انواع منابع در اختیار داشته باشیم. بنابراین مطابق جدول صفحه قبل، مقادیر MAX در B و A را یعنی 5 و 9 در نظر می‌گیریم.

سازمان سنجش آموزش کشور در کلید اولیه و کلید نهایی گزینه سوم را به عنوان پاسخ اعلام کرده بود.

۸- گزینه (۱) صحیح است.

توجه: هنگامی که رفتار فرآیندها از جهت نگهداری و درخواست و یا رهاسازی و درخواست مشخص نباشد، نیاز به ماتریس Need است که در این شرایط برای تشخیص وضعیت امن یا ناامن، الگوریتم تشخیص وضعیت امن یا ناامن بانکداران مورد استفاده قرار می‌گیرد.

توجه: هنگامی که رفتار فرآیندها نگهداری و درخواست باشد، بنابراین به جای ماتریس Need، نیاز به ماتریس Request وجود دارد، که در این شرایط برای کشف بن‌بست، الگوریتم کشف بن‌بست کافمن (Coffman) مورد استفاده قرار می‌گیرد.

khalilifar.ir

توجه: الگوریتم بانکداران، جهت بررسی امن یا ناامن بودن مورد استفاده قرار می‌گیرد و در مورد بن‌بست نظر نمی‌دهد.

توجه: الگوریتم کافمن (Coffman)، جهت کشف بن‌بست مورد استفاده قرار می‌گیرد.

با توجه به ماتریس Need موجود در صورت سؤال، الگوریتم تشخیص وضعیت امن یا ناامن بانکداران مورد استفاده قرار می‌گیرد. همانطور که گفتیم الگوریتم بانکداران فقط در مورد امن یا ناامن بودن سیستم نظر می‌دهد و در مورد بن‌بست نظر نمی‌دهد. بنابراین گزینه سوم پاسخ سؤال نیست.

مطابق صورت سؤال، بردار موجود (Available) به صورت زیر است:

Available	2	1	0	0

براساس صورت سؤال، ماتریس نیاز (Need) به صورت زیر است:

فرآیند	نیاز (Need)			
	R_1	R_2	R_3	R_4
P_1	2	0	0	1
P_2	1	0	1	0
P_3	2	1	0	0

$$(2,1,0,0) \xrightarrow[+(0,1,2,0)]{P_3} (2,2,2,0) \xrightarrow[+(2,0,0,1)]{P_2} (4,2,2,1) \xrightarrow[+(0,0,1,0)]{P_1} (4,2,3,1)$$

به این ترتیب، دنباله امن $< P_3, P_2, P_1 >$ (از چپ به راست) برای این سیستم موجود است. بنابراین سیستم در وضعیت امن (مطمئن) قرار دارد، و گزینه اول پاسخ سؤال است.

۹- گزینه (۴) صحیح است.

ابتدا وضعیت سیستم را قبل از اجابت درخواست فرآیند P_3 بررسی می‌کنیم:

با توجه به ماتریس تخصیص منابع (Allocation) و منابع اولیه، بردار Available را بدست می‌آوریم، برای یافتن بردار Available ابتدا برای هر منبع، منابع تخصیص یافته به هر فرآیند را با هم جمع کرده و سپس از منابع اولیه کسر می‌نماییم.

منبع موجود A $= 8 - (2+3+1+1) = 1$

منبع موجود B $= 6 - (1+2+1) = 2$

منبع موجود C $= 10 - (2+3+2) = 3$

بنابراین بردار Available به صورت زیر است:

Available	1	2	3

براساس رابطه زیر داریم:

$$\text{Need} = \text{MAX} - \text{Allocation}$$

khalilifar.ir

فرآیند	Need		
	A	B	C
P_0	3	5	6
P_1	5	3	3
P_2	2	1	3
P_3	3	5	7
P_4	1	2	3

=

فرآیند	MAX		
	A	B	C
P_0	3	6	8
P_1	7	3	6
P_2	5	3	3
P_3	4	5	9
P_4	2	3	3

−

فرآیند	Allocation		
	A	B	C
P_0	0	1	2
P_1	2	0	3
P_2	3	2	0
P_3	1	0	2
P_4	1	1	0

Available	1	2	3

$$(1,2,3) \xrightarrow[+(1,1,0)]{P_4} (2,3,3) \xrightarrow[+(3,2,0)]{P_2} (5,5,3) \xrightarrow[+(2,0,3)]{P_1} (7,5,6)$$

$$\xrightarrow[+(0,1,2)]{P_0} (7,6,8) \xrightarrow[+(1,0,2)]{P_3} (8,6,10)$$

به این ترتیب و با توجه به ماتریس‌های فوق، دنباله‌ی امنِ $< P_4, P_2, P_1, P_0, P_3 >$ (از چپ به راست) برای این سیستم موجود است. بنابراین قبل از اجابتَ درخواست فرآیند P_3، سیستم در حالت امن قرار دارد و گزینه‌های دوم و سوم نادرست هستند.

الگوریتم درخواست منبع بانکداران

هنگامی که فرآیند P_i، بردار Request[i] را به عنوان اعلام نیاز به منابع مطرح می‌کند، باید این درخواست مطابق مراحل زیر بررسی گردد:

۱) اگر $Request[i] \le Need[i]$ است به مرحله بعدی برو، در غیر اینصورت خطای تقاضای بیش از نیاز اعلام می‌شود. در واقع در این مرحله بررسی می‌شود که اگر فرآیند درخواست بیش از آن چه که در ابتدا اعلام نیاز کرده است را دارد، با این درخواست مخالفت شود.

۲) اگر $Request[i] \le Availabe[i]$ است به مرحله 3 برو، در غیر اینصورت منبع کافی جهت تخصیص به این فرآیند وجود نداشته و این فرآیند باید منتظر بماند.

۳) با فرض این که منابع مورد نیاز به این فرآیند اختصاص می‌یابد، در این صورت با اصلاح ماتریس‌ها و بردارها یک الگوریتم تشخیص وضعیت امن اجرا می‌شود. با اجرای این الگوریتم اگر سیستم در وضعیت امن باقی بماند، این منابع به فرآیند اختصاص می‌یابند، در غیراینصورت اگر با این تخصیص سیستم به وضعیت ناامن وارد شود، ماتریس‌ها و بردارها به حالت قبل بازگردانده شده و فرآیند تا آزاد شدن منابع بیشتر منتظر می‌ماند.

اصلاح ماتریس‌ها و بردارها در صورت تخصیص به صورت زیر انجام می‌شوند:

Available (new) = Available (old) – Request[i]

Need (new) [i] = Need (old) [i] – Request[i]

Allocation (new) [i] = Allocation (old) [i] + Request[i]

khalilifar.ir

مطابق فرض صورت سؤال، اگر در این وضعیت، درخواستی برای یک واحد دیگر از منبع A، توسط فرآیند P_3 صادر شود، مطابق الگوریتم درخواست منبع بانکداران، ابتدا بررسی می‌شود که آیا درخواست فرآیند، کم‌تر از نیاز اعلام شده آن است یا خیر.

$$Request[P_3] \le Need[P_3]$$
$$[1,0,0] \le [3,5,7]$$

چون شرط اول برقرار است، به سراغ شرط دوم می‌رویم.

در شرط دوم بررسی می‌شود که آیا درخواست داده شده، کمتر از منابع آزاد سیستم است یا خیر.

$$Request[P_3] \le Available$$
$$[1,0,0] \le [1,2,3]$$

چون هر دو شرط برقرار است، سراغ مرحله سوم از الگوریتم می‌رویم.

در این مرحله، با فرض اینکه درخواست داده شده، اعمال شود، ماتریس‌ها به صورت زیر بروزرسانی می‌شوند:

فرآیند	Need (new)		
	A	B	C
P_0	3	5	6
P_1	5	3	3
P_2	2	1	3
P_3	②	5	7
P_4	1	2	3

=

فرآیند	MAX		
	A	B	C
P_0	3	6	8
P_1	7	3	6
P_2	5	3	3
P_3	4	5	9
P_4	2	3	3

−

فرآیند	Allocation (new)		
	A	B	C
P_0	0	1	2
P_1	2	0	3
P_2	3	2	0
P_3	②	0	2
P_4	1	1	0

با توجه به ماتریس تخصیص منابع **جدید** ((Allocation(new)) و منابع اولیه، بردار Available(new) را بدست می‌آوریم، برای یافتن بردار Available(new) ابتدا برای هر منبع، منابع تخصیص‌یافته به هر فرآیند را با هم جمع کرده و سپس از منابع اولیه کسر می‌نماییم.

منبع موجود A $= 8-(2+3+2+1) = 0$

منبع موجود B $= 6-(1+2+1) = 2$

منبع موجود C $= 10-(2+3+2) = 3$

بنابراین بردار Available(new) به صورت زیر است:

Available(new)	⓪	2	3

مطابق رابطه زیر نیز می‌توان Available (new) را محاسبه نمود:

Available (new) = Available (old) – Request [P_3]

Available (new) = [1,2,3] – [1,0,0] = [0,2,3]

khalilifar.ir

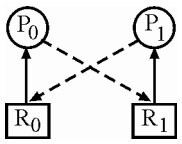
مشاهده می‌شود که با بردار Available(new) موجود، نمی‌توان نیاز هیچ فرآیندی را با توجه به ماتریس Need(new) مرتفع ساخت، بنابراین توالی امن یافت نمی‌شود و سیستم در یک وضعیت ناامن قرار دارد و احتمال وقوع بن‌بست وجود دارد. بنابراین پاسخ سؤال گزینه چهارم است.

توجه: ناامنی نه به معنی وقوع بن‌بست در گذشته و نه به معنی وقوع حتمی بن‌بست در آینده است، بلکه به معنی احتمال وقوع بن‌بست در آینده است.

توجه: در حالت ناامن نیز ممکن است، فرآیندها در موقعیت رهاسازی منابع در اختیار خود قرار بگیرند و سیستم به بن‌بست نرسد. به بیان دیگر در حالت ناامن وقوع بن‌بست قطعی نیست. اما اگر در حالت ناامن فرآیندها علاوه بر نگهداری منابع تحت تملک خود، منابع دیگری را مورد درخواست قرار دهند، در این حالت فرآیندهایی که نیاز آن‌ها برطرف نمی‌شود یک به یک در صف انتظار قرار می‌گیرند و همچون سرنوشت‌های به هم گره خورده، تا ابد منتظر یکدیگر باقی می‌مانند، یعنی بن‌بست رخ داده است.

مثال: در یک سیستم تعداد منبع اولیه R_0 برابر یک و تعداد منبع اولیه R_1 برابر یک است. اگر فرآیند P_0 یک منبع R_0 را در تملک داشته باشد و برای اتمام، نیاز به یک منبع R_1 نیز داشته باشد و همچنین فرآیند P_1 یک منبع R_1 را در تملک داشته باشد و برای اتمام، نیاز به یک منبع R_0 نیز داشته باشد، وضعیت سیستم را در این شرایط بررسی کنید:

حل: ماتریس و گراف منابع و فرآیندها به صورت زیر است:



فرآیند	MAX	
	R_0	R_1
P_0	1	1
P_1	1	1

=

فرآیند	Allocation	
	R_0	R_1
P_0	1	0
P_1	0	1

+

فرآیند	Need	
	R_0	R_1
P_0	0	1
P_1	1	0

با توجه به ماتریس تخصیص منابع (Allocation) و منابع اولیه، بردار Available را بدست می‌آوریم:

R_0 منبع موجود $= 1-(1) = 0$

R_1 منبع موجود $= 1-(1) = 0$

بنابراین بردار Available به صورت زیر است:

Available	0	0

khalilifar.ir

مشاهده می‌شود که با بردار Available موجود، نمی‌توان نیاز هیچ فرآیندی را با توجه به ماتریس Need مرتفع ساخت، بنابراین توالی امن یافت نمی‌شود و سیستم در یک وضعیت ناامن قرار دارد و احتمال وقوع بن‌بست وجود دارد.

توجه: در این شرایط ناامن، اگر فرآیند P_0 یا P_1 و یا هر دو، در موقعیت رهاسازی منابع در اختیار خود قرار بگیرند، یعنی رهاسازی کنند و سپس درخواست منابع باقی‌مانده خود را صادر کنند، بن‌بست رخ نمی‌دهد.

توجه: در این شرایط ناامن، اگر فرآیند P_0 در عین حال اینکه منبع R_0 را تحت تملک و نگهداری خود دارد، منبع R_1 را نیز درخواست کند، از آن‌جا که منبع R_1 در اختیار فرآیند P_1 می‌باشد، فرآیند P_0 در صف انتظار، می‌خوابد (نگهداری و انتظار) حال اگر در ادامه ماجرا، فرآیند P_1 نیز همین رویه را در پیش گیرد، یعنی فرآیند P_1 نیز در عین حال اینکه منبع R_1 را تحت تملک و نگهداری خود دارد، منبع R_0 را نیز درخواست کند، از آن‌جا که منبع R_0 در اختیار فرآیند P_0 می‌باشد، فرآیند P_1 نیز در صف انتظار، می‌خوابد (نگهداری و انتظار). این سرنوشت‌های به هم گره خورده، در ادامه بن‌بست را به ارمغان می‌آورد.

توجه: از دو توجه فوق این نتیجه استنباط می‌گردد که در شرایط ناامن، **احتمال وقوع بن‌بست** وجود دارد. در واقع در شرایط ناامن وقوع بن‌بست و عدم بن‌بست به رفتار فرآیندها در آینده، بستگی دارد. یعنی فرآیندها رفتار نگهداری و درخواست را در پیش گیرند و یا رفتار رهاسازی و درخواست را.

khalilifar.ir

تست‌های فصل هشتم: مدیریت دیسک

۱- یک دیسک را در نظر بگیرید که شامل 100 سیلندر است (0 تا 99). زمان لازم برای عبور هد از یک سیلندر به سیلندر مجاور یک واحد زمانی است. در زمان صفر هد بر روی سیلندر صفر است و درخواستی از گذشته وجود ندارد. شش درخواست در زمان‌های مختلف مطابق جدول زیر وارد می‌شوند.

زمان ورود درخواست: 0 10 20 70 80 90

سیلندر درخواست شده: 21 75 16 68 2 17

در زمان حرکت هد به سمت یک سیلندر، ورود درخواست جدید تأثیری بر حرکت ندارد. ترتیب اجرای درخواست‌ها برای الگوریتم SCAN (یا آسانسور) چیست؟ (مهندسی کامپیوتر – دولتی۹۰)

۱) 0 21 75 16 68 2 17

۲) 0 21 75 16 17 2 68

۳) 0 21 75 16 2 17 68

۴) 0 21 75 68 16 2 17

۲- فرض کنید در سیستمی، مدیریت دیسک یکبار از زمان‌بندی SSTF (کوچک‌ترین زمان دستیابی اول) و یکبار از FIFO (به ترتیب درخواست) استفاده کند. در صورتی که جابجایی بین هر دو شیار مجاور زمانی ثابت (2ms) طول بکشد و نوک خواندن - نوشتن روی شیار ۴۰ قرار داشته باشد، زمان جابجایی بین شیارها برای سرویس‌دهی به درخواست‌های زیر در هر دو زمان‌بندی FIFO و SSTF به ترتیب چند میلی‌ثانیه است و کدام زمان‌بندی بهتر عمل می‌کند؟ ترتیب درخواست‌ها برای شیارها (از راست به چپ)، 97, 100, 55, 35, 5, 14, 7, 44, 41 است.

(مهندسی IT- دولتی۹۰)

۱) 310، 260 و زمان‌بندی SSTF از FIFO بهتر عمل می‌کند.

۲) 155، 178 و زمان‌بندی FIFO از SSTF بهتر عمل می‌کند.

۳) 310، 310 و SSTF مشابه FIFO عمل می‌کند.

۴) 310، 356 و زمان‌بندی FIFO از SSTF بهتر عمل می‌کند.

۳- اگر هد (head) دیسک روی سیلندر 53 دیسک قرار داشته باشد و به سمت سیلندر صفر در حرکت باشد و به ترتیب از سمت چپ به راست سیلندرهای 67، 65، 124، 14، 122، 37، 183، 98 درخواست شده باشند. الگوریتم برنامه‌ریزی C-SCAN به کدام ترتیب سیلندرهای درخواست شده را بررسی می‌نماید؟ (مهندسی کامپیوتر – دولتی۹۳)

۱) 183، 124، 122، 98، 14، 37، 67، 65

۲) 65، 67، 98، 122، 124، 183، 14، 37

۳) 14، 37، 183، 124، 122، 98، 67، 65

۴) 183، 124، 122، 98، 67، 65، 14، 37

---

khalilifar.ir

۴- دیسکی با شیارهای 0 تا 225 را در نظر بگیرید که در آن بازوی دیسک در شیار 45 باشد و در جهت افزایش شماره شیارها حرکت می‌کند. در این دیسک درخواست شیار به ترتیب از راست به چپ 40، 67، 11، 240، 87 می‌رسند. کدام گزینه درباره روش و تعداد شیارهایی که بازوی دیسک طی می‌کند به ترتیب درست است؟ (مهندسی کامپیوتر- دولتی ۹۵)

۱) c-scan، 453 ۲) c-look، 454 ۳) scan، 460 ۴) look، 470

۵- درخواست‌های مربوط به دیسک که توسط درایور دیسک دریافت شده‌اند، به ترتیب برای سیلندرهای 5، 25، 18، 3، 39، 8 و 35 هستند. زمان پیگرد (seek) برای حرکت بازو از یک سیلندر به سیلندر دیگر (سیلندرهای متوالی) برابر با 5 میلی ثانیه است. چه مدت زمان نیاز است تا این درخواست‌ها توسط الگوریتم SSF (کوتاه‌ترین زمان پیگرد، اول) پاسخ داده شوند؟ فرض می‌شود که هنگامی که آخرین درخواست مطرح شد و هنوز هیچکدام از درخواست‌ها پاسخ داده نشده‌اند، محل بازو بر روی سیلندر 20 است. (مهندسی IT – دولتی ۹۵)

۱) 125 ۲) 295 ۳) 575 ۴) 750

۶- فرض کنید دیسکی دارای 100 سیلندر است (سیلندر 0 تا 99) و رفتن هِد خواندن و نوشتن از یک سیلندر به سیلندر مجاور، یک واحد زمان طول می‌کشد. در زمان صفر، هِد در سیلندر صفر قرار دارد و هیچ درخواستی موجود نیست. شش درخواست در زمان‌هایی که در جدول نشان داده است مطرح می‌شوند. اگر از الگوریتم آسانسور استفاده شود، زمان لازم برای پاسخ دادن به همه درخواست‌ها چقدر است؟ (در زمان‌هایی که هِد در حال حرکت است، رسیدن یک درخواست باعث تغییر مقصد هد در آن حرکت نمی‌شود. از زمان گردش دیسک هم صرف‌نظر کنید.)

(مهندسی کامپیوتر – دولتی ۹۶)

زمان	شماره سیلندر
0	21
10	75
20	16
70	68
80	2
90	17

۱) 163 ۲) 172 ۳) 199 ۴) 200

---

khalilifar.ir

پاسخ تست‌های فصل هشتم: مدیریت دیسک

۱- گزینه (۴) صحیح است.

توجه: مطابق فرض سؤال، در زمان صفر هد بر روی سیلندر صفر است و درخواستی از گذشته وجود ندارد.

شش درخواست در زمان‌های مختلف مطابق جدول زیر وارد می شوند:

زمان ورود درخواست:	0	10	20	70	80	90
سیلندر درخواست شده:	21	75	16	68	2	17

فرض سؤال: در زمان حرکت هد به سمت یک سیلندر، ورود درخواست جدید تأثیری بر حرکت ندارد.

توجه: هد دیسک تحت کنترل نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک قرار دارد، و به خودی خود هد دیسک فهم و شعور ندارد، یعنی بدون نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک، هد دیسک نمی‌داند کجا برود و کجا نرود، چه کند و چه نکند. بلکه نرم‌افزار زمان‌بندی دیسک به واسطه استفاده از یک الگوریتم خاص، توسط کنترل‌کننده هد دیسک، هد دیسک را کنترل و جابه‌جا می‌کند، در واقع این نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک است که به هد دیسک فهم و شعور می‌دهد و به هد می‌گوید که کجا برود و کجا نرود، چه کند و چه نکند. بنابراین در زمان حرکت مکانیکی هد به سمت یک سیلندر، هد شخصاً تصمیم جدیدی نمی‌گیرد، بلکه تابع تصمیمات نرم‌افزار زمان‌بندی و کنترل‌کننده خود است، که فرض سؤال این است که ورود درخواست جدید تأثیری بر حرکت هد ندارد که این خواست نرم‌افزار زمان‌بندی دیسک برای هد دیسک است.

توجه: در الگوریتم SCAN، اگر درخواستی به صف برسد که جلوی هد باشد، این درخواست سریعاً سرویس داده می‌شود ولی اگر درخواست پشت سر هد باشد، بایستی صبر شود، تا هد به انتهای دیسک رفته، تغییر جهت داده و برگردد. لذا در این مسأله در ابتدا هد به سیلندر 21 رفته و سپس با توجه به درخواست‌های وارد شده تا زمان 21 (یعنی درخواست‌های 75 و 16) به سیلندر 75 می‌رود. (جهت حرکت فعلاً صعودی است).

توجه: درصورت سؤال مطرح شده است که در زمان حرکت هد به سمت یک سیلندر، ورود درخواست جدید تأثیری بر حرکت ندارد، یعنی ابتدا شخص بدون توجه به حواشی، وارد یک جزیره می‌شود، به جزیره که رسید، دعوت‌های خود را بررسی می‌کند، سپس دعوتی را قبول می‌کند که در جهت حرکتش باشد، سپس مجدداً بدون توجه به حواشی، وارد جزیره بعدی می‌شود و این روال تا انتها به همین منوال ادامه می‌یابد.

توجه: در صورت سؤال بیان شده است که زمان لازم برای ورود هد از یک سیلندر به سیلندر مجاور یک واحد زمانی است. بنابراین هنگامی که در سیلندر 21 قرار داریم، به معنی حضور در

khalilifar.ir

زمان 21 است. بنابراین واضح است که تا قبل از زمان 21 درخواست سیلندر 75 در زمان 10 و درخواست سیلندر 16 در زمان 20 صادر شده است. بنابراین ابتدا وارد جزیره (سیلندر) 21 شدیم، سپس وارد جزیره (سیلندر) 75 شدیم، حال که در سیلندر 75 قرار داریم، به معنی حضور در زمان 75 است.

بنابراین واضح است که تا قبل از زمان 75، درخواست سیلندر 68 در زمان 70 صادر شده است. بنابراین در لحظه 75، درخواست‌های 16 و 68 در سیستم وجود دارند، ولی با توجه به اینکه درخواست‌ها در پشت سر هد قرار دارند، در این لحظه به هیچ یک سرویس داده نمی شوند و هد با توجه به الگوریتم SCAN تا انتهای دیسک (سیلندر 99) حرکت کرده و جهت آن برعکس می‌شود. حال که در سیلندر 99 قرار داریم، به معنی حضور در زمان 99 است. بنابراین واضح است که تا قبل از زمان 99، درخواست سیلندر 2 در زمان 80 و درخواست سیلندر 17 نیز در زمان 90 صادر شده است. بنابراین در لحظه 99، درخواست‌های 16، 68، 2 و 17 در سیستم وجود دارند، حال جهت حرکت هد به شکل نزولی از سیلندر 99 به سیلندر صفر تغییر می‌کند و در این پیمایش و حرکت به ترتیب به درخواست‌های 68، 17، 16 و 2 سرویس داده می‌شود، یعنی داریم:

$$0 \rightarrow 21 \rightarrow 75 \rightarrow 68 \rightarrow 17 \rightarrow 16 \rightarrow 2$$

که در هیچ کدام از گزینه‌ها این ترتیب وجود ندارد. اما اگر فرض کنیم که در الگوریتم SCAN هد تا جایی حرکت می‌کند که درخواستی وجود داشته باشد و سپس جهت آن بالعکس شود (یعنی الگوریتم Look) داریم:

$$0 \rightarrow 21 \rightarrow 75 \rightarrow 68 \rightarrow 16 \rightarrow 2 \rightarrow 17$$

توجه کنید هنگامی که هد به سیلندر 75 می‌رسد جهت آن پس از آن بالعکس و در جهت نزولی می‌شود چون درخواست‌های موجود 16 و 68 است. در واقع حضور در سیلندر 75 به معنی حضور در زمان 75، درخواست سیلندر با شماره بیشتر وجود ندارد، بلکه سیلندرهای 16 و 68 با شماره کمتر وجود دارد. بنابراین در حرکت نزولی از سیلندر 75 ازبین درخواست‌های موجود 16 و 68، ابتدا درخواست 68 پاسخ داده می‌شود. هنگامی که در سیلندر 68 قرار می‌گیریم، در زمان $(75+(\overset{7}{\cancel{75}}-68)) = 82$ نیز حضور داریم.

بنابراین هنگامی که در سیلندر 68 قرار داریم، به معنی حضور در زمان 82 است. بنابراین واضح است که تا قبل از زمان 82، درخواست سیلندر 2 در زمان 80 صادر شده است. بنابراین در لحظه 82، درخواست‌های 16 و 2 در سیستم وجود دارند. بنابراین در ادامه حرکت نزولی از سیلندر 68 به سیلندر 16 می‌رویم. هنگامی که در سیلندر 16 قرار می‌گیریم، در زمان $(82+(\overset{52}{\cancel{68}}-16)) = 134$ نیز حضور داریم. بنابراین هنگامی که در سیلندر 16 قرار داریم، به معنی حضور در زمان 134 است. بنابراین واضح است که تا قبل از زمان 134، درخواست سیلندر 17 نیز در زمان 90 صادر

khalilifar.ir

شده است. بنابراین در لحظه 134، درخواست‌های 2 و 17 در سیستم وجود دارند. بنابراین در ادامه حرکت نزولی از سیلندر 16 به سیلندر 2 می‌رویم. هنگامی که در سیلندر 2 قرار می‌گیریم، در زمان $148 = ((\cancel{16}^{14} - 2) + 134)$ نیز حضور داریم. بنابراین هنگامی که در سیلندر 2 قرار داریم، به معنی حضور در زمان 148 است. در این زمان همه درخواست‌ها صادر شده‌اند و درخواست جدید دیگری وجود ندارد و تنها درخواست باقی مانده از قبل، درخواست سیلندر 17 می‌باشد. توجه کنید هنگامی که هد به سیلندر 2 می‌رسد جهت آن پس از آن بالعکس و در جهت صعودی می‌شود چون تنها درخواست موجود سیلندر 17 است. در واقع هنگام حضور در سیلندر 2 به معنی حضور در زمان 148، درخواست سیلندر با شماره کمتر وجود ندارد، بلکه سیلندر 17 با شماره بیشتر وجود دارد. بنابراین در حرکت صعودی از سیلندر 2 به سیلندر 17 حرکت می‌کنیم. تا به درخواست سیلندر 17 پاسخ داده شود. هنگامی که در سیلندر 17 قرار می‌گیریم، در زمان $163 = ((\cancel{17}^{15} - 2) + 148)$ نیز حضور داریم. بنابراین هنگامی که در سیلندر 17 قرار داریم، به معنی حضور در زمان 163 است و کار تمام می‌شود، بنابر روال مطرح شده داریم:

$$0 \rightarrow 21 \rightarrow 75 \rightarrow 68 \rightarrow 16 \rightarrow 2 \rightarrow 17$$

روال فوق با گزینه چهارم تطابق دارد، بنابراین گزینه چهارم درست خواهد بود.

توجه: البته بهتر بود طراح محترم الگوریتم Look را مورد پرسش قرار می‌داد و نه SCAN که بعد الگوریتم Look در گزینه‌ها باشد.

۲- گزینه (۴) صحیح است.

زمان‌بندی FIFO :

$$40 \xrightarrow{1} 41 \xrightarrow{3} 44 \xrightarrow{37} 7 \xrightarrow{7} 14 \xrightarrow{9} 5 \xrightarrow{30} 35 \xrightarrow{20} 55 \xrightarrow{45} 100 \xrightarrow{3} 97$$

مجموع جابجایی بین شیارها با زمان‌بندی FIFO: $1+3+37+7+9+30+20+45+3=155$

زمان کل جابجایی بین شیارها با زمان‌بندی FIFO: $155 \times 2 = 310\,ms$

زمان‌بندی SSTF (Shortest Seek Time First) :

$$40 \xrightarrow{1} 41 \xrightarrow{3} 44 \xrightarrow{9} 35 \xrightarrow{20} 55 \xrightarrow{41} 14 \xrightarrow{7} 7 \xrightarrow{2} 5 \xrightarrow{92} 97 \xrightarrow{3} 100$$

مجموع جابجایی بین شیارها با زمان‌بندی SSTF : $1+3+9+20+41+7+2+92+3=178$

زمان کل جابجایی بین شیارها با زمان‌بندی SSTF : $178 \times 2 = 356\,ms$

با آنکه اغلب SSTF بهتر از FIFO عمل می‌کند ولی الزاماً اینگونه نیست و در این مسأله مشاهده می‌شود که FIFO بهتر از SSTF است.

---

khalilifar.ir

۳- گزینه (۲) صحیح است.

الگوریتم C-SCAN همانند روش SCAN می‌باشد، اما هد فقط در یک جهت حرکت به درخواست‌ها سرویس می‌دهد (در واقع در مسیر برگشت به هیچ درخواستی سرویس نمی‌دهد). در این روش، هد در یک جهت درخواست‌ها را سرویس داده و وقتی به انتهای دیسک رسید، سریعاً به ابتدای دیسک برگشته و دوباره در همان جهت شروع به سرویس دادن می‌کند. با این روش نیازهای درونی، بیرونی و میانی به یک اندازه انتظار می‌کشند.

توجه: مطابق صورت سؤال، هد روی سیلندر 53 دیسک قرار دارد و به سمت سیلندر صفر در حال حرکت است. بنابراین داریم:

$$53 \rightarrow 37 \rightarrow 14 \rightarrow 183 \rightarrow 124 \rightarrow 122 \rightarrow 98 \rightarrow 67 \rightarrow 65$$

توجه: گزینه چهارم، SCAN معمولی است.

۴- گزینه (۱) صحیح است.

مطابق صورت سؤال دیسکی با شیارهای 0 تا 225 وجود دارد، همچنین مطابق فرض صورت سؤال، درخواست شیار به ترتیب از راست به چپ 40، 67، 11، 240، 87 وجود دارد که یکی از درخواست‌ها از حداکثر شماره شیار یعنی 225 بزرگتر است (240>225)، بنابراین اگر بخواهیم آخرین شیار را 225 در نظر بگیریم پاسخ به درخواست 240 امکان‌پذیر نیست، و همچنین اگر بخواهیم درخواست 240 را پاسخ دهیم، آنگاه آخرین شیار مشخص نیست، به هر شکل مطابق مفروضات صورت سؤال، سؤال قابل حل نیست.

توجه: سازمان سنجش آموزش کشور در کلید اولیه خود ابتدا گزینه اول را به عنوان پاسخ اعلام نمود، سپس در کلید نهایی مجدداً گزینه اول را با تأثیر مثبت به عنوان پاسخ اعلام کرد که باز هم پاسخ درست نبود.

۵- گزینه (۲) صحیح است.

در الگوریتم (SFF: Shortest Seek Time)، درخواستی انتخاب می‌شود که بتوان آن را با حداقل حرکت هد، پاسخ داد. در واقع تقاضای با حداقل زمان جستجو برآورده خواهد شد.

مطابق فرض سؤال دنباله درخواست‌ها به صورت زیر است:

5، 25، 18، 3، 39، 8 و 35

و هد در ابتدای کار بر روی سیلندر 20 قرار دارد.

در این روش ابتدا درخواست 18 سرویس داده می‌شود. زیرا نزدیکترین درخواست به مکان فعلی هد است، سپس درخواست‌های 25، 35، 39، 8، 5 و در نهایت 3 سرویس می‌گیرند.

یعنی داریم:

$$20 \xrightarrow[2]{} 18 \xrightarrow[7]{} 25 \xrightarrow[10]{} 35 \xrightarrow[4]{} 39 \xrightarrow[31]{} 8 \xrightarrow[3]{} 5 \xrightarrow[2]{} 3$$

khalilifar.ir

در نتیجه، تعداد کل حرکات هد به صورت زیر خواهد بود:

$2+7+10+4+31+3+2 = 59$

مطابق فرض سؤال، اگر حرکت از هر سیلندر به سیلندر دیگر (سیلندرهای متوالی) یعنی مجاور 5 میلی‌ثانیه طول بکشد، مجموع زمان جستجو در این درخواست‌ها، به صورت زیر خواهد بود:

$T_{Total} = 59 \times 5 = 295 \text{ ms}$

۶- گزینه ( ) صحیح است.

توجه: مطابق فرض سؤال، در زمان صفر هد بر روی سیلندر صفر است و درخواستی از گذشته وجود ندارد.

شش درخواست در زمان‌های مختلف مطابق جدول زیر وارد می شوند:

زمان ورود درخواست:	0	10	20	70	80	90
سیلندر درخواست شده:	21	75	16	68	2	17

فرض سوال: در زمان حرکت هد به سمت یک سیلندر، ورود درخواست جدید تأثیری بر حرکت ندارد.

توجه: هد دیسک تحت کنترل نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک قرار دارد، و به خودی خود هد دیسک فهم و شعور ندارد، یعنی بدون نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک، هد دیسک نمی‌داند کجا برود و کجا نرود، چه کند و چه نکند. بلکه نرم‌افزار زمان‌بندی دیسک به واسطه استفاده از یک الگوریتم خاص، توسط کنترل‌کننده هد دیسک، هد دیسک را کنترل و جابه‌جا می‌کند، در واقع این نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک است که به هد دیسک فهم و شعور می‌هد و به هد می‌گوید که کجا برود و کجا نرود، چه کند و چه نکند. بنابراین در زمان حرکت مکانیکی هد به سمت یک سیلندر، هد شخصا تصمیم جدیدی نمی‌گیرد، بلکه تابع تصمیمات نرم‌افزار زمان‌بندی و کنترل‌کننده خود است، که فرض سوال این است که ورود درخواست جدید تأثیری بر حرکت هد ندارد که این خواست نرم‌افزار زمان‌بندی دیسک برای هد دیسک است.

توجه: در الگوریتم SCAN، اگر درخواستی به صف برسد که جلوی هد باشد، این درخواست سریعاً سرویس داده می‌شود ولی اگر درخواست پشت سر هد باشد، بایستی صبر شود، تا هد به انتهای دیسک رفته، تغییر جهت داده و برگردد. لذا در این مسأله در ابتدا هد به سیلندر 21 رفته و سپس با توجه به درخواست‌های وارد شده تا زمان 21 (یعنی درخواست‌های 75 و 16) به سیلندر 75 می‌رود. (جهت حرکت فعلاً صعودی است).

توجه: درصورت سوال مطرح شده است که در زمان حرکت هد به سمت یک سیلندر، ورود درخواست جدید تأثیری بر حرکت ندارد، یعنی ابتدا شخص بدون توجه به حواشی، وارد یک جزیره می‌شود، به جزیره که رسید، دعوت‌های خود را بررسی می‌کند، سپس دعوتی را قبول

khalilifar.ir

می‌کند که در جهت حرکتش باشد، سپس مجدداً بدون توجه به حواشی، وارد جزیره بعدی می‌شود و این روال تا انتها به همین منوال ادامه می‌یابد.

توجه: در صورت سوال بیان شده است که زمان لازم برای ورود هد از یک سیلندر به سیلندر مجاور یک واحد زمانی است. بنابراین هنگامی که در سیلندر 21 قرار داریم، به معنی حضور در زمان 21 است. بنابراین واضح است که تا قبل از زمان 21 درخواست سیلندر 75 در زمان 10 و درخواست سیلندر 16 در زمان 20 صادر شده است. بنابراین ابتدا وارد جزیره (سیلندر) 21 شدیم، سپس وارد جزیره (سیلندر) 75 شدیم، حال که در سیلندر 75 قرار داریم، به معنی حضور در زمان 75 است.

بنابراین واضح است که تا قبل از زمان 75، درخواست سیلندر 68 در زمان 70 صادر شده است. بنابراین در لحظه 75، درخواست‌های 16 و 68 در سیستم وجود دارند، ولی با توجه به اینکه درخواست‌ها در پشت سر هد قرار دارند، در این لحظه به هیچ یک سرویس داده نمی شوند و هد با توجه به الگوریتم SCAN تا انتهای دیسک (سیلندر 99) حرکت کرده و جهت آن برعکس می‌شود. حال که در سیلندر 99 قرار داریم، به معنی حضور در زمان 99 است. بنابراین واضح است که تا قبل از زمان 99، درخواست سیلندر 2 در زمان 80 و درخواست سیلندر 17 نیز در زمان 90 صادر شده است. بنابراین در لحظه 99، درخواست‌های 16، 68، 2 و 17 در سیستم وجود دارند، حال جهت حرکت هد به شکل نزولی از سیلندر 99 به سیلندر صفر تغییر می‌کند و در این پیمایش و حرکت به ترتیب به درخواست‌های 68، 17، 16 و 2 سرویس داده می‌شود، یعنی داریم:

$$0 \rightarrow 21 \rightarrow 75 \rightarrow 68 \rightarrow 17 \rightarrow 16 \rightarrow 2$$

در نتیجه، تعداد کل حرکات هد به صورت زیر خواهد بود:

$$21 + 54 + 7 + 51 + 1 + 14 = 148$$

مطابق فرض سوال، اگر حرکت از هر سیلندر به سیلندر دیگر (سیلندرهای متوالی) یعنی مجاور یک واحد زمانی طول بکشد، مجموع زمان جستجو در این درخواست‌ها، به صورت زیر خواهد بود:

$$T_{Total} = 148 \times 1 = 148$$

که در هیچ کدام از گزینه‌ها این ترتیب و به تبع زمان پاسخ به همه درخواست‌ها وجود ندارد. اما اگر فرض کنیم که در الگوریتم SCAN هد تا جایی حرکت می‌کند که درخواستی وجود داشته باشد و سپس جهت آن بالعکس شود (یعنی الگوریتم Look) داریم:

$$0 \rightarrow 21 \rightarrow 75 \rightarrow 68 \rightarrow 16 \rightarrow 2 \rightarrow 17$$

توجه کنید هنگامی که هد به سیلندر 75 می‌رسد جهت آن پس از آن بالعکس و در جهت نزولی می‌شود چون درخواست‌های موجود 16 و 68 است. در واقع حضور در سیلندر 75 به معنی

khalilifar.ir

حضور در زمان 75، درخواست سیلندر با شماره بیشتر وجود ندارد، بلکه سیلندرهای 16 و 68 با شماره کمتر وجود دارد. بنابراین در حرکت نزولی از سیلندر 75 ازبین درخواست‌های موجود 16 و 68، ابتدا درخواست 68 پاسخ داده می‌شود. هنگامی که در سیلندر 68 قرار می‌گیریم، در زمان $(75+(\overset{7}{\cancel{75}-68}))=82$ نیز حضور داریم.

بنابراین هنگامی که در سیلندر 68 قرار داریم، به معنی حضور در زمان 82 است. بنابراین واضح است که تا قبل از زمان 82، درخواست سیلندر 2 در زمان 80 صادر شده است. بنابراین در لحظه 82، درخواست‌های 16 و 2 در سیستم وجود دارند. بنابراین در ادامه حرکت نزولی از سیلندر 68 به سیلندر 16 می‌رویم. هنگامی که در سیلندر 16 قرار می‌گیریم، در زمان $(82+(\overset{52}{\cancel{68}-16}))=134$ نیز حضور داریم. بنابراین هنگامی که در سیلندر 16 قرار داریم، به معنی حضور در زمان 134 است. بنابراین واضح است که تا قبل از زمان 134، درخواست سیلندر 17 نیز در زمان 90 صادر شده است. بنابراین در لحظه 134، درخواست‌های 2 و 17 در سیستم وجود دارند. بنابراین در ادامه حرکت نزولی از سیلندر 16 به سیلندر 2 می‌رویم. هنگامی که در سیلندر 2 قرار می‌گیریم، در زمان $(134+(\overset{14}{\cancel{16}-2}))=148$ نیز حضور داریم. بنابراین هنگامی که در سیلندر 2 قرار داریم، به معنی حضور در زمان 148 است. در این زمان همه درخواست‌ها صادر شده‌اند و درخواست جدید دیگری وجود ندارد و تنها درخواست باقی مانده از قبل، درخواست سیلندر 17 می‌باشد. توجه کنید هنگامی که هد به سیلندر 2 می‌رسد جهت آن پس از آن بالعکس و در جهت صعودی می‌شود چون تنها درخواست موجود سیلندر 17 است. در واقع هنگام حضور در سیلندر 2 به معنی حضور در زمان 148، درخواست سیلندر با شماره کمتر وجود ندارد، بلکه سیلندر 17 با شماره بیشتر وجود دارد. بنابراین در حرکت صعودی از سیلندر 2 به سیلندر 17 حرکت می‌کنیم. تا به درخواست سیلندر 17 پاسخ داده شود. هنگامی که در سیلندر 17 قرار می‌گیریم، در زمان $(148+(\overset{15}{\cancel{17}-2}))=163$ نیز حضور داریم. بنابراین هنگامی که در سیلندر 17 قرار داریم، به معنی حضور در زمان 163 است و کار تمام می‌شود، بنابر روال مطرح شده داریم:

$$0 \rightarrow 21 \rightarrow 75 \rightarrow 68 \rightarrow 16 \rightarrow 2 \rightarrow 17$$

در نتیجه، تعداد کل حرکات هد به صورت زیر خواهد بود:

$$21+54+7+52+14+15=163$$

مطابق فرض سوال، اگر حرکت از هر سیلندر به سیلندر دیگر(سیلندرهای متوالی) یعنی مجاور یک واحد زمانی طول بکشد، مجموع زمان جستجو در این درخواست‌ها، به صورت زیر خواهد بود:

$$T_{Total}=163\times 1=163$$

روال فوق با گزینه اول تطابق دارد، بنابراین گزینه اول می‌تواند درست باشد.

khalilifar.ir

توجه: البته بهتر بود طراح محترم الگوریتم Look را مورد پرسش قرار می‌داد و نه SCAN که بعد الگوریتم Look در گزینه‌ها باشد.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه اول را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سؤال حذف گردید، که کار درستی بوده است.

khalilifar.ir

khalilifar.ir

تست‌های کنکور کارشناسی ارشد سال ۱۳۹۷

۱- کدام عبارت درست‌تر است؟ (مهندسی کامپیوتر – دولتی ۹۷)

۱) Virtualization به شبیه‌سازی یک سیستم عامل وابسته به سخت‌افزار، بر روی یک سیستم عامل وابسته به سخت‌افزار دیگر اطلاق می‌شود.

۲) NUMA یک حافظه توزیع شده است که در آن هر پردازنده یا هسته، به بخش‌های مختلف اختصاصی دسترسی دارد.

۳) System Daemon یک برنامه سیستمی مقیم در حافظه است که در صورت لزوم به صورت ناهمگام اجرا می‌شود.

۴) Emulation به فرآیند شبیه‌سازی یک سیستم عامل داخل سیستم عامل دیگر اطلاق می‌شود.

۲- سیستمی با ترجمه آدرس دو- سطحی و اندازه هر صفحه 4 کیلوبابت در نظر بگیرید. اگر اندازه هر مدخل جدول صفحه برابر 2 بایت (شامل اطلاعات ترجمه و دیگر اطلاعات کنترلی لازم) باشد، چه تعداد فضای بیتی به ترتیب (از راست به چپ) برای جا به جایی (Offset)، اندیس به جدول صفحه اول و اندیس به جدول صفحه دوم برای آدرس مجازی (Virtual Address) 32-بیتی لازم است؟ (مهندسی کامپیوتر – دولتی ۹۷)

۱) 10, 10, 12 ۲) 9, 11, 12 ۳) 12, 10, 10 ۴) 12, 11, 9

۳- در یک سیستم متشکل از 4 قاب که در ابتدا خالی هستند، رشته دستیابی به قاب‌ها را به ترتیب از چپ به راست 1, 2, 3, 4, 5, 6, 2, 3, 2, 1, 6, 7 در نظر بگیرید. اگر سیستم صفحه‌بندی تماما مبتنی بر درخواست (pure demand paging) باشد. در صورت استفاده از الگوریتم‌های FIFO و LRU به ترتیب (از راست به چپ) تعداد نقص صفحه (page fault)، کدام است؟

(مهندسی کامپیوتر – دولتی ۹۷)

۱) 10, 10 ۲) 9, 10 ۳) 6, 6 ۴) 5, 6

۴- کدام روش بیشترین تسریع را در اجرای یک برنامه ایجاد می‌کند؟ (مهندسی IT – دولتی ۹۷)

۱) 20 درصد برنامه 60 درصد تسریع یابد.

۲) 30 درصد برنامه 50 درصد تسریع یابد.

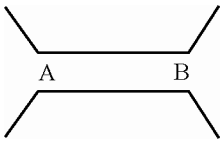
۳) 50 درصد برنامه 30 درصد تسریع یابد.

۴) 40 درصد برنامه 4 برابر تسریع یابد.

---

khalilifar.ir

۵- مسئله پل را در نظر بگیرید که هر خودرو برای عبور باید از هر دو سر پل به ترتیب عبور کند. اگر تعداد زیادی خودرو در هر دو طرف این پل برای عبور از آن وجود داشته باشد و در هر لحظه امکان عبور فقط یک خودرو باشد. کدام مورد درست‌تر است؟ (مهندسی IT – دولتی ۹۷)



۱) این مسئله فقط با سمافور قابل حل است.
۲) این مسئله فقط با مانیتور قابل حل است.
۳) این مسئله نه با سمافور و نه با مانیتور قابل حل است.
۴) این مسئله هم با سمافور و هم با مانیتور قابل حل است.

۶- در یک سیستم بی‌درنگ، سه فرایند به صورت زیر به طور متناوب تولید می‌شود. در این نمایش، 35^* بدین معنی است که فرایند P_2 پس از 35 واحد زمانی از لحظه ایجاد، دوباره تکرار می‌شود، موعد (Deadline) آن برابر 100 (35+65) است. در صورتی که از الگوریتم غیرقبضه‌ای «اول زودترین موعد» (Earliest-deadline first) برای زمان‌بندی استفاده شود، اگر فرآیند P_4 در زمان صفر با دوره تناوب 100 واحد زمانی ایجاد شود، برای آنکه زمان‌بندی سیستم امکان‌پذیر باشد، حداکثر مقدار Burst Time آن کدام است؟ (مهندسی IT – دولتی ۹۷)

Process	Arrival	Burst Time	Period/deadline
P_1	@ 0	20	50
P_2	@ 15	25	65+35*
P_3	@ 25	20	90

۱) 5 ۲) 10 ۳) 15 ۴) مقداری وجود ندارد.

۷- در سیستم‌های تعبیه شده بی‌درنگ سخت که پاسخ در زمانی مشخص باید تضمین شود، کدام روش نگاشت ریسمان‌های کاربر به ریسمان‌های سیستمی، مناسب است؟ (مهندسی IT – دولتی ۹۷)

۱) یک به یک ۲) چند به یک ۳) چند به چند ۴) دوسطحی

۸- یک دیسک سخت متشکل از 200 شیار که از 0 تا 199 شماره‌گذاری شده است و سر بازوی آن روی شیار 50 و در جهت افزایش شیار است را در نظر بگیرید. اگر درخواست پیمایش شیارهای زیر به ترتیب (از چپ به راست) باشد، و الگوریتم C-LOOK برای زمان‌بندی استفاده شده باشد، تعداد کل شیار پیمایش شده کدام است؟ (مهندسی IT – دولتی ۹۷)

95,180,34,119,11,123,62,64

۱) 299 ۲) 322 ۳) 337 ۴) 382

---

khalilifar.ir

۹- وضعیت موجود در یک سیستم به صورت جدول زیر است. در دو حالت ۱) وضعیت جاری و ۲) وضعیتی که فرآیند P_1 درخواست منابع بیشتر به صورت $(A = 0, B = 5, C = 2)$ بدهد، کدام مورد از نظر امن بودن به ترتیب (از راست به چپ)، درست است؟ (مهندسی IT – دولتی ۹۷)

	Max	Allocation	Available
	A B C	A B C	A B C
P_0	0 0 1	0 0 1	
P_1	1 7 5	1 0 0	
P_2	2 3 5	1 3 5	
P_3	0 6 5	0 6 3	
Total		2 9 9	1 5 2

۱) امن، امن ۲) امن، ناامن ۳) ناامن، امن ۴) ناامن، ناامن

۱۰- حافظه‌ای با بخش‌های (partitions) 100، 400، 200 و 500 کیلوبایت را در نظر بگیرید. اگر از روش بدترین تطابق (Worst Fit) برای اختصاص فضای حافظه به فرایندها استفاده شود، مشخص کنید فرایندهای با اندازه های زیر به ترتیب (از راست به چپ) در کدام بخش از حافظه قرار می‌گیرد؟ (مهندسی IT – دولتی ۹۷)

فرایند	P_1	P_2	P_3	P_4
اندازه (کیلوبایت)	200	395	100	254

۱) 400-100-500-200 ۲) 500-100-400-200

۳) 500-400-500- فضا وجود ندارد ۴) 500-400-200- فضا وجود ندارد

khalilifar.ir

پاسخ تست‌های کنکور کارشناسی ارشد سال ۱۳۹۷

۱- گزینه (۳) صحیح است.

صورت سوال به این شکل است:

عبارت درست‌تر است؟

۱) Virtualization به شبیه‌سازی یک سیستم عامل وابسته به سخت‌افزار، بر روی یک سیستم عامل وابسته به سخت‌افزار دیگر اطلاق می‌شود.

گزینه اول پاسخ سوال نیست. زیرا مجازی‌سازی را می‌توان تکنولوژی استفاده از نرم‌افزار به جای سخت‌افزار دانست.

از مجازی‌سازی برای ایجاد کردن چندین سرور مجازی بر روی یک سخت‌افزار استفاده می‌شود. بعنوان مثال شما می‌خواهید از نرم‌افزارهایی استفاده کنید که فقط بر Windows XP اجرا می‌شود ولی سیستم عامل نصب شده بر روی سیستم شما Windows 7 است و قصد پاک کردن آن را هم ندارید در چنین شرایطی کاربر باید از مجازی‌سازی استفاده کند و هر دوی سیستم عامل‌ها را به روی یک سیستم فیزیکی نصب و اجرا کنند. و یا اینکه شما مدیر شبکه یک سازمان هستید و قصد گسترش ابعاد شبکه سازمان را دارید به طور حتم الویت شما فراهم کردن سرورهای بیشتر است که بسیار مشکل و هزینه بر خواهد بود اما با بهرگیری از مجازی‌سازی همه‌ی محدودیت‌ها و موانع از سر راه شما برداشته می‌شود و شما می‌توانید به راحتی و با کمترین هزینه شبکه سازمان گسترش دهید. همانطور که قبلا گفتیم از مجازی‌سازی برای ایجاد چندین سیستم مجازی استفاده می‌شود تا بتوانیم رفع نیاز کنیم. اگر قصد بررسی مجازی‌سازی را در لایه‌های معماری داشته باشیم به ترتیب در اولین لایه دستگاه SAN Storage یا ذخیره‌سازی اطلاعات قرار خواهد گرفت و در لایه بعدی سخت‌افزارها و سیستم‌ها تامین منابع قرار خواهند گرفت و در آخر سرور میزبان قرار می‌گیرد. از جمله برنامه‌ها و نرم‌افزارهای پرکاربرد مجازی‌سازی می‌توان از Vmware، Microsoft Hyper-V، Sun xVM و Citrix نام برد. در واقع این برنامه‌ها همان سرور میزبان (Host Server) در لایه آخر معماری هستند که سیستم مورد نیاز ما را در خود نگهداری می‌کنند. با استفاده از این روش می توان از منابع سخت افزاری موجود به نحو احسنت استفاده کرد و از بروز بسیاری از مشکلات و هزینه ها پیشگیری کرد.

تکنولوژی مجازی سازی شامل سه شاخه اصلی می‌باشد:

۱- Server virtualization

این تکنولوژی که با نام‌های دیگری مثل Hardware virtualization و OS Virtualization نیز شناخته می‌شود، دو ویژگی زیر را برای ما مهیا می‌نماید:

khalilifar.ir

الف) به جای راه اندازی تعداد زیادی سرور سخت‌افزاری در شبکه خود که مستلزم هزینه زیاد، هزینه و زحمات نگهداری بالا و مشکلات بسیار زیادی می‌باشد، یک یا تعداد بسیار کمتری سرور فیزیکی مناسب (با مشخصات سخت افزاری بالا) راه‌اندازی می‌نماییم و بر روی این سرورهای فیزیکی، سیستم‌عامل‌های مورد نیاز را به صورت ماشین‌های مجازی (virtual machine) راه‌اندازی می‌نماییم.

توجه: این ماشین‌های مجازی در حقیقت نسخه شبیه‌سازی شده نرم‌افزاری سیستم‌های سخت افزاری هستند.

ب) با استفاده از این تکنولوژی، بر روی هر یک از سرورهای فیزیکی تعداد زیادی سیستم‌عامل همزمان و مستقل از یکدیگر سرویس می‌دهند و اگر اختلال یا مشکلی در هر یک از این سیستم عامل‌ها روی دهد در دیگری تاثیری نمی‌گذارد.

به دلیل استفاده همزمان چندین سیستم عامل از سخت‌افزار یک سرور، این تکنولوژی «استفاده بهینه از سخت افزار» را برای ما مهیا می‌نماید.

۲- Application virtualization

مواقع زیادی این نیاز پیش می‌آید که که برای برخی از کاربران سازمان، امکان استفاده از نرم‌افزاهای سنگینی مثل photoshop ، Autocad و غیره فراهم نماییم، اما سیستم فعلی آنها مشخصات سخت افزاری ضعیفی دارد و امکان نصب و استفاده از این نرم‌افزارهای سنگین در سیستم آنها وجود ندارد.

با راه اندازی تکنولوژی Application virtualization که یکی از سرویس‌های معروف مایکروسافتی آن سرویسی به نام RemoteApp می‌باشد، این نرم افزارها بر روی یک یا تعداد کمی سرور با مشخصات سخت افزاری بالا به روشی خاص نصب می‌کنیم و سپس نوع خاصی فایل shortcut از آن نرم افزار ایجاد کرده و آنها را در اختیار آن کاربران قرار می‌دهیم.

هنگامی که کاربران آن فایل shortcut را بر روی سیستم خود اجرا می‌نمایند، همگی به نرم افزاری که بر روی سرور نصب شده است متصل می‌شوند و در حقیقت آن نرم افزار بر روی سرور اجرا می‌شود و کاربر فقط صفحه کار با نرم افزار را بر روی سیستم خود مشاهده می‌کند (دقیقاً شبیه زمانی که نرم‌افزار بر روی سیستم خود کاربر نصب شده بود). این تکنولوژی این قابلیت را ارایه می‌کند که نرم‌افزار همزمان چندین بار بر روی سرور اجرا شود.

درهنگام ذخیره فایل خروجی، کاربر می‌تواند آن فایل را بر روی سیستم محلی خود و یا در درایوهای سرور دخیره نماید.

khalilifar.ir

۳- Desktop virtualization

در حالت سنتی، برای هر یک از کاربران سازمان یک کیس سخت‌افزاری مجزا تهیه می‌شود و سیستم عامل و نرم افزارهای مورد نیاز آن کاربر به صورت جداگانه بر روی این سیستم نصب می‌شوند.

ضمناً در صورت نیاز به ارتقا سخت افزار سیستم‌های کاربران در آینده، باید تک تک سیستم‌ها به صورت مجزا ارتقا داده شوند و قطعات سخت‌افزاری مورد نیاز درون کیس آنها بسته شود که این پروسه در تعداد بالا، فرایندی زمان بر و طاقت فرسا می‌باشد.

در تکنولوژی Desktop virtualization، برای کارمندان سازمان نیز virtual machineهایی در روی سرورهای سازمان راه اندازی می‌نماییم و مدیریت سخت افزارها و منابع آنها به صورت متمرکز و با سرعت و سهولت بسیار بیشتر از پشت سرور انجام می‌شود.

۲) NUMA یک حافظه توزیع شده است که در آن هر پردازنده یا هسته، به بخش‌های مختلف اختصاصی دسترسی دارد.

گزینه دوم پاسخ سوال نیست. زیرا Non-uniform Memory Access (به‌صورت مخفف NUMA) یک طراحی حافظه است که در این طرح هر processor (پردازنده) حافظه‌ی محلی مخصوص به خود را دارد و سرعت دسترسی پردازنده به حافظه محلی خود بالاتر از حافظه‌های غیرمحلی است. NUMA (نوما) نقطه مقابل معماری SMP است که در آن تمام پردازنده‌ها از یک حافظه مشترک استفاده می‌کنند، مناسب پروسس‌هایی وابسته به یک کاربر یا task است، در نتیجه اجرای پروسس در یک نود و با یک RAM اختصاصی سرعت اجرا را بیشتر می‌کند. برای اجرای بهینه برنامه‌ها باید هر پروسس تا حد ممکن از RAM محلی پردازنده‌ای که در آن مقیم است استفاده کند.

۳) System Daemon یک برنامه سیستمی مقیم در حافظه است که در صورت لزوم به صورت ناهمگام اجرا می‌شود. گزینه سوم پاسخ سوال است، زیرا daemon یک برنامه کامپیوتری است که بعنوان یک فرآیند در background سیستم عامل اجرا می‌شود بدین معنی که در کنترل مستقیم کاربر نیست و کار خاصی را در زمان مشخص و یا در پاسخ به یک رویداد خاص بصورت تکراری و ناهمگام انجام می دهد. در لینوکس و البته دیگر سیستم عامل‌های چند وظیفه‌ای، مدام با اصطلاحی به نام Daemon مواجه می‌شوید، Daemon در مدیریت سرور لینوکس نقش دارد.

یک daemon فرآیندی است که مدت زمان زیادی در پس زمینه سیستم عامل در حال اجرا است تا به درخواست‌های سرویس‌ها پاسخ دهد. این اصطلاح بیشتر در لینوکس استفاده می‌شود. اما در سیستم عامل‌های دیگر مثل ویندوز و مکینتاش هم وجود دارد. در لینوکس به طور قراردادی در انتهای نام هر Daemon یک کاراکتر d هم وجود دارد. به عنوان مثال inetd ، nfsd، sshd، named و lpd از جمله دیمون‌ها هستند.

پس Daemon یک برنامه است که به عنوان یک فرآیند پشت صحنه اجرا می‌شود و در ارتباط مستقیم با کاربر نیست، یعنی شما آن برنامه را به صورت مستقیم نمی‌بینید. در محیط لینوکس

khalilifar.ir

فرآیند والد یک Daemon اغلب و نه همیشه، یک فرآیند init است. به همین دلیل است که عبارت init را همیشه در کنار Daemon خواهید شنید.

سیستم همیشه Daemon ها را در زمان بالا آمدن اجرا می‌کند تا هر کدام گوش به زنگ باشند تا کاری را انجام دهند. مثل پاسخ به درخواست‌های شبکه، فعالیت‌های سخت‌افزاری و برخی از فعالیت‌ها که مربوط به نرم‌افزارهای خاصی می‌شوند. Daemonها حتی می‌توانند پیکربندی‌های سخت افزاری (udevd) ، اجرای وظایف زمان‌بندی شده (cron) و دسته‌ای از وظایف دیگر را انجام دهند که تمامی آنها در پس زمینه سیستم شما انجام خواهند شد.

بدین ترتیب فهمیدیم که Daemonها برنامه‌های پشت پرده‌ای هستند که توسط خود سیستم در زمان اجرای کامپیوتر راه‌اندازی می‌شوند و هر کدام کاری را برای ماشین انجام می‌دهند.

۴) Emulation به فرآیند شبیه‌سازی یک سیستم عامل داخل سیستم عامل دیگر اطلاق می‌شود.

گزینه چهارم پاسخ سوال نیست. زیرا اگر از نرم‌افزارهای Simulator یا شبیه‌ساز استفاده کرده باشید و یا با ابزارهای Emulator یا تقلید کننده آشنایی داشته باشید. شاید برای شما هم جالب باشد تفاوت بین این دو واژه را بیشتر بدانید. در وهله اول شما باید بدانید که این دو مفهوم با همدیگر تفاوت‌های اساسی دارند و نباید واژه‌های Emulator و Simulator به جای هم استفاده شود. دقت کنید زمانیکه صحبت از Simulator یا شبیه‌ساز می‌شود ما در خصوص یک سیستم صحبت می‌کنیم که هم می‌تواند نرم‌افزار و هم می‌تواند سخت‌افزاری باشد و این سیستم رفتارهایی بسیار نزدیک و شبیه به سیستم واقعی را دارد و در زمان استفاده از این Simulator شما تصور می کنید که در حال استفاده از سیستم اصلی هستید. اما نحوه پیاده‌سازی Simulator ها کاملا متفاوت است، در واقع شبیه‌سازها یا Simulator ها دقیقا از قوانین و رفتارهایی که سیستم واقعی دارد پیروی نمی‌کنند و برای خودشان قوانینی دارند که ممکن است به هیچ عنوان در سیستم واقعی شبیه‌سازی نشده اتفاق نیوفتد. در واقع زمانیکه صحبت از Simulator یا شبیه‌ساز می‌شود که شما در خصوص یک سیستم یا بهتر بگوییم در خصوص ایده و روش کارکرد یک سیستم صحبت می کنید و در خصوص جزئیات کامل کار کردن سیستم صحبتی ندارید.

نرم افزارهایی وجود دارند که شبیه‌ساز پرواز با هواپیما هستند، سخت‌افزارهایی هم وجود دارند که همین کار شبیه‌سازی پرواز را انجام می‌دهند، نرم افزارهایی وجود دارند که برای ما شبکه را شبیه‌سازی می کنند. اینگونه نرم‌افزارها یا سخت‌افزارها به شما این احساس را می‌دهند که در حال کار کردن با یک سیستم واقعی هستید و برای مثال یک هواپیما را از زمین بلند می‌کنید و بر روی زمین می‌نشانید یا سویچ‌ها و روترهای شبکه را پیکربندی می‌کنید و بین آنها ارتباط برقرار می‌کنید. اما این محیط‌های شبیه‌سازی شده کاملا از محیط واقعی جدا هستند و هیچ ارتباطی با محیط واقعی ندارند، شما در یک نرم‌افزار شبیه‌ساز پرواز می‌توانید یک هواپیما را بصورت کاملا سر و ته پرواز بدهید اما آیا واقعا در محیط واقعی هم می‌توانید اینکار را انجام دهید؟ شما در نرم افزارهایی مثل Packet Tracer یا NetSim روترها و سویچ های شبکه را شبیه‌سازی می‌کنید و بر روی آنها

khalilifar.ir

دستورات خود را وارد می‌کنید و در یک محیط شبیه‌سازی شده تست ارتباطی هم می گیرید، اما آیا می‌توانید درجه گرمایی که CPU روترها و سویچ های شما بعد از هر پیکربندی دارند را احساس کنید؟ اینگونه محیط‌ها بیشتر مصارف آموزشی و آشنایی با شکل کلی کار با محیط واقعی را دارند، هر چند واقعی هم طراحی شوند باز هم محیط شبیه‌سازی شده هستند و بروز مشکل در چنین محیط‌هایی هیچ تاثیری در محیط واقعی نخواهد داشت. بسیاری از دستورات و کارهایی که در محیط‌های شبیه‌سازی شده انجام می‌دهید ممکن است در محیط واقعی به درستی کار نکند.

زمانیکه صحبت از Emulator یا مقلد می‌شود در خصوص یک سیستم کاملا مشابه با آنچه در محیط واقعی است صحبت می‌کنیم. این سیستم تقلیدکننده دقیقا همان قوانینی را دارد که در سیستم واقعی وجود دارد و می‌توان از آن Copy و Paste سیستم اصلی یاد کرد. نرم‌افزارهای Emulator حتی سورس کد مشابه و در اصطلاح Clone سیستم اصلی هستند، شما هر کاری که در محیط واقعی با نرم‌افزار می‌توانید انجام بدهید در محیط Emulator هم می‌توانید عینا انجام دهید. حتی ورودی و خروجی نرم‌افزار و روش پردازش آن نیز کاملا شبیه به محیط اصلی نرم‌افزار است، تنها تفاوت در محیط کاری است که بر روی آن نرم‌افزار اجرا می‌شود که طبیعتا محیطی به غیر از محیط اصلی اجرای نرم افزار اصلی خواهد بود. قوانین سیستم اصلی و سیستم emulator کاملا شبیه به هم هستند و غیرقابل تغییر هستند. برای مثال شما زمانیکه می‌خواهید نرم افزاری برای سیستم عامل اندروید بنویسید می‌توانید از Emulator ای به نام BlueStacks استفاده کنید. BlueStacks یک نرم‌افزار شبیه‌ساز نیست زیرا عینا برای شما سیستم عامل اندروید را روی ویندوز نمایش می‌دهد و تمامی قوانین اندروید را بر روی آن قرار می‌دهد. شما نرم افزارهای اندرویدی خود را برای تست می‌توانید در این محیط نصب کنید و اجرا کنید Emulator های بازی هم به همین شکل عمل می‌کنند، ممکن است شما بازی‌های PlayStation یا Xbox را بتوانید بصورت کامل در سیستم شخصی خودتان با استفاده از یک Emulator اجرا کنید. در واقع در اینجا شما یک کپی اصلی از بازی اصلی گرفته‌اید و این Emulator یا مقلد است که امکان اجرای آن بر روی سیستم اصلی را می دهد. حتی Bug ها و مشکلات و Crash هایی که ممکن است در سیستم اصلی به وجود بیاید عینا در سیستم Emulate شدن نیز به وجود می‌آید زیرا سورس یا هسته نرم‌افزار به هیچ عنوان تغییر نکرده است.

بصورت کلی زمانیکه صحبت از Emulator می شود در خصوص یک کپی از سیستم اصلی صحبت می‌شود که عین دستگاه و سیستم واقعی کار می‌کند. اما زمانیکه صحبت از Simulator یا شبیه‌ساز می‌شود در واقع در خصوص یک سیستم مدل‌سازی یا Modeling از نرم‌افزار یا سیستم اصلی صحبت می‌کنیم. توجه کنید که همیشه قرار نیست یک سیستم Simulator چیزی شبیه به یک سیستم Emulator شود. اگر بخواهیم مثال شبکه‌ای بزنیم که شما بیشتر برایتان این موضوع جا بیافتند می‌توانیم بگوییم نرم‌افزار مثل Packet Tracer یک نرم‌افزار شبیه ساز یا Simulator است اما نرم افزاری مثل GNS3 یک نرم‌افزار Emulator است زیرا در GNS3 شما سیستم عامل اصلی

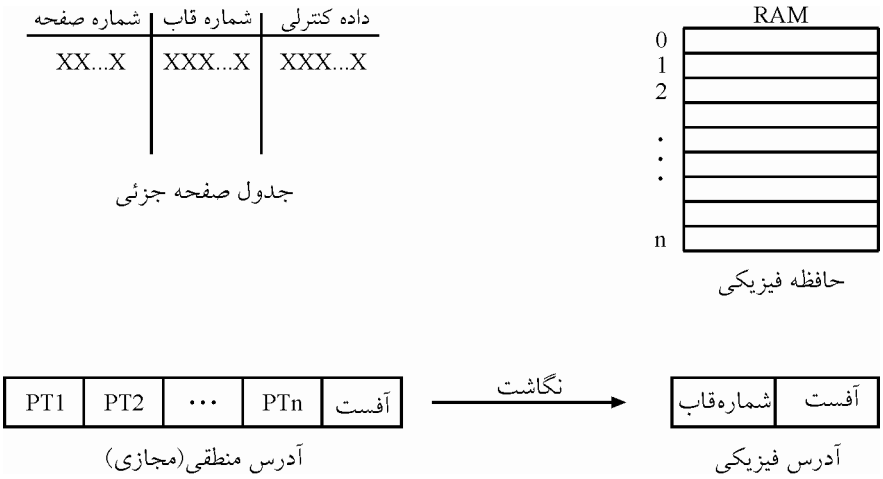
khalilifar.ir

روترها و سویچ ها را بر روی نرم‌افزار Emulator نصب می‌کنید و آن را اجرا می کنید. از نظر سرعت، سرعت Emulator ها به مراتب کندتر از Simulator ها است.

۲- گزینه (۱) صحیح است.

در اینجا برای جدول صفحه جزئی محدودیتی به اندازه یک قاب (صفحه) داریم.

بنابراین اندازه جدول صفحه جزئی برابر اندازه قاب (صفحه) می‌باشد. بنابراین برای محاسبه تعداد سطرهای جدول صفحه جزئی، کافی است، اندازه قاب که برابر اندازه جدول صفحه جزئی است بر اندازه عرض جدول صفحه جزئی تقسیم گردد. به شکل زیر توجه کنید:



توجه: عرض جدول صفحه همواره برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب است، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد.

بنابراین داریم:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

$2B$ = عرض جدول صفحه جزئی

توجه: مطابق فرض سؤال، هر مدخل جدول صفحه (عرض جدول صفحه جزئی) 2 بایت در نظر گرفته شده است.

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^2 \times 2^{10}B}{2^1 B} = 2^{11} = 2048$$

$2^{32}B$ =اندازه فرآیند (فضای آدرس مجازی)

$4KB = 4096B = 2^2 \times 2^{10}B = 2^{12}B$ =اندازه صفحه = اندازه قاب

khalilifar.ir

$$\text{f : تعداد صفحات فرآیند} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه}} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{r : تعداد سطرهای جدول صفحه جزئی} = 2048 = 2^{11}$$

توجه: دقت کنید که اندازه فرآیند(فضای آدرس مجازی) مطابق اطلاعات صورت سوال برابر $2^{32}\,B$ است.

توجه: در اینجا برای جدول صفحه جزئی محدودیتی به اندازه یک قاب (صفحه) داریم. اما چرا این محدودیت وجود دارد؟ زیرا کل جدول صفحه در حالت تک سطحی داخل یک قاب جا نمی‌شود در واقع اندازه جدول صفحه تک سطحی از اندازه یک قاب بیشتر است. بنابراین مطابق آنچه گفتیم رابطه‌ی زیر برقرار است:

$$\text{اندازه فرآیند (فضای آدرس مجازی)} = 2^{32}\,B$$

$$\text{اندازه قاب} = \text{اندازه صفحه} = 4KB = 4096B = 2^2 \times 2^{10}\,B = 2^{12}\,B$$

$$\text{تعداد صفحات فرآیند} = \text{تعداد درایه‌های جدول صفحه تک سطحی} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه}} = \frac{2^{32}}{2^{12}} = 2^{20}$$

توجه: عرض جدول صفحه همواره برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب است، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد.

بنابراین داریم:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

$$\text{عرض جدول صفحه جزئی} = 2B$$

اندازه جدول صفحه تک سطحی و بدون استفاده از جداول صفحه چندسطحی

$$\text{اندازه جدول صفحه تک سطحی} = \text{تعداد درایه‌های جدول صفحه تک سطحی} \times \text{عرض جدول صفحه} = 2^{20} \times 2B = 2^{21}\,B$$

اندازه قاب <اندازه جدول صفحه تک سطحی

$$2^{21}\,B > 2^{12}\,B$$

بنابراین واضح است که اندازه جدول صفحه تک سطحی از اندازه قاب **بیشتر و بزرگتر** است، پس با **محدودیت قاب** مواجه هستیم و باید به سمت طراحی جداول صفحه چندسطحی حرکت کنیم.

توجه: اغلب سیستم‌های کامپیوتری، فضای آدرس منطقی(مجازی) بزرگی را پشتیبانی می‌کنند، مانند کامپیوترهایی که آدرس‌های 32 یا 64 بیتی را پشتیبانی می‌کنند، در چنین محیط‌هایی جدول صفحه معمولی بسیار بزرگ خواهد شد. برای مثال یک سیستم را با 32 بیت فضای آدرس منطقی(مجازی) در نظر بگیرید. اگر اندازه صفحه در این سیستم برابر با 4KB (2^{12}) باشد، در اینصورت جدول صفحه شامل بیش از یک میلیون درایه خواهد بود. $\left(2^{32} / 2^{12} = 2^{20}\right)$. فرض کنید

khalilifar.ir

هر درایه جدول صفحه معمولی، شامل 4 بایت باشد، در اینصورت هر فرآینـد بـه (4×2^{20}) 4MB فضای آدرس فیزیکی فقط برای نگهداری جدول صفحه معمولی نیاز دارد. همچنین مشخص است که با توجه به محدودیت اندازه قاب (4KB) نمی‌تـوان ایـن جـدول صـفحه معمـولی را بصـورت پیوسته درون یک قاب جا داد. 4MB اندازه جدول صفحه معمولی در فضای 4KB مربوط بـه یـک قاب جا نمی‌شود. یک راه حل تقسیم جدول صفحه معمـولی بـه جـداول صـفحه جزئـی و ایجـاد جدول صـفحه چندسـطحی اسـت، در ایـن حالـت جـدول صـفحه معمـولی نیـز هماننـد فرآینـد صفحه‌بندی می‌شود.

حال اطلاعات کافی برای محاسبه تعداد بیت PT1 یعنی اندیس به جدول صفحه اول و PT2 یعنـی اندیس به جدول صفحه دوم را در اختیار داریم:

روش تجزیه

تعداد صفحات فرآیند باید در اندازه (2^{11})r تجزیه گردد.

$$\text{تعداد صفحات فرآیند} = 2^{20} = 2^{9\nearrow PT1} \times 2^{11\nearrow PT2}$$

روش لگاریتم

$$\text{تعداد سطوح جدول چند سطحی} = d = \left\lceil \log_r^f \right\rceil = \left\lceil \log_{2^{11}}^{2^{20}} \right\rceil = 2$$

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{2^{20}}{2^{11}} = 2^9$$

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{r} = \frac{2^9}{2^{11}} < 1$$

توجه: سطح اول، یک جدول به حساب می‌آید، که 2^9 سطر دارد.

توجه: تعداد تقسیم متوالی برابر 2 است، بنابراین تعداد سطوح جدول صفحه چند سطحی برابـر 2 است.

$$\text{تعداد بیت آفست} = \log_2^{\text{اندازه صفحه}} = \log_2^{2^{12}} = 12 \text{ بیت}$$

بنابراین شکل آدرس منطقی (مجازی) به صورت زیر خواهد بود.

PT1	PT2	آفست
9 بیت	11 بیت	12 بیت

32 بیت

khalilifar.ir

توجه: در قابی که جدول صفحه سطح اول در آن قرار دارد، به دلیل مصرف نشدن کل 2^{11} سطر موجود از فضای حداکثری ممکن، قدری **تکه تکه شدن داخلی** داریم. زیرا از کل 2^{11} سطر موجود از فضای حداکثری ممکن داخل جدول صفحه، فقط و فقط 2^{9} سطر در جدول صفحه سطح اول مورد استفاده قرار می‌گیرد. یعنی مقدار 3KB فضا در قابی که جدول صفحه سطح اول در آن قرار دارد به هدر رفته است که همان تکه تکه شدن داخلی در انتهای قابی است که جدول صفحه سطح اول در آن قرار دارد، به صورت زیر:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

عرض جدول صفحه جزئی = 2B

اندازه صفحه = اندازه قاب = $4KB = 4096B = 2^2 \times 2^{10}B = 2^{12}B$

$$\left[\left(2^{11} \times 2B\right) - \left(2^{9} \times 2B\right)\right] = \left[2^{12} - 2^{10}\right] = \left[4KB - 1KB\right] = 3KB$$

توجه: در قاب‌هایی که جداول صفحه سطح دوم در آن قرار دارند به دلیل مصرف شدن کل 2^{11} سطر موجود از فضای حداکثری ممکن، **تکه تکه شدن داخلی** نداریم. زیرا از کل 2^{11} سطر موجود از فضای حداکثری ممکن داخل جدول صفحه، 2^{11} سطر در جداول صفحه سطح دوم مورد استفاده قرار گرفته است.

مطابق آنچه گفتیم دو دسترسی به جداول صفحه جزئی سطح اول و دوم برای ترجمه آدرس و یک دسترسی به داده اصلی (مقصد) لازم است، که مجموع آن شامل سه دسترسی به حافظه می‌گردد. بنابراین نیاز به سه دسترسی به حافظه است.

یک بار دیگر صورت سوال را به دقت بررسی کنید، سیستمی با ترجمه آدرس دو- سطحی و اندازه هر صفحه 4 کیلوبابت در نظر بگیرید. اگر اندازه هر مدخل جدول صفحه برابر 2 بایت (شامل اطلاعات ترجمه و دیگر اطلاعات کنترلی لازم) باشد، چه تعداد فضای بیتی به ترتیب (از راست به چپ) برای جا به جایی (Offset)، اندیس به جدول صفحه اول و اندیس به جدول صفحه دوم برای آدرس مجازی (Virtual Address) 32- بیتی لازم است؟

۱) 10, 10, 12 ۲) 9, 11, 12 ۳) 12, 10, 10 ۴) 12, 11, 9

با توجه به شرایط صورت سوال و مفروضات مساله گزینه اول نمی‌تواند «**طراحی بهینه**» باشد. اما اگر مصلحت و صلاح دانشجویانی که یکسال مطالعه کرده‌اند را در نظر بگیریم و اندیس به جدول صفحه اول و اندیس به جدول صفحه دوم مطرح شده در صورت سوال را به ترتیب PT2 و PT1 در نظر بگیریم، آنگاه گزینه دوم **می‌تواند** پاسخ منطقی برای سوال باشد، در غیر اینصورت و درحالت سختگیرانه این سوال در حالت «**طراحی بهینه**» پاسخ درستی ندارد و باید حذف می‌شد، که نشد. سازمان سنجش آموزش کشور، در کلید اولیه و نهایی خود، گزینه اول را به عنوان پاسخ اعلام کرده بود. و البته حالت درست این است که جدول صفحه اول مطابق PT1 باشد و جدول

khalilifar.ir

صفحه دوم مطابق PT2 باشد. یعنی مقدار درست PT1 برابر 9 و مقدار درست PT2 برابر 11 باشد که در گزینه دوم این مقدارها به ترتیب برقرار نیست.

توجه مهم: در «**طراحی بهینه و طراحی غیربهینه**» هیچگاه مقدار PT1 نمی‌تواند از مقدار PT2 بزرگتر باشد به عبارت دیگر مقدار PT1 همواره کوچکتر یا مساوی مقدار PT2 است.

سوال: چرا در حالت «**طراحی بهینه**»، گزینه اول نمی‌تواند درست باشد؟ روال زیر را در نظر بگیرید:

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد صفحات فرآیند}}{\text{r}} = \frac{\text{f}}{\text{r}} = \frac{2^{20}}{2^{11}} = 2^9$$

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{\text{r}} = \frac{2^9}{2^{11}} < 1$$

توجه: مطابق فرض سؤال، هر مدخل جدول صفحه (عرض جدول صفحه جزئی) 2 بایت در نظر گرفته شده است.

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^2 \times 2^{10}\text{B}}{2^1\text{B}} = 2^{11} = 2048$$

توجه: در این شرایط و رسیدن به گزینه اول، برای رسیدن به 2^{10}، باید مقدار عرض جدول صفحه در سطح اول و دوم که این عرض فقط و فقط شامل حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب در سطح اول و دوم است را برابر 2^2B در نظر بگیرید که این فرض کاملا نادرست است و به تبع رابطه زیر را بر قرار کنید:

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^2 \times 2^{10}\text{B}}{2^2\text{B}} = 2^{10} = 1024$$

که در حالت «**طراحی بهینه**» امکان پذیر نیست. زیرا نمی‌توان عرض جدول صفحه را بی‌دلیل افزایش داد.

توجه: عرض جدول صفحه همواره برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب است، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد.

بنابراین داریم:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه

2B = عرض جدول صفحه

khalilifar.ir

اما اگر بخواهیم به گزینه اول برسیم، باید حالت «طراحی غیربهینه» را در نظر بگیریم:
یعنی عرض جدول صفحه را همان مقدار **2B** در نظر بگیریم، اما برای رسیدن به تعداد سطرهای جدول صفحه مطابق فرض طراح محترم سوال یعنی برابر مقدار 2^{10} سطر برای جدول صفحه سطح اول و دوم، می‌بایست قدری تکه تکه شدن داخلی در تمام قاب‌های جداول سطح اول و دوم تحمل کنیم که در ساخت سیستم عامل‌های واقعی اینطور نیست و اصلا عمومیت ندارد و طراحی «**غیربهینه**» محسوب می‌شود.

توجه: مطابق فرض سؤال، هر مدخل جدول صفحه (عرض جدول صفحه جزئی) 2 بایت در نظر گرفته شده است، که در حالت «**طراحی بهینه**» تعداد سطرهای جدول صفحه سطح اول به صورت زیر است:

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^2 \times 2^{10}\text{B}}{2^1\text{B}} = 2^{11} = 2048$$

حالت «**طراحی غیربهینه**» به صورت زیر است:

توجه: در قاب‌هایی که جداول صفحه سطح اول و دوم در آن قرار دارد به دلیل مصرف نشدن کل 2^{11} سطر موجود از فضای حداکثری ممکن، قدری **تکه تکه شدن داخلی** داریم. زیرا مطابق فرض سوال و نظر طراح محترم روی درست بودن گزینه اول، از کل 2^{11} سطر موجود از فضای حداکثری ممکن، فقط و فقط 2^{10} سطر در جداول صفحه سطح اول و دوم مورد استفاده قرار می‌گیرد. یعنی مقدار 2KB فضا در تمام قاب‌هایی که جداول صفحه سطح اول و دوم در آن قرار دارد به هدر رفته است که همان تکه تکه شدن داخلی در انتهای قاب‌هایی است که جداول صفحه سطح اول و دوم در آن قرار دارد، به صورت زیر:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

$2\text{B} = \text{عرض جدول صفحه جزئی}$

$\text{اندازه صفحه} = \text{اندازه قاب} = 4\text{KB} = 4096\text{B} = 2^2 \times 2^{10}\text{B} = 2^{12}\text{B}$

$$\left[\left(2^{11} \times 2\text{B}\right) - \left(2^{10} \times 2\text{B}\right)\right] = \left[2^{12} - 2^{11}\right] = \left[4\text{KB} - 2\text{KB}\right] = 2\text{KB} = 2 \times 2^{10}\text{B}$$

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} =$$

$$\rightarrow \frac{\left[\left(2^2 \times 2^{10}\text{B}\right) - \left(2\text{KB}\right)\right]}{2^1\text{B}} = \frac{\left[\left(4\text{KB}\right) - \left(2\text{KB}\right)\right]}{2^1\text{B}} = \frac{\left[2\text{KB}\right]}{2^1\text{B}} = \frac{\left[2 \times 2^{10}\text{B}\right]}{2^1\text{B}} = \frac{\left[2^{11}\text{B}\right]}{2^1\text{B}} = 2^{10}$$

توجه: دقت کنید که در رابطه فوق مقدار 2KB را به عنوان تکه تکه شدن داخلی از کل فضای قاب یعنی 4KB کم کردیم، که در فضای باقی‌مانده یعنی 2KB می‌توان 2^{10} سطر را جا داد.

khalilifar.ir

حال اطلاعات کافی برای محاسبه تعداد بیت PT1 یعنی اندیس به جدول صفحه اول و PT2 یعنی اندیس به جدول صفحه دوم را در اختیار داریم:

روش تجزیه

تعداد صفحات فرآیند باید در اندازه $r(2^{10})$ تجزیه گردد.

$$\text{تعداد صفحات فرآیند} = 2^{20} = 2^{10^{\nearrow PT1}} \times 2^{10^{\nearrow PT2}}$$

روش لگاریتم

$$\text{تعداد سطوح جدول چند سطحی} = d = \left\lceil \log_r^f \right\rceil = \left\lceil l0g_{2^{10}}^{2^{20}} \right\rceil = 2$$

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{2^{20}}{2^{10}} = 2^{10}$$

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{r} = \frac{2^{10}}{2^{10}} = 1$$

توجه: سطح اول، یک جدول به حساب می‌آید، که 2^{10} سطر دارد.

توجه: در قاب‌هایی که جداول صفحه سطح اول و دوم در آن قرار دارد به دلیل مصرف نشدن کل 2^{11} سطر موجود از فضای حداکثری ممکن، قدری **تکه تکه شدن داخلی** داریم. زیرا از کل 2^{11} سطر موجود از فضای حداکثری ممکن داخل جدول صفحه، فقط و فقط 2^{10} سطر در جدول صفحه سطح اول و دوم مورد استفاده قرار می‌گیرد. یعنی مقدار 2KB فضا در قاب‌هایی که جدول صفحه سطح اول و دوم در آن قرار دارد به هدر رفته است که همان تکه تکه شدن داخلی در انتهای قابی است که جداول صفحه سطح اول و دوم در آن قرار دارد، به صورت زیر:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

$2B$ = عرض جدول صفحه جزئی

$$\text{اندازه صفحه = اندازه قاب} = 4KB = 4096B = 2^2 \times 2^{10}B = 2^{12}B$$

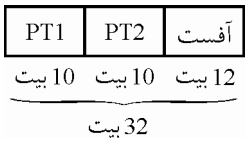
$$\left[\left(2^{11} \times 2B\right) - \left(2^{10} \times 2B\right)\right] = \left[2^{12} - 2^{11}\right] = \left[4KB - 2KB\right] = 2KB = 2 \times 2^{10}B$$

توجه: تعداد تقسیم متوالی برابر 2 است، بنابراین تعداد سطوح جدول صفحه چند سطحی برابر 2 است.

$$\text{تعداد بیت آفست} = \log_2^{\text{اندازه صفحه}} = \log_2^{2^{12}} = 12 \text{ بیت}$$

بنابراین شکل آدرس منطقی (مجازی) به صورت زیر خواهد بود.

khalilifar.ir



۳- گزینه (۱) صحیح است.

الگوریتم LRU (Least Recently Used)

ایده اصلی این الگوریتم این است که اگر صفحه‌ای در چند دستور اخیر مراجعات زیادی داشته است، به احتمال قوی در دستورات بعدی هم ارجاعات زیادی خواهد داشت، همچنین اگر یک صفحه، اخیرا هیچ مراجعه‌ای نداشته، احتمالاً در آینده نزدیک هم ارجاعی نخواهد داشت. در واقع این الگوریتم بیان می‌کند هنگام وقوع خطای نقص صفحه، صفحه‌ای را حذف کنید که طولانی‌ترین زمان عدم استفاده را دارد.

می‌توان گفت LRU تقریبی از الگوریتم بهینه می‌باشد که در آن به جای توجه به آینده، به گذشته توجه می‌شود.

توجه: الگوریتم LRU ناهنجاری بی‌لیدی ندارد، بنابراین با افزایش تعداد قاب، همواره، نقص صفحه کاهش می‌یابد.

اگر 4 قاب در نظر گرفته شود:

نگاه به گذشته ←

رشته مراجعات	1	2	3	4	5	6	2	3	2	1	6	7
قاب 1	1	1	1	1	5	5	5	5	5	1	1	1
قاب 2		2	2	2	2	6	6	6	6	6	6	6
قاب 3			3	3	3	3	2	2	2	2	2	2
قاب 4				4	4	4	4	3	3	3	3	7
نقص صفحه	*	*	*	*	*	*	*	*		*		*

در این حالت 10 نقص صفحه به وقوع پیوست.

الگوریتم FIFO (First In First Out)

این الگوریتم ساده‌ترین الگوریتم از نظر پیاده‌سازی است. در این روش سیستم عامل لیستی از صفحات را به ترتیب ورود به حافظه نگه می‌دارد. وقتی یک خطای نقص صفحه رخ می‌دهد، سیستم عامل قدیمی‌ترین صفحه را برای بیرون رفتن انتخاب می‌کند. ایده این روش این است که قدیمی‌ترین صفحه شانس مورد استفاده قرار گرفتن را به اندازه کافی در اختیار داشته و اکنون باید این شانس به صفحه دیگری داده شود.

نکته: نقص الگوریتم FIFO این است که حتی اگر صفحه‌ای بارها و به طور مکرر استفاده شود،

khalilifar.ir

سرانجام به قدیمی‌ترین صفحه تبدیل و حذف می‌شود، در صورتی که احتمالاً بلافاصله باید دوباره به حافظه آورده شود.

رشته مراجعات	1	2	3	4	5	6	2	3	2	1	6	7
قاب 1	1	1	1	1	2	3	4	5	5	6	6	2
قاب 2		2	2	2	3	4	5	6	6	2	2	3
قاب 3			3	3	4	5	6	2	2	3	3	1
قاب 4				4	5	6	2	3	3	1	1	7
نقص صفحه	*	*	*	*	*	*	*	*		*		*

در این حالت 10 نقص صفحه به وقوع پیوست.

۴- گزینه (۴) صحیح است.

مطابق قانون امدال (amdahl) داریم:

$$T_{improved} = [T_{unaffected}] + \left[T_{affected} \times \frac{1}{improvment\ factor} \right]$$

مثال: برنامه‌ای را در نظر بگیرید که زمان اجرای آن 100 ثانیه است، 80 ثانیه‌ی آن مربوط به عملیات ضرب می‌باشد، اگر سرعت عملیات ضرب 60 درصد تسریع پیدا کند، آنگاه زمان اجرا چقدر خواهد بود؟

توجه: به تفاوت کلمه **سرعت عملیات** و **زمان اجرا** دقت کنید، به طور مثال، سرعت عملیات کلی دو برابر شود یعنی زمان اجرای کلی از 100 ثانیه به 50 ثانیه بهبود یابد.

$$T_{improved} = [0.2 \times 100] + \left[0.8 \times 100 \times \frac{1}{1+0.6} \right] = [20] + [50] = 70$$

بنابراین زمان اجرای کلی 30 ثانیه بهبود یافت.

توجه: دقت کنید که 80 ثانیه‌ی عملیات ضرب، سرعت عملیات آن 60 درصد تسریع پیدا کرد و 80 ثانیه به 50 ثانیه بهبود یافت، برای درک بهتر فرض کنید که 80 ثانیه‌ی عملیات ضرب، سرعت عملیات آن 100 درصد تسریع پیدا کند که 80 ثانیه به 40 ثانیه بهبود می‌یابد. یعنی **سرعت** عملیات ضرب دو برابر شده است که به تبع آن **زمان اجرای** عملیات ضرب نصف می‌شود، مطابق رابطه‌ی زیر:

$$T_{improved} = [0.2 \times 100] + \left[0.8 \times 100 \times \frac{1}{1+1} \right] = [20] + [40] = 60$$

توجه: چنانچه در زمان اجرا تمام بخش‌های فرآیند مورد نظر به طور یکنواخت درگیر باشند، می‌توانیم مساله را به شکل زیر حل کنیم:

توجه: فرض کنیم فرآیندی زمان اجرای آن یک ثانیه طول بکشد.

khalilifar.ir

اگر مطابق گزینه‌ی **اول** عمل نماییم، رابطه‌ی زیر را خواهیم داشت:

$$T_{improved} = [0.8 \times 1] + \left[0.2 \times 1 \times \frac{1}{1+0.6}\right] = [0.8] + [0.125] = 0.925\,s$$

اگر مطابق گزینه‌ی **دوم** عمل نماییم، رابطه‌ی زیر را خواهیم داشت:

$$T_{improved} = [0.7 \times 1] + \left[0.3 \times 1 \times \frac{1}{1+0.5}\right] = [0.7] + [0.2] = 0.9\,s$$

اگر مطابق گزینه‌ی **سوم** عمل نماییم، رابطه‌ی زیر را خواهیم داشت:

$$T_{improved} = [0.5 \times 1] + \left[0.5 \times 1 \times \frac{1}{1+0.3}\right] = [0.5] + [0.384] = 0.884\,s$$

اگر مطابق گزینه‌ی **چهارم** عمل نماییم، رابطه‌ی زیر را خواهیم داشت:

$$T_{improved} = [0.6 \times 1] + \left[0.4 \times 1 \times \frac{1}{4}\right] = [0.6] + [0.1] = 0.7\,s$$

بنابراین واضح است که روش 40 درصد برنامه 4 برابر تسریع یابد یعنی گزینه‌ی چهارم، **بیشترین تسریع را در زمان اجرای یک برنامه ایجاد می‌کند. بنابراین گزینه‌ی چهارم پاسخ سوال است.**

۵- گزینه (۴) صحیح است.

صورت سوال به این شکل است:

مسئله پل را در نظر بگیرید که هر خودرو برای عبور باید از هر دو سر پل به ترتیب عبور کند. اگر تعداد زیادی خودرو در هر دو طرف این پل برای عبور از آن وجود داشته باشد و در هر لحظه امکان عبور فقط یک خودرو باشد. کدام مورد درست‌تر است؟

در هر لحظه امکان عبور فقط یک خودرو یعنی برقراری شرط انحصار متقابل برای عبور از پل، که برقراری شرط انحصار متقابل توسط هم سمافور و هم مانیتور امکان‌پذیر است.

۱) این مسئله فقط با سمافور قابل حل است.

گزینه اول نادرست است، زیرا برقراری شرط انحصار متقابل توسط هم سمافور و هم مانیتور امکان‌پذیر است.

۲) این مسئله فقط با مانیتور قابل حل است.

گزینه دوم نادرست است، زیرا برقراری شرط انحصار متقابل توسط هم سمافور و هم مانیتور امکان‌پذیر است.

۳) این مسئله نه با سمافور و نه با مانیتور قابل حل است.

گزینه سوم نادرست است، زیرا برقراری شرط انحصار متقابل توسط هم سمافور و هم مانیتور امکان‌پذیر است.

khalilifar.ir

۴) این مسئله هم با سمافور و هم با مانیتور قابل حل است.

گزینه چهارم درست است، زیرا برقراری شرط انحصار متقابل توسط هم سمافور و هم مانیتور امکان‌پذیر است.

توجه: سازمان سنجش آموزش کشور، ابتدا در کلید اولیه خود گزینه دوم را به عنوان پاسخ اعلام نمود، اما در کلید نهایی گزینه‌ی چهارم را به عنوان پاسخ نهایی اعلام نمود، که کار درستی بوده است.

۶- گزینه (۴) صحیح است.

زمان‌بندی تمام وقایع متناوب مستقل از نوع زمان‌بندی در یک سیستم بی‌درنگ به شرطی امکان‌پذیر است که رابطه‌ی زیر برقرار باشد:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1$$

C_i : زمان اجرای فرآیند

P_i : دوره تناوب

مطابق فرض سوال، فرآیند P_2 پس از 35 واحد زمانی از لحظه‌ی ایجاد، دوباره تکرار می‌شود، که موعد (Deadline) آن برابر100 است. بنابراین فرآیند P_2 یکبار در لحظه‌ی 15 با موعد 100 واحد زمانی ایجاد می‌شود و یکبار هم در لحظه‌ی 35 با موعد 100 واحد زمانی ایجاد می‌شود که این سناریو به طور متناوب تکرار می‌شود. بنابراین مطابق رابطه‌ی فوق داریم:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} = \left[\frac{C_1}{P_1}\right] + \left[\frac{C_{21}}{P_{21}} + \frac{C_{22}}{P_{22}}\right] + \left[\frac{C_3}{P_3}\right] + \left[\frac{C_4}{P_4}\right] =$$
$$\left[\frac{20}{50}\right] + \left[\frac{25}{100} + \frac{25}{100}\right] + \left[\frac{20}{90}\right] + \left[\frac{x}{100}\right] = 0.4 + 0.5 + 0.22 + \left[\frac{x}{100}\right] \le 1$$

توجه: مطابق فرض سوال، فرآیند P_4 در زمان صفر با دوره تناوب 100 واحد زمانی ایجاد می‌شود.

واضح است که حاصل جمع عبارت $0.4+0.5+0.22$ از مقدار یک بزرگتر است یعنی $0.4+0.5+0.22>1$ بنابراین رابطه‌ی زیر برقرار نیست:

$$0.4 + 0.5 + 0.22 + \left[\frac{x}{100}\right] \le 1$$

بنابراین برای فرآیند P_4 برای آنکه زمان‌بندی سیستم امکان‌پذیر باشد، مقداری برای Burst Time یا زمان اجرای آن وجود ندارد.

توجه: در سیستم‌های بی‌درنگ به دلیل تناوبی بودن اجرای فرآیندها، زمان ورود به سیستم معنا ندارد و در رابطه‌ی فوق لحاظ نمی‌گردد.

---

khalilifar.ir

۷- گزینه (۴) صحیح است.

به طور کلی مدیریت و زمان‌بندی نخ‌ها به سه روش زیر انجام می‌گردد:

۱- روش سطح کاربر یا مدل چند به یک (many to one)

در این روش فقط زمان‌بند پردازنده و زمان‌بند چند نخی در سطح کاربر وجود دارد و زمان‌بند چند نخی در سطح هسته در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل فقط فرآیندها را می‌شناسد و هیچ اطلاعاتی از نخ‌ها ندارد. در واقع اولویت‌بندی نخ‌ها، مدیریت نخ‌ها و زمان‌بند چندنخی در سطح کاربر و توسط یک بسته نرم‌افزاری انجام می‌گردد. بدین معنی که نخ‌ها را برنامه‌نویس مشخص می‌کند و مدیریت آن‌ها را نیز بر عهده می‌گیرد. بنابراین زمان‌بند پردازنده، براساس الگوریتم مشخصی مثلاً نوبت چرخشی پردازنده را در اختیار یکی از فرآیندهای آماده قرار می‌دهد. سپس زمان‌بند چند نخی در سطح کاربر، متناسب با کاربردی که در آن فرآیند به کار گرفته می‌شود، الگوریتم زمان‌بند را انتخاب کرده و تصمیم می‌گیرد که پردازنده در اختیار کدام‌یک از نخ‌های آماده در فرآیند موردنظر قرار گیرد و تا زمانی که پردازنده در تملک فرآیند باشد و یا تا قبل از پایان برش زمانی مربوط به فرآیند، نخ‌های یک فرآیند از پردازنده بهره‌مند می‌شوند و به محض مسدود شدن یک نخ، و یا پایان برش زمانی یک فرآیند، یا اتمام فرآیند، پردازنده به فرآیند بعدی تعلق می‌گیرد.

توجه: در این روش نخ ماهیت منطقی دارد و از دید کاربر فقط وجود دارد، در واقع از نظر سیستم عامل ماهیت فیزیکی ندارد، بنابراین نخ کاربر در این روش همانند یک تابع در فرآیند می‌باشد که از رجیستر و پشته مختص به خود نیز بهره‌مند نمی‌باشد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، بین پدران خانواده‌ها تقسیم گردد و این پدران خانواده‌ها باشند که تصمیم بگیرند به هر عضو خانواده چه مقدار نقدینگی تعلق بگیرد. در این روش فقط پدران شماره حساب مختص به خود را دارند. اما اگر یکی از اعضای خانواده خطایی انجام دهد و محکوم گردد، آنگاه تمام اعضای خانواده برای مدتی از خدمات دولت محکوم می‌گردند، زیرا در این مدل، دولت فقط پدران خانواده را می‌شناسد و از اعضای خانواده اطلاعی ندارد. بنابراین حساب پدر خانواده برای مدتی مسدود می‌گردد.

توجه: عمل تعویض متن مابین نخ‌های یک فرآیند کاربر، کاملاً در سطح کاربر و با سربار بسیار ناچیز (در حد فراخوانی نخ بعدی) و بدون تغییر حالت پردازنده به مد هسته پردازنده، توسط زمان‌بند چندنخی در سطح کاربر (برنامه‌های کاربر) انجام می‌گردد. اما عمل تعویض متن مابین فرآیندها (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل و در مد هسته پردازنده، توسط زمان‌بند پردازنده برای انتخاب یک فرآیند جدید بر اساس یک الگوریتم خاص انجام می‌گردد.

khalilifar.ir

توجه: به دلیل آنکه نخ‌های یک فرآیند کاربر، تماماً در فضای کاربر مدیریت می‌شوند، اگر نخ موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل نه تنها آن نخ، بلکه کل فرآیند کاربر را که شامل تمام نخ‌های دیگر می‌باشد، مسدود می‌کند، زیرا هسته سیستم عامل خبری از نخ‌های داخل فرآیند کاربر ندارد. در واقع در این روش هسته سیستم عامل نخ‌ها را همانند توابع داخل یک فرآیند می‌بیند، یعنی هسته سیستم عامل، یک فرآیند چند نخی سطح کاربر را، مانند یک فرآیند تک نخی اما دارای چند تابع مختلف می‌بیند!

توجه: این راه‌کار، منجر به عدم امکان هم‌روندی (در سیستم‌های تک‌پردازنده‌ای) نخ‌های داخل یک فرآیند کاربر در حالت انسداد یک نخ داخل یک فرآیند کاربر می‌گردد.

البته اگر نخی داخل یک فرآیند کاربر مسدود نگردد، امکان هم‌روندی میان نخ‌های داخل فرآیند کاربر برقرار است. مانند یک تیم فوتبال 11 نفره که اگر بازیکنی مرتکب خطا گردد، از آن‌جا که داور فقط نام تیم را می‌شناسد و نه تک‌تک بازیکنان تیم را، آن‌گاه کل تیم را جریمه، اخراج و مسدود می‌کند. اما اگر هیچ‌یک از بازیکنان تیم مرتکب خطایی نگردد، واضح است که هم‌روندی برقرار است.

توجه: فرض کنید یک کیک داریم که آن را به چهار قسمت مساوی تقسیم کرده‌ایم، همچنین فرض کنید خوردن هر بخش کیک یک ساعت زمان بخواهد، اگر یک نفر بخواهد تمام این کیک را بخورد، پس 4 ساعت طول می‌کشد اگر 4 نفر بخواهند تمام این کیک را بخورند و به هر نفر یک بخش کیک داده شود، آنگاه خوردن تمام کیک به‌طور موازی ۱ ساعت طول خواهد کشید. در روش سطح کاربر، یک فرآیند چند نخی کاربر نمی‌تواند از امتیازات چند پردازنده‌ای بهره ببرد، زیرا در روش سطح کاربر، هسته سیستم عامل در هر لحظه فقط یک پردازنده را در اختیار نخ‌های یک فرآیند کاربر قرار می‌دهد. حتی اگر چند پردازنده موجود باشد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر نیست. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند کاربر وجود ندارد.

توجه: نرم‌افزارهای POSIX P-threads و Mach C-thread به عنوان یک بسته نرم‌افزاری، می‌توانند جهت مدیریت نخ‌ها و زمان‌بند چند نخی در سطح کاربر مورد استفاده قرار گیرند.

توجه: در این روش تخصیص منابع و زمان‌بندی پردازنده، بر روی فرآیندها انجام می‌شود. همچنین زمان‌بندی نخ‌های سطح کاربر، بر عهده زمان‌بند چندنخی سطح کاربر خواهد بود.

توجه: سیستم عامل سولاریس، مدل چند به یک را پیاده سازی می‌کند.

۲- روش سطح هسته یا مدل یک به یک (one to one)

در این روش فقط زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته وجود دارد و زمان‌بند چند نخی در سطح کاربر در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل نه تنها فرآیندها را می‌شناسد بلکه از وجود نخ‌های داخل یک فرآیند نیز در صورت وجود نخ آگاه است.

khalilifar.ir

در واقع مدیریت نخ‌ها و زمان‌بند چندنخی (نخ‌های فرآیندهای کاربر و نخ‌های فرآیندهای سیستم عامل) در سطح هسته، توسط هسته سیستم عامل انجام می‌گردد و کاربر هیچ دیدی از این کار ندارد. انگار که اجتماع تمام نخ‌های فرآیندهای سیستم عامل و کاربر را در نظر بگیرید، حال بر روی تک نخ‌ها بر اساس یک الگوریتم خاص حرکت کنید. بنابراین زمان‌بند چند نخی در سطح هسته، براساس الگوریتم مشخصی مثلاً نوبت چرخشی پردازنده را در اختیار یکی از نخ‌های آماده قرار می‌دهد. توجه کنید در این روش پردازنده دیگر در تملک فرآیند نیست بلکه در تملک نخ‌ها است. در واقع تا زمانی که پردازنده در تملک یک نخ باشد و تا قبل از مسدود شدن نخ، و یا تمام شدن نخ و یا پایان برش زمانی مربوط به نخ، نخ می‌تواند از پردازنده بهره ببرد و به محض مسدود شدن یک نخ، یا پایان برشی زمان مربوط به نخ یا اتمام نخ، پردازنده می‌تواند به نخ بعدی که ممکن است، نخ بعدی، نخی هم‌خانواده با نخ قبلی در یک فرآیند باشد، یا نخی در یک فرآیند دیگر باشد، تعلق بگیرد. در واقع زمان‌بند چند نخی در سطح هسته سیستم عامل تعیین می‌کند که نخ بعدی که باید شروع به کار کند متعلق به همان فرآیند باشد و یا از یک فرآیند دیگر انتخاب شود.

توجه: برای انجام زمان‌بندی چند نخی، هسته سیستم عامل باید علاوه بر جدول فرآیندها، یک جدول نخ (شبیه جدول فرآیند) داشته باشد که اطلاعات تمامی نخ‌های موجود در سیستم را نگهداری کند. مجدداً تأکید می‌کنیم که در این حالت هر نخ TCB خاص خود را دارد. به عبارت دیگر هر نخ رجیستر و پشته مختص به خود را دارد.

توجه: در این روش نخ ماهیت فیزیکی دارد، و از دید کاربر و سیستم عامل وجود دارد، بنابراین در این روش، هر نخ، رجیستر و پشته مختص به خود را دارد، به عبارت دیگر هر نخ TCB مختص به خود را دارد.

توجه: عملیات مرتبط با نخ‌های سطح کاربر، مانند ایجاد (بارگذاری TCB مختص به نخ) و پایان دادن (ذخیره‌سازی TCB مختص به نخ) بر عهده هسته سیستم عامل است و توسط زمان‌بند چند نخی در سطح هسته سیستم عامل انجام می‌گردد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، بدون اعمال هیچ‌گونه اولویت‌بندی بین تک تک اعضاء یک کشور تقسیم گردد (حال این اعضاء شهروند عام باشد یا خاص) در این روش هر یک از اعضای کشور شماره حساب مختص به خود را دارند. اما اگر یکی از افراد کشور خطایی انجام دهد و محکوم گردد، آنگاه فقط همان فرد برای مدتی از خدمات دولت محروم می‌گردد و دولت به بقیه اعضاء خانواده آن فرد همچنان خدمات ارائه می‌دهد. زیرا دولت از مشخصات تک تک اعضاء جامعه آگاه است.

توجه: عمل تعویض متن، مابین نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل، در مد هسته پردازنده توسط زمان‌بند چندنخی در سطح هسته سیستم

khalilifar.ir

عامل انجام می‌گردد و عمل تعویض متن مابین فرآیندها (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل و در مد هسته پردازنده توسط زمان‌بند پردازنده انجام می‌گردد.
توجه: به دلیل آنکه نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل)، تماماً در فضای هسته سیستم عامل مدیریت می‌شوند و هسته سیستم عامل از وجود نخ‌های یک فرآیند آگاه است، اگر نخ موجود در یک فرآیند، یک فراخوان سیستمی مسدود کننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم‌خانواده با آن نخ مسدود شده، همچنان می‌توانند از پردازنده بهره ببرند.
توجه: این راه کار، منجر به امکان هم‌روندی (در سیستم‌های تک پردازنده‌ای) و امکان توازی (در سیستم‌های چندپردازنده‌ای) میان فرآیندهای مختلف و یا میان نخ‌های داخل یک فرآیند می‌شود. البته اگر نخی مسدود نگردد، درجه همروندی و توازی بالاتر هم خواهد رفت، زیرا در اینصورت همه نخ‌ها به طور هم‌روند یا موازی در حال حرکت هستند. مانند یک تیم فوتبال 11 نفره که اگر بازیکنی مرتکب خطا گردد، همروندی یا توازی همچنان برقرار است، چون داور تک‌تک بازیکنان را می‌شناسد و فقط بازیکن خاطی را مسدود، جریمه و اخراج می‌کند و بقیه بازیکنان تیم به بازی خود ادامه می‌دهند، اما اگر هیچ یک از بازیکنان تیم مرتکب خطا نگردند، واضح است که همروندی و توازی بالاتر هم خواهد بود.
توجه: مثال کیک مطرح شده را مجدداً به یاد آورید، در روش سطح هسته، یک فرآیند چندنخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چندپردازنده‌ای بهره ببرد. در روش سطح هسته، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر است. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد.
توجه: در این روش، منابع به فرآیندها اختصاص می‌یابد ولی زمان‌بندی پردازنده، بر روی نخ‌ها انجام می‌گیرد و زمان‌بندی نخ‌های سطح کاربر و نخ‌های سطح هسته، برعهده زمان‌بند چندنخی سطح هسته می‌باشد.
توجه: سیستم عامل لینوکس، خانواده سیستم عامل ویندوز و سولاریس 9 مدل یک به یک را پیاده‌سازی می‌کنند.

۳- روش ترکیبی (سطح کاربر و هسته) یا مدل چند به چند (many to many)

این روش از اجتماع دو روش سطح کاربر و هسته ابداع گردیده است. در این روش علاوه بر زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته سیستم عامل، زمان‌بند چند نخی نیز در سطح کاربر برای زمان‌بندی و اولویت‌دهی نخ‌های فرآیند کاربر وجود دارد. در واقع زمان‌بندی نخ‌های فرآیندهای سیستم عامل، توسط هسته سیستم عامل و زمان‌بندی نخ‌های فرآیندهای کاربر، توسط برنامه کاربر انجام می‌گردد، که این امر منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد. در

khalilifar.ir

این روش، زمان‌بند چند نخی در سطح کاربر، نخ‌های کاندید خود را از میان نخ‌های متعدد در فرآیندهای مختلف کاربر براساس یک الگوریتم خاص انتخاب و تحویل یک زمان‌بند چند نخی در سطح هسته می‌دهد. در واقع انتخاب نخ‌های کاندید (زمان‌بندی) در فرآیندهای کاربر به خود کاربر واگذار شده است که همانطور که گفتیم منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد. حال اجتماع حاصل از نخ‌های کاندید فرآیندهای کاربر و نخ‌های فرآیندهای سیستم عامل توسط زمان‌بند چندنخی هسته سیستم عامل براساس یک الگوریتم خاص زمانبندی می‌شود.

از توضیحات فوق این مفهوم برداشت می‌شود که هسته سیستم عامل باید این قابلیت را داشته باشد که کاربر بتواند نخ‌های فرآیندهای سطح خود را به هسته سیستم عامل معرفی کند. بنابراین در این روش هسته سیستم عامل نه تنها فرآیندهای کاربر را می‌شناسد، بلکه از وجود نخ‌های کاربر داخل فرآیندهای کاربر نیز در صورت وجود نخ آگاه است.

توجه: برای معرفی نخ‌های کاندید فرآیندهای سطح کاربر به زمانبند چندنخی در سطح هسته سیستم عامل، علاوه بر جایگاه‌های مخصوص نخ‌های فرآیندهای سیستم عامل در زمان‌بند چندنخی در سطح هسته سیستم عامل، تعدادی جایگاه، ویژه نخ‌های سطح کاربر نیز، در زمانبند چندنخی در سطح هسته سیستم عامل در نظر گرفته شده است، به این جایگاه **ویژه که محیط اجرای نخ** نیز نامیده می‌شود، **LWP** گفته می‌شود. در واقع هر نخ کاندید انتخاب شده توسط زمانبند چندنخی در سطح کاربر، پس از معرفی به زمانبند چندنخی در سطح هسته، در یکی از جایگاه‌های ویژه که همان LWP است، جهت زمان‌بندی توسط زمان‌بند چندنخی در سطح هسته، قرار می‌گیرد.

توجه: LWP سرواژه‌ی عبارت Light Weight Process و به معنی فرآیند سبک وزن است.

توجه: هنگامی که یک نخ به زمان‌بند چندنخی در سطح هسته معرفی می‌گردد و در ادامه در یک LWP جهت زمان‌بندی توسط زمان‌بند چندنخی در سطح هسته، قرار می‌گیرد، در طول حیات خود ممکن است، در LWP های متفاوتی بخش‌هایی از اجرای خود را طی کند، مثلاً یک نخ در صورت رسیدن به عملیات ورودی و خروجی، جایگاه خود یعنی LWP را واگذار می‌کند و در اجرای بعدی پس از پایان عملیات ورودی و خروجی ممکن است به یک LWP دیگر منتسب شود. توجه کنید که **LWP محیط اجرای نخ** می‌باشد.

توجه: برای انجام زمان‌بندی چندنخی، هسته سیستم عامل باید علاوه بر جدول فرآیندها، یک جدول نخ (شبیه جدول فرآیند) داشته باشد که اطلاعات تمامی نخ‌های موجود در سیستم را نگهداری کند.

توجه: در این روش نخ ماهیت فیزیکی دارد و از دید کاربر و سیستم عامل وجود دارد، بنابراین در این روش، هر نخ، رجیستر و پشته مختص به خود را دارد، به عبارت دیگر هر نخ TCB مختص به خود را دارد.

khalilifar.ir

توجه: عملیات مرتبط با نخ‌های سطح کاربر، مانند ایجاد (بارگذاری TCB مختص به نخ) و پایان دادن (ذخیره‌سازی TCB مختص به نخ) بر عهده هسته سیستم عامل است و توسط زمان‌بند چندنخی سطح هسته انجام می‌گردد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، با اعمال نوعی اولویت‌بندی بین برخی از اعضاء یک کشور تقسیم گردد (حال این اعضاء شهروند عام باشند یا خاص) در این روش هریک از اعضاء اولویت‌دار و منتخب کشور شماره حساب مختص به خود را دارند. اما اگر یکی از اعضاء خانواده خطایی انجام دهد و محکوم گردد، آنگاه فقط همان فرد برای مدتی از خدمات دولت محروم می‌گردد و دولت به بقیه اعضاء خانواده آن فرد، همچنان خدمات ارائه می‌دهد، زیرا دولت از مشخصات تک تک اعضاء آن خانواده آگاه است.

توجه: عمل تعویض متن مابین نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل، در مد هسته پردازنده توسط زمانبند چندنخی در سطح هسته سیستم عامل انجام می‌گردد و عمل تعویض متن مابین فرآیندها (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل و در مد هسته پردازنده توسط زمانبند پردازنده انجام می‌گردد.

توجه: به دلیل آنکه نخ‌های کاندید فرآیندهای سطح کاربر، به هسته سیستم عامل معرفی می‌گردند و هسته سیستم عامل از وجود نخ‌های کاندید فرآیندهای سطح کاربر آگاه است، اگر یک نخ کاندید موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم‌خانواده با آن نخ مسدود شده، همچنان می‌تواند از پردازنده بهره ببرند. این راه‌کار، منجر به امکان هم‌روندی (در سیستم‌های تک‌پردازنده‌ای) و امکان توازی (در سیستم‌های چندپردازنده‌ای) میان فرآیندهای مختلف کاربر و یا میان نخ‌های داخل یک فرآیندکاربر می‌شود. البته اگر نخی مسدود نگردد، درجه هم‌روندی و توازی بالاتر هم خواهد رفت، زیرا در اینصورت همه نخ‌ها به طور هم‌روند یا موازی در حال حرکت هستند.

مانند یک تیم فوتبال 11 نفره که اگر بازیکنی مرتکب خطا گردد، هم‌روندی یا توازن همچنان برقرار است، چون داور تک تک بازیکنان را می‌شناسد و فقط بازیکن خاطی را مسدود، جریمه و اخراج می‌کند و بقیه بازیکنان تیم به بازی خود ادامه می‌دهند، اما اگر هیچ‌یک از بازیکنان تیم مرتکب خطا نگردد، واضح است هم‌روندی و توازی بالاتر هم خواهد رفت.

توجه: همانند نخ‌های سطح کاربر، همروندی و توازی برای نخ‌های سطح هسته سیستم عامل نیز، در حالت انسداد یک نخ یا عدم انسداد یک نخ، برقرار است.

توجه: مثال کیک مطرح شده را مجدداً به یاد آورید، در روش ترکیبی، یک فرآیند چندنخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چندپردازنده‌ای بهره ببرد. در

khalilifar.ir

روش ترکیبی، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر است. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد.
توجه: در این روش نیز، منابع به فرآیندها اختصاص می‌یابد ولی زمان‌بندی پردازنده، بر روی نخ‌ها، انجام می‌گیرد. اما زمان‌بندی نخ‌های سطح کاربر، بر عهده زمان‌بند چندنخی سطح کاربر و زمان‌بندی نخ‌های سطح هسته، بر عهده زمان‌بند چندنخی سطح هسته خواهد بود.
در نرم‌افزارهای بی‌درنگ باید خروجی و پاسخ نهایی در یک زمان مشخص و از پیش تعیین شده حاصل شود. در این نرم‌افزارها، زمان نقشی کلیدی ایفا می‌کند و زمان پاسخ باید به موقع و تضمین شده باشد. نرم‌افزارهای بی‌درنگ معمولاً به عنوان یک دستگاه کنترلی در یک کاربرد خاص (مثلاً صنعتی) به کار گرفته می‌شوند. در این نرم‌افزارها دیر پاسخ دادن به همان بدی پاسخ ندادن است. در این نوع نرم‌افزارها هدف اصلی طراحان، پاسخگویی سریع (در مهلت تعیین شده) به رویدادها و درخواست‌ها می‌باشد و راحتی کاربران و بهره‌وری منابع در درجه‌های بعدی اهمیت، قرار دارند. نتیجه اینکه زمان پاسخ در سیستم‌های بی‌درنگ الزاما باید به موقع و تضمین شده باشد. در طرف مقابل، در سیستم‌های اشتراک زمانی و عمومی، داشتن زمان پاسخ کوتاه مطلوب است ولی الزامی نیست.

به طور کلی سیستم‌های بی‌درنگ به دو نوع زیر طبقه‌بندی می‌شوند:

۱- سیستم بی‌درنگ سخت (Hard Real-Time)

۲- سیستم بی‌درنگ نرم (Soft Real-Time)

در سیستم‌های بی‌درنگ سخت، ضرب‌العجل‌ها یا مهلت زمانی(deadline) باید تحت هر شرایطی رعایت شود، مانند ترمز اتومبیل، باید گرفته شود و زمان بسیار نزدیک است این بی‌درنگ سخت است، دیر پاسخ دادن به همان بدی پاسخ ندادن است، نباید دیر پاسخ دهد یا دستگاه کنترل ضربان قلب انسان، اگر ضربان نبض نبود همه رو باید سریع بیدار کند و نباید دیر پاسخ دهد. در سیستم‌های بی‌درنگ سخت، معمولاً وسایل ذخیره‌سازی ثانویه همچون دیسک به دلیل کندی آن وجود ندارد و به جای آن از حافظه‌های ROM استفاده می‌شود. سیستم عامل‌های **پیشرفته** نیز در این سیستم‌ها وجود ندارد چرا که سیستم عامل کاربر را از سخت‌افزار جدا می‌کند و این جداسازی باعث عدم قطعیت در زمان پاسخگویی می‌شود. به دلیل نیاز به پاسخ‌دهی سریع و تضمین شده سیستم‌های بی‌درنگ از حافظه مجازی استفاده نمی‌کنند. در سیستم‌های بی‌درنگ سخت مهلت زمانی(deadline) باید پشتیبانی شود. در برخی کاربردها (مثل کنترل صنعتی) در کامپیوترها از سیستم عامل استفاده نمی‌شود. از آنجا که در سیستم‌های کنترل صنعتی برنامه می‌بایست در اسرع وقت در مقابل یک اتفاق، از خود عکس العمل نشان دهد، وجود واسط سیستم عامل باعث کند شدن مراحل می‌گردد. البته در سیستم‌های بی‌درنگ سخت می‌تواند سیستم عامل باشد، اما سیستم عامل باید با شرایط سیستم‌های بی‌درنگ سخت سازگار باشد یعنی سیستم عامل هم بی‌درنگ

khalilifar.ir

باشد. سیستم‌عامل بی‌درنگ نوعی سیستم‌عامل است که در آن، زمان پارامتر کلیدی است. سیستم بی درنگ به سیستمی گفته می شود که درستی اجرای یک عملیات در آن تنها به درست بودن عملیات از نظر منطقی بستگی نداشته باشد بلکه اجرای آن عملیات در یک بازه زمانی مشخص نیز در درستی اجرای عملیات در نظر گرفته شود. در سیستم‌های بی‌درنگ سخت (hard real-time) یا به عبارتی سیستم‌های بی‌درنگ بدون وقفه (immediate real-time) پایان اجرای یک عملیات پس از ضرب‌الاجل بی‌فایده تلقی می‌شود و نوشدارو پس از مرگ سهراب است.

اما در سیستم‌های بی‌درنگ نرم، زیر پاگذاشتن ضرب‌الاجل‌ها با اینکه نامطلوب است اما قابل تحمل است. یعنی می‌توان با چند لحظه تاخیر نیز کنار آمد. مثل رزرو بلیط هواپیما در صورت خالی شدن لحظه‌ای یک صندلی، خوب است اولویت و حق رعایت شود و فورا رزرو انجام شود، ولی اگر هم ضرب‌الاجل‌ها گاها رعایت نشد خیلی هم فاجعه بار نیست و چنین تاخیری قابل تحمل است یا هشدار پیام خالی شدن کاغذ یک دستگاه فتوکپی، خوب است پیام خالی شدن کاغذ فورا اعلام شود، ولی اگر هم ضرب‌الاجل‌ها گاها رعایت نشد خیلی هم فاجعه بار نیست و چنین تاخیری قابل تحمل است. سیستم‌های پخش زنده صدا و تصویر نیز معمولاً سیستم‌های بی‌درنگ نرم هستند که در صورت عدم پاسخگویی سیستم در ضرب‌الاجل با پایین آوردن کیفیت صدا و تصویر وضعیت را مدیریت می‌کنند. نتیجه اینکه در سیستم‌های بی‌درنگ نرم، رعایت مهلت زمانی مطلوب است، ولی اجباری نیست و تضمین هم نمی‌شود. به بیان دیگر سیستم تلاش می‌کند که کار خود را در مهلت زمانی خاص انجام دهد، ولی اجباری هم در این کار نیست. و حتی انجام کار پس از پایان مهلت زمانی، بازهم معنا دارد. پس در سیستم بی‌درنگ سخت احتمال تاخیر زمانی تا پس از مهلت زمانی به طور قطعی و تضمین شده وجود ندارد. اما در سیستم بی‌درنگ نرم احتمال تاخیر زمانی تا پس از مهلت زمانی وجود دارد.

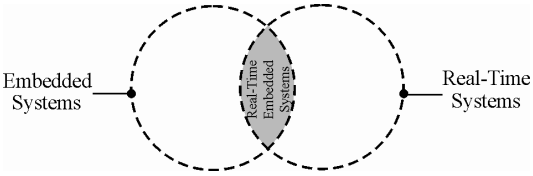
توجه: می‌توان گفت یک سیستم بی‌درنگ سخت تضمین می‌کند که کارها و وظایف بحرانی به موقع انجام شود، اما در یک سیستم بی‌درنگ نرم، یک وظیفه بحرانی نسبت به سایر وظایف اولویت خیلی بالاتری دارد و تا پایان تکمیل شدنش این ارجحیت را حفظ می‌کند. در واقع در سیستم های بی‌درنگ سخت پس از پایان مهلت زمانی، ادامه و تکمیل یک کار، دیگر معنی ندارد. از آنجا که سیستم‌های بی‌درنگ نرم مهلت زمانی(deadline) را پشتیبانی نمی‌کنند، استفاده آنها در کنترل صنعتی ریسک آور است. هر چند که سیستم‌های بی درنگ نرم می‌بایست پاسخی سریع داشته باشند ولی مساله پاسخ‌دهی به حادی سیستم‌های بی‌درنگ سخت نمی‌باشد.

انسان در وادی زندگی نیازهای گوناگونی دارد، یکی از نیازهای اساسی انسان، نیاز به امنیت است. اما گاهاً، ممکن است در معرض عوامل محیطی و بیرونی و یا حتی درونی امنیت انسان در شرایط هشیاری یا ناهشیاری به مخاطره بیفتد. بنابراین نیاز است تا مکانیزمی همواره هوشیار و همیشه بیدار و با اشراف لحظه به لحظه، مخاطرات پیرامون انسان را رصد و تحت کنترل خود قرار دهد تا در موقع لزوم و به صورت آنی، بی‌درنگ، در لحظه و در زمان حقیقی و واقعی (تا دیر نشده) با

khalilifar.ir

تهدید مقابله کند، نرم‌افزارهای بی‌درنگ این نگهبان همیشه هوشیار و همیشه بیدار هستند. مانند نرم‌افزارهای ترمز اتومبیل، کنترل ضربان قلب اتاق بیهوشی، کنترل فشار کابین هواپیما و ...

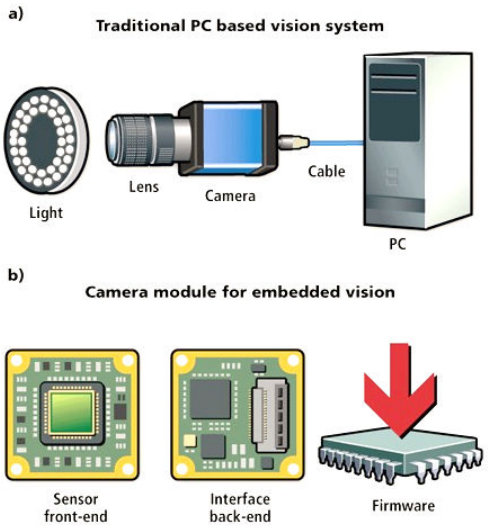
در عصر حاضر دو مفهوم جدید وجود دارد که باعث بوجود آمدن نسل جدیدی از سیستم‌های پرکاربرد شده است. این دو مفهوم یکی «سیستم‌های نهفته» یا Embedded Systems و دیگری «سیستم‌های بی‌درنگ» یا Real-Time System می‌باشند. در اغلب اوقات این دو سیستم به صورت تلفیقی و تحت عنوان «سیستم‌های نهفته بی‌درنگ» یا Real-Time Embedded Systems مورد استفاده قرار می‌گیرند. این سیستم‌ها توانایی کنترل دامنه‌ی وسیعی از وسایل مکانیکی و الکترونیکی را دارا می‌باشند. اگر به اطراف خود نگاهی بیاندازید، انواع مختلف آنها را مشاهده می‌کنید. تلویزیون، لوازم منزل و آشپزخانه، تلفن‌های همراه هوشمند، سیستم‌های کنترل اتومبیل، سیستم‌های کنترل ترافیک، سیستم‌های اتوماسیون صنعتی، ربات‌ها، موشک‌های نظامی و ... همه و همه مثال‌هایی از این سیستم‌ها می‌باشند. سیستم‌های **نهفته** که به آنها سیستم‌های **تعبیه شده** یا **توکار** نیز گفته می‌شود، سیستم‌های کامپیوتری هستند که شامل اجزای الکترونیکی و یا مکانیکی می‌باشند و وظیفه‌ی مشاهده (Monitor)، پاسخ دادن (Respond) و کنترل (Control) محیط خارجی سیستم را بر عهده دارد. این محیط خارجی بوسیله دستگاه‌های ورودی و خروجی نظیر سنسورها (Sensors)، عمل‌کننده‌ها (Actuators) و ... با سیستم‌های نهفته ارتباط دارد. علت نام‌گذاری سیستم‌های نهفته (Embedded Systems) این است که در گذشته با نگاه کردن به درون یک سیستم مثل PC و Laptop نگاه ناظر می‌توانست اجزای فیزیکی داخلی آن نظیر CPU و RAM را ببیند، اما در سیستم‌های نهفته این اجزا درون IC هستند و روی بورد الکترونیکی به نوعی پنهان شده است و به صورت مجزا قابل مشاهده نیست.



بر خلاف کامپیوترهای همه منظوره (به عنوان مثال کامپیوترهای شخصی) که برای رفع نیازهای عمومی طراحی شده‌اند، سیستم‌های نهفته به گونه‌ای طراحی می‌شوند که برای یک کاربرد خاص با **کمترین هزینه**، **بهترین کارایی** را از خود نشان دهند. بنابراین مشخصه‌ی کلیدی، سیستم‌های نهفته، طراحی اختصاصی برای انجام یک کار مشخص است. از آنجاکه سیستم‌های نهفته برای یک کار مشخص اختصاص یافته‌اند، مهندسین طراح می‌توانند محصول را برای کاهش اندازه و قیمت بهینه کرده و اطمینان‌پذیری و کارایی آنرا بالا ببرند، برخی از سیستم‌های نهفته با بهره‌گیری از مزیت‌های تولید با تیراژ بالا و به تبع مقرون به صرفه‌بودن هزینه‌های تولید، به شکل انبوه تولید شده‌اند. امروزه درون اکثر وسایل و دستگاه‌های پیرامون ما (خودپرداز، تلفن همراه، اتومبیل و

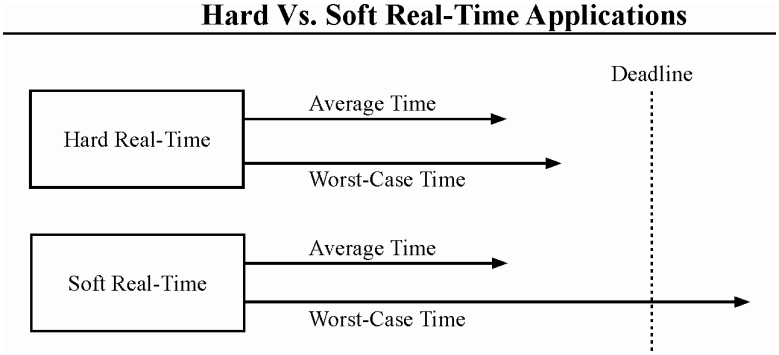
khalilifar.ir

ماشین لباسشویی) سیستم نهفته قرار دارد. شکل زیر یک سیستم امنیتی را در دو شکل نهفته و PC Based (مبتنی بر PC) نشان می‌دهد.



در علم کامپیوتر، محاسبات بی‌درنگ (RTC: Real –Time Computing) و یا محاسبات واکنشی (Reactive Commuting)، سیستم‌های نرم‌افزاری و سخت افزاری را توصیف می‌کند که زمان در آنها اهمیت دارد و بایستی این تضمین را حاصل کنند که خروجی صحیح سیستم، در یک بازه‌ی مشخصی حتما تولید شود، چرا که تولید این خروجی در خارج از این بازه زمانی، حتی اگر صحیح نیز باشد، دیگر مطلوب نخواهد بود. این آستانه‌های زمانی، سرحد زمانی یا Deadlne نامیده می‌شود. پاسخ‌های سیستم‌های بی‌درنگ به رویدادها اغلب در حدود میلی ثانیه و میکروثانیه هستند.

همانطور که گفتیم سیستم‌های بی‌درنگ در عمل بر دو نوع تقسیم‌بندی می‌شوند. یکی سیستم‌های بی‌درنگ نرم (Soft) و دیگری سیستم‌های بی‌درنگ سخت (Hard) که در شکل زیر قابل مشاهده است:



khalilifar.ir

در یک سیستم **بی‌درنگ سخت**، زمان پاسخ به رویداد هم در حالت میانگین و هم در بدترین حالت هر دو کمتر از زمان Deadline هستند، اما در سیستم‌های **بی‌درنگ نرم**، زمان پاسخ به رویداد فقط در حالت میانگین کمتر از زمان Deadline است، و در بدترین حالت حتی ممکن است بیشتر از زمان Deadline باشد و از Deadline هم عبور کند. در حقیقت هنگام طراحی یک سیستم بی‌درنگ از نوع سخت باید دقیق و سخت‌گیرانه عمل کرد، اما در طراحی سیستم بی‌درنگ از نوع نرم نیازی به این سخت‌گیری زیاد نمی‌باشد.

مفهوم سیستم بی‌درنگ نهفته (Real Time Embedded Systems)، یک مفهوم علمی است که علوم متفاوتی را الزاما در بر می‌گیرد. این سیستم‌ها، همان طور که از نامش پیداست، تلفیقی از دو سیستم نهفته و بی‌درنگ است. یک سیستم نهفته بی‌درنگ را معمولا اینگونه تعریف می‌کنند، «سیستمی که محیط را از طریق دریافت داده‌ها، پردازش آنها و سپس برگرداندن سریع تاثیر پردازش این داده‌ها به محیط، کنترل می‌کند.»

توجه: در سیستم عامل‌های عمومی و همه منظوره یا (General Purpose Operating System) GPOS کاربر هیچ گونه کنترلی بر تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها ندارد، ولی در سیستم عامل‌های بی‌درنگ یا RTOS (Real Time Operating System) به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین **اولویت‌ها، مهلت‌ها** و **کنترل زمان‌بندی فرآیندها و نخ‌ها** داده شود.

صورت سوال به این شکل است:

در سیستم‌های تعبیه شده بی‌درنگ سخت که پاسخ در زمانی مشخص باید تضمین شود، کدام روش نگاشت ریسمان‌های کاربر به ریسمان‌های سیستمی، مناسب است؟

۱) یک به یک

گزینه اول پاسخ سوال نیست، زیرا در روش یک به یک فقط زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته وجود دارد و زمان‌بند چند نخی در سطح کاربر در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل نه تنها فرآیندها را می‌شناسد بلکه از وجود نخ‌های داخل یک فرآیند نیز در صورت وجود نخ آگاه است. در واقع مدیریت نخ‌ها و زمان‌بند چندنخی (نخ‌های فرآیندهای کاربر و نخ‌های فرآیندهای سیستم عامل) در سطح هسته، توسط هسته سیستم عامل انجام می‌گردد و کاربر هیچ دیدی از این کار ندارد. در سیستم‌های بی‌درنگ به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها داده شود. در روش یک به یک، سطح هسته وجود دارد و به تبع همروندی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود دارد که این امر منجر به این می‌شود که زمان پاسخ کوتاه باشد و پاسخ قبل از اتمام مهلت زمانی تولید شود. اما در روش یک به یک، سطح کاربر وجود ندارد، که این امر

khalilifar.ir

منجر به این می‌شود که اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده نشود.

۲) چند به یک

گزینه دوم پاسخ سوال نیست، زیرا در روش چند به یک فقط زمان‌بند پردازنده و زمان‌بند چند نخی در سطح کاربر وجود دارد و زمان‌بند چند نخی در سطح هسته در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل فقط فرآیندها را می‌شناسد و هیچ اطلاعاتی از نخ‌ها ندارد. در واقع اولویت‌بندی نخ‌ها، مدیریت نخ‌ها و زمان‌بند چندنخی در سطح کاربر و توسط یک بسته نرم‌افزاری انجام می‌گردد. بدین معنی که نخ‌ها را برنامه‌نویس مشخص می‌کند و مدیریت آن‌ها را نیز بر عهده می‌گیرد. در سیستم‌های بی‌درنگ به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها داده شود. در روش چند به یک، سطح هسته وجود ندارد و به تبع همروندی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود ندارد که این امر منجر به این می‌شود که زمان پاسخ طولانی شود و به تبع این احتمال وجود دارد که پاسخ بعد از اتمام مهلت زمانی تولید شود. هرچند که در روش چند به یک، سطح کاربر وجود دارد، که این امر منجر به این می‌شود که اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده شود.

۳) چند به چند

گزینه سوم پاسخ سوال نیست، زیرا روش چند به چند از اجتماع دو روش سطح کاربر و هسته ابداع گردیده است. در روش چند به چند علاوه بر زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته سیستم عامل، زمان‌بند چند نخی نیز در سطح کاربر برای زمان‌بندی و اولویت‌دهی نخ‌های فرآیند کاربر وجود دارد. در واقع زمان‌بندی نخ‌های فرآیندهای سیستم عامل، توسط هسته سیستم عامل و زمان‌بندی نخ‌های فرآیندهای کاربر، توسط برنامه کاربر انجام می‌گردد، که این امر منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد. در این روش، زمان‌بند چند نخی در سطح کاربر، نخ‌های کاندید خود را از میان نخ‌های متعدد در فرآیندهای مختلف کاربر براساس یک الگوریتم خاص انتخاب و تحویل یک زمان‌بند چند نخی در سطح هسته می‌دهد. در واقع انتخاب نخ‌های کاندید (زمان‌بندی) در فرآیندهای کاربر به خود کاربر واگذار شده است که همانطور که گفتیم منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد.

در سیستم‌های بی‌درنگ به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها داده شود. در روش چند به چند، سطح هسته وجود دارد و به تبع همروندی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود دارد که این امر منجر به این می‌شود که زمان پاسخ کوتاه باشد و پاسخ قبل از اتمام مهلت زمانی تولید شود. اما یک ریسک دارد اینکه

khalilifar.ir

در روش چند به چند برنامه‌نویس به تعداد دلخواه نخ ایجاد می‌کند، اما ممکن است تعداد LWP کمتری به آنها منتسب شود تا سربار هسته کمتر شود. زیرا در روش چند به چند نخ‌های سطح کاربر به تعداد کم‌تر یا مساوی از نخ‌های سطح هسته نگاشت می‌شود. برای مثال اگر یک فرآیند پنج درخواست (نخ) همزمان داشته باشد، اگر چهار LWP داشته باشیم، یکی از درخواست‌ها (نخ‌ها) تا بازگشت نتیجه یکی از چهار درخواست (نخ) دیگر از سطح هسته به جریان نخواهد افتاد و این یعنی تاخیر و ممکن است زمان پاسخ طولانی شود و به تبع این احتمال وجود دارد که پاسخ بعد از اتمام مهلت زمانی تولید شود. هرچند که در روش چند به چند، سطح کاربر وجود دارد، که این امر منجر به این می‌شود که اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده شود.

۴) دوسطحی

گزینه چهارم پاسخ سوال است، زیرا روش دو سطحی به نوعی روش **چند به چند پلاس** است، یعنی روش دوسطحی بهبود یافته روش چند به چند است، در واقع روش دوسطحی علاوه بر اینکه روش چند به چند را در خود دارد، جهت بهبود کارایی روش یک به یک را نیز در خود قرار داده است. در سیستم‌های بی‌درنگ به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها داده شود. در روش دوسطحی (Two-Level Model) یا همان چند به چند پلاس، سطح هسته وجود دارد و به تبع همروندی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود دارد که این امر منجر به این می‌شود که زمان پاسخ کوتاه باشد و پاسخ قبل از اتمام مهلت زمانی تولید شود. اما یک ریسک دارد اینکه در روش چند به چند برنامه‌نویس به تعداد دلخواه نخ ایجاد می‌کند، اما ممکن است تعداد LWP کمتری به آنها منتسب شود تا سربار هسته کمتر شود. زیرا در روش چند به چند نخ‌های سطح کاربر به تعداد کم‌تر یا مساوی از نخ‌های سطح هسته نگاشت می‌شود. برای مثال اگر یک فرآیند پنج درخواست (نخ) همزمان داشته باشد، اگر چهار LWP داشته باشیم، یکی از درخواست‌ها (نخ‌ها) تا بازگشت نتیجه یکی از چهار درخواست (نخ) دیگر از سطح هسته به جریان نخواهد افتاد و این یعنی تاخیر و ممکن است زمان پاسخ طولانی شود و به تبع این احتمال وجود دارد که پاسخ بعد از اتمام مهلت زمانی تولید شود. اما از آنجا که روش دوسطحی علاوه بر داشتن روش چند به چند، روش یک به یک را نیز دارد، می‌تواند در مواقع لزوم برای پاسخ سریع از روش یک به یک نیز استفاده نماید، زیرا در روش یک به یک هر رشته نخ کاربر به یک رشته نخ هسته نگاست می‌شود. همچنین در روش دوسطحی (Two-Level Model) یا همان چند به چند پلاس، سطح کاربر وجود دارد، که این امر منجر به این می‌شود که اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده شود. روش دوسطحی در سیستم عامل‌هایی نظیر IRIX، HP-UX و

khalilifar.ir

True64nIx پشتیبانی می‌شود. نسخه‌های قبل از Solaris9 نیز مدل دوسطحی را پشتیبانی می‌کنند، اما Solaris9 از روش یک به یک استفاده می‌کند.

۸- گزینه (۲) صحیح است.

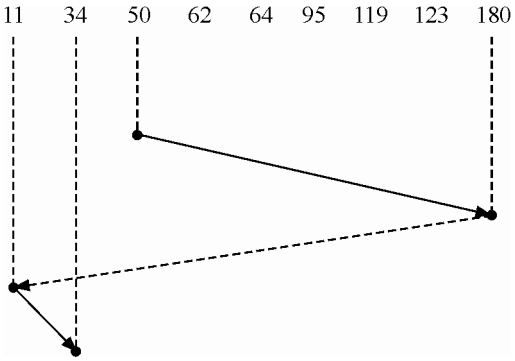
الگوریتم C-LOOK همانند روش LOOK می‌باشد، اما هد فقط در یک جهت حرکت به درخواست‌ها سرویس می‌دهد (در واقع در مسیر برگشت به هیچ درخواستی سرویس نمی‌دهد). در این روش، هد در یک جهت درخواست‌ها را سرویس داده و وقتی به آخرین درخواست رسید، سریعاً به اولین درخواست دیسک برگشته و دوباره در همان جهت شروع به سرویس دادن می‌کند. با این روش نیازهای درونی، بیرونی و میانی به یک اندازه انتظار می‌کشند.

توجه: مطابق صورت سؤال، هد روی شیار 50 دیسک قرار دارد و به سمت شیار 199 یعنی در جهت افزایش شیار مطابق فرض سوال در حال حرکت است. بنابراین داریم:

$$50 \xrightarrow{12} 62 \xrightarrow{2} 64 \xrightarrow{31} 95 \xrightarrow{24} 119 \xrightarrow{4} 123 \xrightarrow{57} 180 \xrightarrow{169} 11 \xrightarrow{23} 34$$

مجموع جابجایی بین شیارها با زمان‌بندی C-LOOK :

$$(180-50)+(180-11)+(34-11)=130+169+23=322$$



۹- گزینه (۱) صحیح است.

ابتدا وضعیت سیستم را قبل از اجابت درخواست فرآیند P_1 بررسی می‌کنیم:

با توجه به ماتریس تخصیص منابع (Allocation) و منابع در دسترس یعنی بردار Available، منابع اولیه را بدست می‌آوریم، برای یافتن منابع اولیه ابتدا برای هر منبع، منابع تخصیص یافته به هر فرآیند را با هم جمع کرده و سپس با منابع در دسترس یعنی بردار Available جمع می‌نماییم.

منبع اولیه A $= 1+(1+1)=3$

منبع اولیه B $= 5+(3+6)=14$

منبع اولیه C $= 2+(1+5+3)=11$

khalilifar.ir

بنابراین منابع اولیه به صورت زیر است:

منابع اولیه	3	14	11

براساس رابطه زیر داریم:

$$Need = MAX - Allocation$$

فرآیند	Need		
	A	B	C
P_0	0	0	0
P_1	0	7	5
P_2	1	0	0
P_3	0	0	2

=

فرآیند	MAX		
	A	B	C
P_0	0	0	1
P_1	1	7	5
P_2	2	3	5
P_3	0	6	5

−

فرآیند	Allocation		
	A	B	C
P_0	0	0	1
P_1	1	0	0
P_2	1	3	5
P_3	0	6	3

Available	1	5	2

$$(1,5,2) \xrightarrow[+(0,0,1)]{P_0} (1,5,3) \xrightarrow[+(0,6,3)]{P_3} (1,11,6) \xrightarrow[+(1,3,5)]{P_2} (2,14,11)$$

$$\xrightarrow[+(1,0,0)]{P_1} (3,14,11)$$

به این ترتیب و با توجه به ماتریس‌های فوق، دنباله‌ی امنِ $< P_0, P_3, P_2, P_1 >$ (از چپ به راست) برای این سیستم موجود است. بنابراین قبل از اجابت درخواست فرآیند P_1، سیستم در حالت امن قرار دارد و گزینه‌های سوم و چهارم نادرست هستند.

الگوریتم درخواست منبع بانکداران

هنگامی که فرآیند P_i، بردار Request[i] را به عنوان اعلام نیاز به منابع مطرح می‌کند، باید این درخواست مطابق مراحل زیر بررسی گردد:

۱) اگر $Request[i] \le Need[i]$ است به مرحله بعدی برو، در غیر اینصورت خطای تقاضای بیش از نیاز اعلام می‌شود. در واقع در این مرحله بررسی می‌شود که اگر فرآیند درخواست بیش از آن چه که در ابتدا اعلام نیاز کرده است را دارد، با این درخواست مخالفت شود.

۲) اگر $Request[i] \le Availabe[i]$ است به مرحله 3 برو، در غیر اینصورت منبع کافی جهت تخصیص به این فرآیند وجود نداشته و این فرآیند باید منتظر بماند.

۳) با فرض این که منابع مورد نیاز به این فرآیند اختصاص می‌یابد، در این صورت با اصلاح ماتریس‌ها و بردارها یک الگوریتم تشخیص وضعیت امن اجرا می‌شود. با اجرای این الگوریتم اگر سیستم در وضعیت امن باقی بماند، این منابع به فرآیند اختصاص می‌یابند، در غیراینصورت اگر با این تخصیص سیستم به وضعیت ناامن وارد شود، ماتریس‌ها و بردارها به حالت قبل بازگردانده شده و فرآیند تا آزاد شدن منابع بیشتر منتظر می‌ماند.

khalilifar.ir

اصلاح ماتریس‌ها و بردارها در صورت تخصیص به صورت زیر انجام می‌شوند:

Available (new) = Available (old) – Request[i]

Need (new) [i] = Need (old) [i] – Request[i]

Allocation (new) [i] = Allocation (old) [i] + Request[i]

مطابق فرض صورت سؤال، اگر در این وضعیت، درخواستی برای صفر واحد دیگر از منبع A، پنج واحد دیگر از منبع B و دو واحد دیگر از منبع C توسط فرآیند P_1 صادر شود، مطابق الگوریتم درخواست منبع بانکداران، ابتدا بررسی می‌شود که آیا درخواست فرآیند، کم‌تر از نیاز اعلام شده آن است یا خیر.

$$Request[P_1] \le Need[P_1]$$

$$[0,5,2] \le [0,7,5]$$

چون شرط اول برقرار است، به سراغ شرط دوم می‌رویم.

در شرط دوم بررسی می‌شود که آیا درخواست داده شده، کمتر از منابع آزاد سیستم است یا خیر.

$$Request[P_1] \le Available$$

$$[0,5,2] \le [1,5,2]$$

چون هر دو شرط برقرار است، سراغ مرحله سوم از الگوریتم می‌رویم.

در این مرحله، با فرض اینکه درخواست داده شده، اعمال شود، ماتریس‌ها به صورت زیر بروزرسانی می‌شوند:

فرآیند	Need (new)		
	A	B	C
P_0	0	0	0
P_1	0	②	③
P_2	1	0	0
P_3	0	0	2

=

فرآیند	MAX		
	A	B	C
P_0	0	0	1
P_1	1	7	5
P_2	2	3	5
P_3	0	6	5

–

فرآیند	Allocation (new)		
	A	B	C
P_0	0	0	1
P_1	1	⑤	②
P_2	1	3	5
P_3	0	6	3

با توجه به ماتریس تخصیص منابع **جدید** (Allocation(new)) و منابع اولیه، بردار Available(new) را بدست می‌آوریم، برای یافتن بردار Available(new) ابتدا برای هر منبع، منابع تخصیص‌یافته به هر فرآیند را با هم جمع کرده و سپس از منابع اولیه کسر می‌نماییم.

منبع موجود A $= 3-(1+1) = 1$

منبع موجود B $= 14-(5+3+6) = 0$

منبع موجود C $= 11-(1+2+5+3) = 0$

بنابراین بردار Available(new) به صورت زیر است:

Available(new)	①	⓪	⓪

مطابق رابطه زیر نیز می‌توان Available (new) را محاسبه نمود:

khalilifar.ir

Available (new) = Available (old) – Request $[P_1]$

Available (new) = [1,5,2] – [0,5,2] = [1,0,0]

$$(1,0,0) \xrightarrow[+(0,0,1)]{P_0} (1,0,1) \xrightarrow[+(1,3,5)]{P_2} (2,3,6) \xrightarrow[+(1,5,2)]{P_1} (3,8,8) \xrightarrow[+(0,6,3)]{P_3} (3,14,11)$$

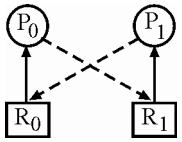
به این ترتیب و با توجه به ماتریس‌های فوق، دنباله‌ی امنِ $< P_0, P_2, P_1, P_3 >$ (از چپ به راست) برای این سیستم موجود است. بنابراین بعد از اجابت درخواست فرآیند P_1، سیستم بازهم در حالت امن قرار دارد و گزینه دوم نیز نادرست و به تبع گزینه‌ی اول درست و پاسخ سوال است.

توجه: ناامنی نه به معنی وقوع بن‌بست در گذشته و نه به معنی وقوع حتمی بن‌بست در آینده است، بلکه به معنی احتمال وقوع بن‌بست در آینده است.

توجه: در حالت ناامن نیز ممکن است، فرآیندها در موقعیت رهاسازی منابع در اختیار خود قرار بگیرند و سیستم به بن‌بست نرسد. به بیان دیگر در حالت ناامن وقوع بن‌بست قطعی نیست. اما اگر در حالت ناامن فرآیندها علاوه بر نگهداری منابع تحت تملک خود، منابع دیگری را مورد درخواست قرار دهند، در این حالت فرآیندهایی که نیاز آن‌ها برطرف نمی‌شود یک به یک در صف انتظار قرار می‌گیرند و همچون سرنوشت‌های به هم گره خورده، تا ابد منتظر یکدیگر باقی می‌مانند، یعنی بن‌بست رخ داده است.

مثال: در یک سیستم تعداد منبع اولیه R_0 برابر یک و تعداد منبع اولیه R_1 برابر یک است. اگر فرآیند P_0 یک منبع R_0 را در تملک داشته باشد و برای اتمام، نیاز به یک منبع R_1 نیز داشته باشد و همچنین فرآیند P_1 یک منبع R_1 را در تملک داشته باشد و برای اتمام، نیاز به یک منبع R_0 نیز داشته باشد، وضعیت سیستم را در این شرایط بررسی کنید:

حل: ماتریس و گراف منابع و فرآیندها به صورت زیر است:



فرآیند	MAX	
	R_0	R_1
P_0	1	1
P_1	1	1

=

فرآیند	Allocation	
	R_0	R_1
P_0	1	0
P_1	0	1

+

فرآیند	Need	
	R_0	R_1
P_0	0	1
P_1	1	0

با توجه به ماتریس تخصیص منابع (Allocation) و منابع اولیه، بردار Available را بدست می‌آوریم:

منبع موجود $R_0 = 1 - (1) = 0$

khalilifar.ir

R_1 منبع موجود $= 1 - (1) = 0$

بنابراین بردار Available به صورت زیر است:

Available	0	0

مشاهده می‌شود که با بردار Available موجود، نمی‌توان نیاز هیچ فرآیندی را با توجه به ماتریس Need مرتفع ساخت، بنابراین توالی امن یافت نمی‌شود و سیستم در یک وضعیت ناامن قرار دارد و احتمال وقوع بن‌بست وجود دارد.

توجه: در این شرایط ناامن، اگر فرآیند P_0 یا P_1 و یا هر دو، در موقعیت رهاسازی منابع در اختیار خود قرار بگیرند، یعنی رهاسازی کنند و سپس درخواست منابع باقی‌مانده خود را صادر کنند، بن‌بست رخ نمی‌دهد.

توجه: در این شرایط ناامن، اگر فرآیند P_0 در عین حال اینکه منبع R_0 را تحت تملک و نگهداری خود دارد، منبع R_1 را نیز درخواست کند، از آن‌جا که منبع R_1 در اختیار فرآیند P_1 می‌باشد، فرآیند P_0 در صف انتظار، می‌خوابد (نگهداری و انتظار) حال اگر در ادامه ماجرا، فرآیند P_1 نیز همین رویه را در پیش گیرد، یعنی فرآیند P_1 نیز در عین حال اینکه منبع R_1 را تحت تملک و نگهداری خود دارد، منبع R_0 را نیز درخواست کند، از آن‌جا که منبع R_0 در اختیار فرآیند P_0 می‌باشد، فرآیند P_1 نیز در صف انتظار، می‌خوابد (نگهداری و انتظار). این سرنوشت‌های به هم گره خورده، در ادامه بن‌بست را به ارمغان می‌آورد.

توجه: از دو توجه فوق این نتیجه استنباط می‌گردد که در شرایط ناامن، **احتمال وقوع بن‌بست** وجود دارد. در واقع در شرایط ناامن وقوع بن‌بست و عدم بن‌بست به رفتار فرآیندها در آینده، بستگی دارد. یعنی فرآیندها رفتار نگهداری و درخواست را در پیش گیرند و یا رفتار رهاسازی و درخواست را.

۱۰- گزینه (۳) صحیح است.

روش Worst Fit

در این روش باید کل فضای حافظه جست‌وجو شود تا همیشه بزرگترین فضای موجود را به هر فرآیند تخصیص دهیم!

در واقع ایده این روش این است که پس از تخصیص بزرگترین حفره به یک فرآیند، فضای باقیمانده آنقدر بزرگ هست که باز هم بتوان از آن استفاده کرد. در صورتی که در Best Fit، پس از آن‌که یک حفره به یک فرآیند اختصاص داده شد، فضای باقیمانده آنقدر کوچک است که بعدها به کار هیچ فرآیندی نمی‌آید.

توجه: در روش worst fit اولین پیشنهاد بزرگ برای یک فرآیند آخرین پیشنهاد هم هست چون گزینه بزرگتر دیگری ندارد و بزرگترینی که می‌توانست پیشنهاد دهد، همان اول پیشنهاد داد.

khalilifar.ir

توجه: روش‌های Best Fit و Worst Fit نسبت به روش‌های Next Fit و First Fit قدری کندتر هستند، زیرا در این دو روش تمام حافظه باید جست‌وجو شود.

توجه: در روش Best Fit حافظه پر از حفره‌های بسیار کوچکی می‌شود که به هیچ کار نمی‌آیند. در روش Worst Fit نیز ممکن است فرآیندهای بزرگ دچار گرسنگی شوند، زیرا قسمت‌های بزرگتر، زودتر تخصیص داده شده و کوچک می‌شوند.

به ترتیب از راست به چپ:

$\xleftarrow{P_4(254),\ P_3(100),\ P_2(395),\ P_1(200)}$

راست به چپ

$P_1(200)$:

بزرگترین پیشنهاد

حفره‌ها: (500) , 200 , 400 , 100

$-P_1(200)\ \downarrow$

300

$P_2(395)$:

بزرگترین پیشنهاد

حفره‌ها: 300 , 200 , (400) , 100

$-P_2(395)\ \downarrow$

5

$P_3(100)$:

بزرگترین پیشنهاد

حفره‌ها: (300) , 200 , 5 , 100

$-P_3(100)\ \downarrow$

200

$P_4(254)$:

200 , 200 , 5 , 100

پیشنهادی درخور و شایسته و به بزرگی P_4 وجود ندارد.

khalilifar.ir

تست‌های کنکور کارشناسی ارشد سال ۱۳۹۸

۱- در سیستمی هر پردازه دارای 32 صفحه و اندازه‌ی هر صفحه 4KB است. طول آدرس فیزیکی در این سیستم 22 بیت است. طول آدرس منطقی و اندازه‌ی حافظه اصلی در این سیستم به ترتیب از راست به چپ کدام است؟ (مهندسی کامپیوتر – دولتی ۹۸)

۱) 22 بیت – 2^{12} کیلوبایت.
۲) 22 بیت – 2^{22} کیلوبایت.
۳) 17 بیت – 2^{12} کیلوبایت.
۴) 17 بیت – 2^{22} کیلوبایت.

۲- یک سیستم عامل 3 پردازه دارد که هر کدام از آنها 2 واحد از منبع R را نیاز دارند. کمترین تعداد واحدهای R چه مقداری باشد تا بن‌بست رخ ندهد؟ (مهندسی کامپیوتر – دولتی ۹۸)

۱) 3 ۲) 4 ۳) 5 ۴) 6

۳- در یک سیستم تک پردازنده، چنانچه یک برنامه که نیاز به 30 میلی‌ثانیه کار ورودی، 20 میلی‌ثانیه کار پردازش و 40 میلی‌ثانیه کار خروجی دارد، به تعداد بسیار زیاد به صورت چند برنامگی اجرا شود، در بهترین حالت، بهره‌وری CPU کدام است؟ (پردازنده‌های ورودی و خروجی از پردازنده اصلی مجزا هستند.) (مهندسی کامپیوتر – دولتی ۹۸)

۱) $\frac{1}{90}$ ۲) $\frac{20}{90}$ ۳) $\frac{20}{70}$ ۴) $\frac{20}{40}$

۴- در خصوص الگوریتم زیر، که برای پیاده‌سازی ناحیه بحرانی بین دو پردازه i و j ارائه شده است، کدام مورد درست است؟ (الگوریتم برای پردازه i است و مشابه آن برای j هم وجود دارد.)

(مهندسی کامپیوتر – دولتی ۹۸)

```
while(true){
  Flag[i] = true;
  turn = j
  while(Flag[i] && turn == j)
/* Critical Section */
Flag[i] = false
}
```

۱) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود دارد.
۲) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود ندارد.
۳) انحصار متقابل ندارد، پیشرفت دارد، انتظار محدود ندارد.
۴) انحصار متقابل ندارد، پیشرفت ندارد، انتظار محدود ندارد.

---

khalilifar.ir

۵- در سیستمی 4 پردازه(Process) و 5 منبع یکسان وجود دارد. اگر هر پردازه حداکثر به 2 منبع نیاز داشته باشد، کدام مورد درست است؟ (مهندسی IT – دولتی ۹۸)

۱) حتما در این سیستم بن‌بست رخ می‌دهد.
۲) ممکن است در این سیستم بن‌بست رخ دهد.
۳) هیچگاه در این سیستم بن‌بست رخ نمی‌دهد.
۴) رخ دادن بن‌بست به ترتیب درخواست منابع بستگی دارد.

۶- متوسط زمان انتظار برای پردازه‌های داده شده در حالتی که از الگوریتم «اول-کمترین-زمان» (Shortest Job First) قبضه‌شدنی استفاده می‌کنیم، کدام است؟ (مهندسی IT – دولتی ۹۸)

پردازه	زمان ورود	زمان مورد استفاده از CPU
P_1	2	5
P_2	3	13
P_3	0	8
P_4	5	4
P_5	1	10

۱) 10.2 ۲) 10.6 ۳) 12.75 ۴) 18.2

۷- در یک سیستم عامل در کدام حالت فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد؟ (مهندسی IT – دولتی ۹۸)

۱) اندازه صفحه کوچک شود.
۲) پردازه CPU-bound باشد.
۳) پردازه IO-bound باشد.
۴) محلی بودن ارجاع‌ها در پردازه بیشتر شود.

۸- با اجرای کد زیر در نهایت چند پردازه خواهیم داشت؟ (مهندسی IT– دولتی ۹۸)

```
main ()
{
    for(i = 1;i < 4;i++)
        fork();
}
```

۱) 2 ۲) 4 ۳) 8 ۴) 16

۹- عامل اصلی برای استفاده سیستم عامل از وقفه کدام است؟ (مهندسی IT – دولتی ۹۸)

۱) افزایش بهره‌وری
۲) سادگی در پیاده‌سازی
۳) کاهش زمان ارتباطات
۴) جلوگیری از اتلاف در IO

---

khalilifar.ir

۱۰- در زمان‌بند غیرقبضه‌ای «بعدی- بیشترین- نسبت- زمان پاسخ» (HRRN) پردازه‌ای برای اجرا انتخاب می‌شود که بیشترین نسبت $\text{Ratio} = \frac{\text{waiting time}}{\text{CPU Burst Time}}$ را داشته باشد. چنانچه 4 پردازه به صورت جدول زیر داشته باشیم، در مقایسه با زمان‌بند غیرقبضه‌ای «اول- کمترین- زمان» (Shortest Job First) کدام مورد در خصوص متوسط زمان پاسخ درست است؟

(مهندسی IT – دولتی ۹۸)

پردازه	زمان ورود	زمان مورد استفاده از CPU (CPU Burst Time)
P_1	0	8
P_2	1	9
P_3	2	5
P_4	3	4

۱) متوسط زمان انتظار روش HRRN برابر روش SJF است.
۲) متوسط زمان انتظار روش HRRN کمتر از روش SJF است.
۳) متوسط زمان انتظار روش HRRN بیشتر از روش SJF است.
۴) متوسط زمان انتظار روش HRRN قابل محاسبه نیست.

۱۱- کدام عبارت در مورد نخ‌ها درست نیست؟ (مهندسی IT – دولتی ۹۸)

۱) نخ‌های یک پردازه، دارای برنامه مخصوص به خود هستند.
۲) نخ‌های یک پردازه، از فضای heap مشترک استفاده می‌کنند.
۳) نخ‌های یک پردازه، از فضای آدرس یکسان استفاده می‌کنند.
۴) نخ‌های یک پردازه، از یک پشته مشترک استفاده می‌کنند.

---

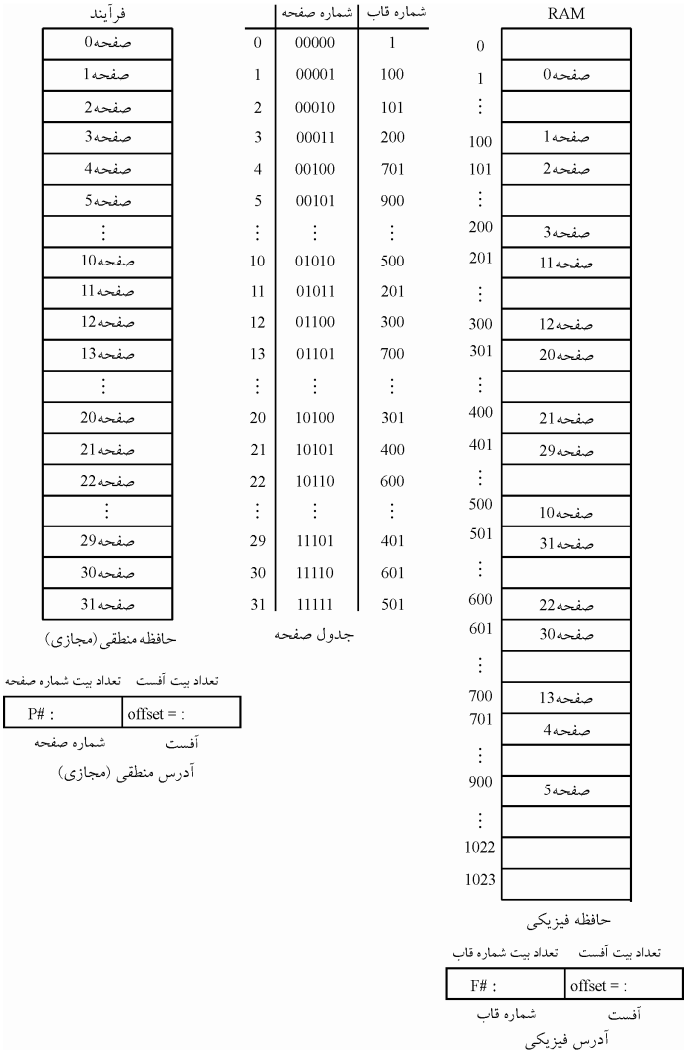
khalilifar.ir

پاسخ تست‌های کنکور کارشناسی ارشد سال ۱۳۹۸

۱- گزینه (۳) صحیح است.

در صورت سوال مطرح شده است که اندازه هر پردازه یعنی فضـای آدرس منطقـی، 32 صـفحه 4 کیلوبایتی و طول آدرس فیزیکی 22 بیت است. همچنین خواسته شده است طول آدرس منطقـی و اندازه‌ی حافظه اصلی محاسبه گردد.

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بنـدی بـه صـورت زیـر است:



khalilifar.ir

$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}} = 32$$

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}}$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^{32} = 5bit$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}}$$

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{4KB} = \log_2^{2^2 \times 2^{10}} = \log_2^{2^{12}} = 12bit$$

همچنین، اندازه آدرس‌های منطقی (مجازی) و فیزیکی به صورت زیر است:

طول آدرس منطقی = تعداد بیت شماره صفحه + تعداد بیت آفست = $5bit + 12bit = 17bit$

طول آدرس فیزیکی = تعداد بیت شماره قاب + تعداد بیت آفست

تعداد بیت شماره قاب = تعداد بیت آفست - طول آدرس فیزیکی = $22bit - 12bit = 10bit$

همچنین داریم:

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{\text{تعداد بیت آدرس منطقی}} = 2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{17} = 2^5 \times 2^{12} = 2^{17} = 128KB$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{\text{تعداد بیت آدرس فیزیکی}} = 2^{\text{تعداد بیت شماره قاب}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{22} = 2^{10} \times 2^{12} = 2^{22}\,bit = 2^{12}\,KB = 4096\,KB$$

توجه: اندازه آدرس منطقی (مجازی) و فیزیکی (حقیقی) الزاماً برابر نیست.

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^{22}}{2^{12}} = 2^{10} = 1024$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{2^{10}} = 10bit$$

اندازه جدول صفحه = تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) × عرض جدول صفحه

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد، به صورت زیر:

عرض جدول صفحه = تعداد بیت‌های شماره قاب + تعداد بیت‌های کنترلی

khalilifar.ir

در سوال تعداد بیت‌های کنترلی بیان نشده است پس 0 بیت مربوط به بیت‌های کنترلی و 10 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس: عرض جدول صفحه فوق برابر $10\text{bit} + 0\text{bit} = 10\,\text{bit}$ می‌باشد.

همانطور که گفتیم، اندازه جدول صفحه، از رابطه زیر محاسبه می‌گردد:

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

که مطابق رابطه فوق داریم:

$$\text{اندازه جدول صفحه} = 32 \times 10\,\text{bit} = 320\,\text{bit} = 40\,\text{Byte}$$

۲- گزینه (۲) صحیح است.

در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

$$\sum_{i=1}^{n} \text{Request}[i] < m + n$$: مجموع درخواست‌های فرآیندها برای منابع

$$\rightarrow 2n < m + n \rightarrow n < m$$

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد احتمال بن‌بست به صورت زیر است:

$$n \times (r-1) = m$$

حال اگر مقدار m حداقل یک واحد بیشتر از $n \times (r-1)$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

$$n \times (r-1) < m$$

$$n \times r - n < m$$

khalilifar.ir

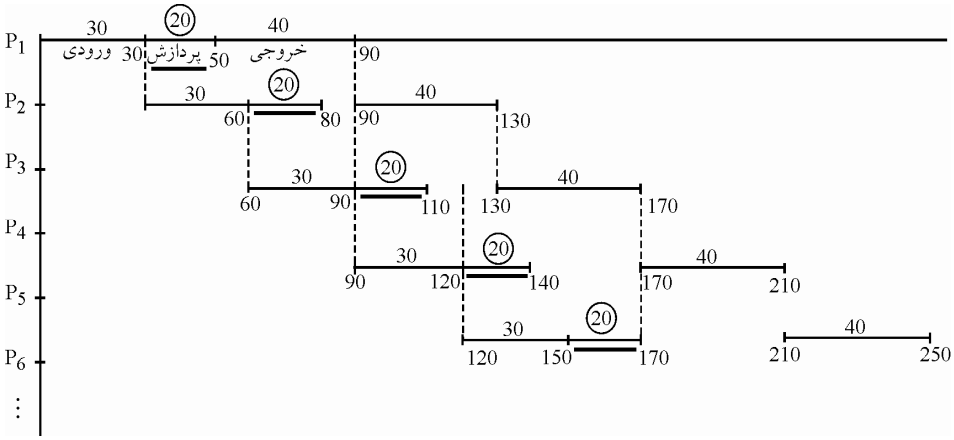
$$n \times r < m + n$$

$$\sum_{i=1}^{n} Request[i] < m + n$$

$$\longrightarrow 3 \times 2 < m + 3 \;\rightarrow\; 6 < m + 3 \;\rightarrow\; 3 < m \rightarrow R = m = 4$$

۳- گزینه (۴) صحیح است.

مطابق فرض سوال، یک برنامه نیاز به 30 میلی‌ثانیه کار ورودی، 20 میلی‌ثانیه کار پردازش و 40 میلی‌ثانیه کار خروجی دارد، نمودار گانت برای n فرآیند به صورت اجرای چند برنامگی به صورت زیر است:



تعداد فرآیندها	زمان مفید CPU	زمان پایان فرآیندها
1	20	90
2	40	130
3	60	170
4	80	210
5	100	250
⋮	⋮	⋮

$$\text{بهره‌وری} = \frac{20 \times n}{90 + (n-1) \times 40} = \frac{20 \times n}{50 + 40n}$$

$$\lim_{n \to \infty} \frac{20n}{50 + 40n} = \frac{20}{40} = \frac{1}{2}$$

khalilifar.ir

۴- گزینه () صحیح است.

شرایط رقابتی (مسابقه)

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

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

۱- شرط انحصار متقابل

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

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

khalilifar.ir

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

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

۲- شرط پیشرفت

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

۳- شرط انتظار محدود

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

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

قحطی (گرسنگی)

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

khalilifar.ir

به ناحیه‌ی بحرانی، باید محدودیتی برای تعداد دفعاتی که سایر فرآیندها می‌توانند وارد ناحیه‌ی بحرانی شوند، وجود داشته باشد تا قحطی رخ ندهد.

بن‌بست

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

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

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

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

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

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

توجه: مقادیر اولیه به صورت زیر است:

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

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

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

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

khalilifar.ir

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

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

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

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

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

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

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

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

دو خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =TRUE;
② turn= 0;

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

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:
③ while(C[0] && turn = 1) do;

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

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

P0:
/*critical_section*/

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

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

khalilifar.ir

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

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

P1:
③ while(C[1] && turn = 0) do;

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

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

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

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

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

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

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

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

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

khalilifar.ir

توجه: مقادیر اولیه به صورت زیر است:

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

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

P0:

① C[0]= TRUE;

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

P1:

① C[1]= TRUE;

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

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتی که دو فرآیند به دستور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود، اما فرآیند P1 باید در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P_0:

② turn= 1;

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

P_1:

② turn= 0;

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

P_0:

③ while(C[0] && turn = 1) do;

khalilifar.ir

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

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

P0:
/*critical_section*/

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

P_0:
③ while(C[1] && turn = 0) do;

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

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

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفق نشدند. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط دوم انحصار متقابل نیز برقرار است.

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

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

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

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۳)** در ادامه فرآیند دوم را داخل ناحیه بحرانی

khalilifar.ir

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

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

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

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

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

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

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

دو خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =TRUE;
② turn= 0;

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

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:
③ while(C[0] && turn = 1) do;

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

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

P0:
/*critical_section*/

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

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

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

khalilifar.ir

P0:
/*critical_section*/
④ C[0]= FALSE;
حال در ادامه فرآیند P0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P0:
/*remainder_section*/
همانطور که در (گام ۲) گفتیم قرار شد که همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.
(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی:
در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.
فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:
③ while(C[1] && turn = 0) do;

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

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.
همانطور که در (گام ۳) گفتیم قرار شد که فرآیند دوم را داخل ناحیه بحرانی خودش قرار بدهیم، خب نتوانستیم قرار بدهیم. پس در ادامه آزمون سوم نمی‌توانیم وارد (گام ۴) و به تبع (گام ۵) شویم. بنابراین شرط پیشرفت برقرار نیست. شرط پیشرفت در صورتی رعایت می‌شود که هر پنج گام قانون سوم به طور کامل طی شود، در غیر اینصورت شرط پیشرفت برقرار نیست. به عبارت دیگر شرط پیشرفت در صورتی برقرار نیست که فرآیندی که داخل ناحیه باقی‌مانده قرار دارد، جلوی پیشرفت (یعنی ورود به ناحیه بحرانی) فرآیند رقیب را بگیرد.

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)
(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۲) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۳) در ادامه یه آدم دیگه رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۴) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۵) در نهایت اگه همون آدم

khalilifar.ir

دوباره تونست بره داخل باجه تلفن، اونوقت شرط پیشرفت برقرار است. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن نبود، وقتی از باجه تلفن اومدی بیرون می‌تونی دوباره بری داخل باجه تلفن. به عبارت دیگر اخلاق می‌گه اگه کسی قصد ورود به باجه تلفن رو نداشته باشه یعنی کسی منتظر زدن تلفن نباشه اونوقت یه شخص دیگه‌ای می‌تونه بارها و بارها داخل باجه تلفن همگانی بره. اخلاق اینو می‌گه، اخلاق.

قانون چهارم ارسطو (آزمون گرسنگی)

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

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

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

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

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

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

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

دو خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =TRUE;
② turn= 0;

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

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:
③ while(C[0] && turn = 1) do;

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

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

khalilifar.ir

P0:

/*critical_section*/

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

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

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

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

P1:

③ while(C[1] && turn = 0) do;

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

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

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۳) می‌شویم. هم اکنون پردازنده پشت ناحیه بحرانی فرآیند P1 در یک حلقه انتظار، دچار انتظار مشغول است.

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

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

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

P0:

/*critical_section*/

④ C[0]=FALSE;

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

P0:

/*remainder_section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۴) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

khalilifar.ir

(گام ۴): فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. یعنی:
فرض کنید فرآیند P0 قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

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

شرط حلقه TRUE است، بنابراین نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.
توجه: شاید در این مرحله پیش خودتان فکر کنید که خب مثل سابق دوباره سراغ فرآیند P1 می‌رویم و با turn= 0 در خط ② راه را برای ورود فرآیند P0 باز می‌کنیم، اول اینکه خب نادرست فکر کردید، دقت کنید که شما اصلا نمی‌توانید مجددا سراغ خط ② از فرآیند P1 بروید چون قبلا از آن خط عبور کرده بودید و اگر هم سراغ فرآیند P1 بروید در ادامه همان خط ③ و حلقه while را مشاهده خواهید کرد. مانند انداختن یک سکه که بعد انداختن به سمت پایین حرکت می‌کند، مگر اینکه دوباره آنرا به سمت بالا پرتاپ کنیم. دوم اینکه از قوانین تبعیت کنید و به هیچ عنوان به مراحل آن دست نزنید و فقط و فقط مطابق قوانین حرکت کنید. اینطوری موفق می‌شوید.
همانطور که در (گام ۴) گفتیم قرار شد که فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب نشد، فرآیند اول نتوانست مجددا وارد ناحیه بحرانی خودش بشود. بنابراین گرسنگی رخ نداده است.

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

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

khalilifar.ir

قانون دوم ارسطو(آزمون بن بست)

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

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند که در این حالت بن‌بست رخ داده است.

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

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

توجه: مقادیر اولیه به صورت زیر است:

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

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

```
P0:
① C[0]= TRUE;
```

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

```
P1:
① C[1]= TRUE;
```

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین تابلوی وضعیت فرآیندها هر دو C[0] = TRUE و C[1] = TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. اما متغیر نوبت turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآیند، شماره‌ی فرآیند خود را در

khalilifar.ir

متغیر نوبت turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیره کند، فرآیندی است که شماره‌اش در متغیر نوبت turn باقی می‌ماند و دیگری اثرش در متغیر نوبت turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغییر turn گره خورده است، بنابراین فرآیندی که دیرتر متغیر نوبت turn را مقداردهی کرده است، باید صبر پیشه کند و متغیر نوبت turn را نگه‌داری کند و در حلقه‌ی انتظار بچرخد. و فرآیندی که زودتر متغیر نوبت turn را مقداردهی کرده است، وارد ناحیه بحرانی می‌شود.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتی که دو فرآیند به دستور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود اما فرآیند P1 باید در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P_0:
② turn= 1;

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

P_1:
② turn= 0;

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

P_0:
③ while(C[0] && turn = 1) do;

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

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

P_0:
/*critical_section*/

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

P_0:
③ while(C[1] && turn = 0) do;

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

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

khalilifar.ir

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

فرم ساده قانون دوم ارسطو (آزمون بن بست)

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

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

صورت سوال به این شکل است:

در خصوص الگوریتم زیر، که برای پیاده‌سازی ناحیه بحرانی بین دو پردازه i و j ارائه شده است، کدام مورد درست است؟ (الگوریتم برای پردازه i است و مشابه آن برای j هم وجود دارد.)

```
while(true){
   Flag[i] = true;
   turn = j
   while(Flag[i] && turn == j)
/*Critical Section*/
 Flag[i] = false
}
```

۱) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود دارد.

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

۲) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود ندارد.

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

۳) انحصار متقابل ندارد، پیشرفت دارد، انتظار محدود ندارد.

گزینه سوم **نادرست** است، زیرا شرط انحصار متقابل برقرار است و برآورده می‌کند، شرط پیشرفت

khalilifar.ir

برقرار نیست و برآورده نمی‌کند و شرط انتظار محدود برقرار است و برآورده می‌کند.

۴) انحصار متقابل ندارد، پیشرفت ندارد، انتظار محدود ندارد.

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

توجه: همانطور که واضح و مشخص هست، همه گزینه‌ها نادرست هستند و هیچ یک از گزینه‌ها نمی‌تواند پاسخ سوال باشد.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه دوم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

۵- گزینه (۳) صحیح است.

در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

$\sum_{i=1}^{n} \text{Request}[i] < m+n$: مجموع درخواست‌های فرآیندها برای منابع

$$\rightarrow 2n < m+n \rightarrow n < m$$

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد احتمال بن‌بست به صورت زیر است:

$$n \times (r-1) = m$$

حال اگر مقدار m حداقل یک واحد بیشتر از $n \times (r-1)$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

khalilifar.ir

$$n \times (r-1) < m$$

$$n \times r - n < m$$

$$n \times r < m + n$$

$$\sum_{i=1}^{n} Request[i] < m + n$$

$$\longrightarrow 4 \times 2 < 5 + 4 \;\rightarrow\; 8 < 9$$

۶- گزینه (۱) صحیح است.

الگوریتم SJF (Shortest Job First)

در این روش ابتدا کاری برای اجرا انتخاب می‌شود که از همه کوتاهتر باشد (زمان اجرای کمتری داشته باشد).

توجه: این الگوریتم، **SPN** (Shortest Process Next) و **SPT** (Shortest Process Time) نیز نامیده می‌شود.

توجه: **SJF** یک الگوریتم **انحصاری** (Non Preemptive) است. در سایر متون فارسی به الگوریتم انحصاری، الگوریتم «**غیرقبضه‌ای**» یا «**غیرقابل پس گرفتن**» یا «**غیرقابل تخلیه پیش هنگام**» نیز گفته می‌شود.

توجه: یک نقص عمده الگوریتمSJF این است که ممکن است باعث قحطی‌زدگی فرآیندهای طولانی شود. به این ترتیب که اگر همواره تعدادی فرآیند کوچک وارد سیستم شوند، اجرای فرآیندهای بزرگ به طور متناوب به تعویق می‌افتد. این روال حتی می‌تواند تا بینهایت ادامه یابد و هیچگاه نوبت به فرآیندهای بزرگ نرسد!!!!!

توجه: در این روش اگر دو فرآیند مدت زمان اجرای برابر داشته باشند، بر اساس FCFS زمان‌بندی می‌شوند.

توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسخ و میانگین زمان گردش کار (بازگشت) فرآیندهاست.

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآیندها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجرای فرآیندها را فقط حدس زده و به طور تقریبی بدست آورد.

الگوریتم SRT (Shortest Remaining Time)

این الگوریتم نسخه **غیرانحصاری** (Preemptive) الگوریتم SJF است. در سایر متون فارسی به الگوریتم غیرانحصاری، الگوریتم «**قبضه‌ای**» یا «**قابل پس گرفتن**» یا «**قابل تخلیه پیش هنگام**» نیز گفته می‌شود.

khalilifar.ir

در این الگوریتم اگر حین اجرای یک فرآیند، فرآیندی وارد شود که زمان اجرای کوتاه‌تری داشته باشد، پردازنده را در اختیار می‌گیرد.

توجه: این الگوریتم، **SRPT** (Shortest Remaining Processing Time)،
SRTF (Shortest Remaining Time First) و
SRTN (Shortest Remaining Time Next) نیز نامیده می‌شود.

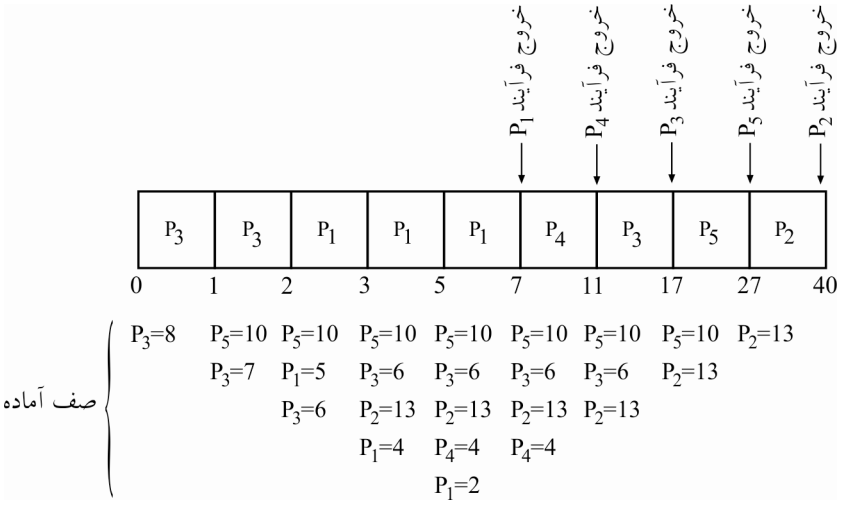
توجه: اگر لحظه ورود همه فرآیندها یکی باشد، الگوریتم SRT مشابه SJF عمل می‌کند.

توجه: در الگوریتم **SRT** نیز همانند الگوریتم SJF، احتمال وقوع **قحطی‌زدگی برای کارهای بزرگ** وجود دارد.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	2	5		
P_2	3	13		
P_3	0	8		
P_4	5	4		
P_5	1	10		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 7 - 2 = 5$

P_2 زمان بازگشت $= 40 - 3 = 37$

P_3 زمان بازگشت $= 17 - 0 = 17$

khalilifar.ir

P_4 زمان بازگشت $= 11 - 5 = 6$

P_5 زمان بازگشت $= 27 - 1 = 26$

میانگین زمان بازگشت $= ATT = \frac{5+37+17+6+26}{5} = \frac{91}{5} = 18.2$

زمان اجرای فرآیند − زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 5 - 5 = 0$

P_2 زمان انتظار $= 37 - 13 = 24$

P_3 زمان انتظار $= 17 - 8 = 9$

P_4 زمان انتظار $= 6 - 4 = 2$

P_5 زمان انتظار $= 26 - 10 = 16$

میانگین زمان انتظار $= AWT = \frac{0+24+9+2+16}{5} = \frac{51}{5} = 10.2$

میانگین زمان اجرا $= AST = \frac{5+13+8+4+10}{5} = \frac{40}{5} = 8$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

18.2= 8 + 10.2

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	2	5	0	5
P_2	3	13	24	37
P_3	0	8	9	17
P_4	5	4	2	6
P_5	1	10	16	26

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

18.2 10.2 8

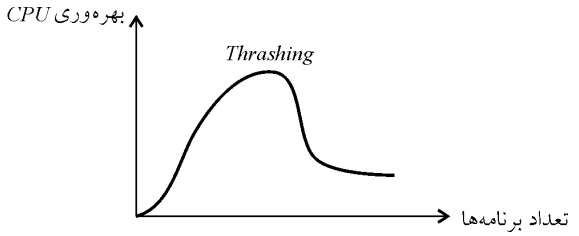
khalilifar.ir

۷- گزینه (۴) صحیح است.

اگر حافظه تخصیص داده شده به یک فرآیند، آنقدر کوچک باشد که نتواند صفحاتی که فرآیند، زیاد با آنها سروکار دارد را در خود جای دهد، سرعت اجرای فرآیند کاهش می‌یابد، زیرا این فرآیند خطاهای نقص صفحه زیادی تولید می‌کند. در این حالت مدت زمان اجرای یک فرآیند به چندین و چند برابر حالت عادی افزایش می‌یابد. اصطلاحاً به برنامه‌ای که در هر 2 یا 3 دستور خود یک خطای نقص صفحه تولید کند، **کوبیده شده (لهیده)** گویند.

توجه: فرآیندی که در حالت Thrashing واقع است، به جای اینکه زمان CPU را به اجرا اختصاص دهد، زمان زیادی را صرف انجام عملیات صفحه‌بندی می‌کند.

توجه: نسبت میزان بهره مفیدCPU با افزایش تعداد فرآیندها به صورت زیر است:



پدیده کوبیدگی

در واقع تا یک نقطه، افزایش تعداد فرآیندها (افزایش سطح چند برنامگی)، بهره‌وری CPU را افزایش می‌دهد، اما از یک نقطه به بعد، افزایش تعداد برنامه‌ها بهره‌وری پردازنده را کاهش می‌دهد. به عنوان مثال دو فرآیند A و B را در نظر بگیرید. فرض کنید برنامه A در حال اجرا به یک صفحه نیاز دارد، بنابراین بعد از یک خطای نقص صفحه، صفحه مورد تقاضایش به حافظه آورده می‌شود و به جای یک صفحه از فرآیند B در یک قاب حافظه قرار می‌گیرد، در این لحظه نوبت به اجرای برنامه B می‌رسد و برنامه B به همان صفحه قدیمی خودش نیاز دارد، بنابراین با یک خطای نقص صفحه آن صفحه را به حافظه برمی‌گرداند و این‌بار صفحه فرآیند A باید حافظه را ترک کند و الی آخر... در واقع در این حالت صفحاتی که فرآیندها زیاد با آن سروکار دارند به طور کامل در حافظه قرار ندارد و کارایی سیستم به شدت کاهش می‌یابد.

توجه: اگر عملکرد سیستم عامل را در قبال درجه چندبرنامگی سیستم بررسی کنیم، متوجه می‌شویم تحت شرایطی خاص، پدیده Thrashing به شدت تشدید می‌شود.

سناریوی زیر را در نظر بگیرید:

می‌دانیم سیستم عامل بر بهره‌وری CPU نظارت دارد و اگر بهره‌وری CPU بسیار کم باشد، درجه چندبرنامگی را با افزودن یک فرآیند جدید به سیستم افزایش می‌دهد تا بهره‌وری CPU افزایش یابد. فرض کنید در این سیستم از یک الگوریتم جایگزینی سراسری برای صفحات استفاده می‌شود که صفحات را بدون توجه به اینکه مربوط به کدام فرآیند هستند، جایگزین می‌کند. حال فرض

khalilifar.ir

کنید فرآیندی وارد یک مرحله جدید از اجرا شده و به چند صفحه جدید نیاز دارد. بنابراین وقفه‌های نقص صفحه برای این فرآیند آغاز می‌شوند و این فرآیند قاب‌های فرآیندهای دیگر را در اختیار می‌گیرد.

از طرفی فرآیندهایی که تعدادی از صفحات آن‌ها از حافظه خارج شده‌اند، به آن صفحات نیاز دارند، بنابراین وقفه‌های نقص صفحه برای آن‌ها نیز شروع می‌شود و به طور مشابه این فرآیندها قاب‌های فرآیندهای دیگر را در اختیار می‌گیرند و این مسئله برای فرآیندهای دیگر تکرار می‌شود. بنابراین فرآیندها پی‌درپی خطای نقص صفحه را تجربه می‌کنند و صفحات درگیر، مرتباً به داخل و خارج حافظه مبادله می‌شوند. به این ترتیب بهره‌وری CPU کاهش می‌یابد و زمانبند پردازنده متوجه این کاهش بهره‌وری می‌گردد و جهت افزایش بهره‌وری CPU درجه چندبرنامگی را کاهش می‌دهد.

توجه: یک راه حل کنترل فرکانس نقص صفحه یا در بدترین حالت مقابله با Trashing کنترل تعداد وقفه‌های نقص صفحه با استفاده از الگوریتم فرکانس نقص صفحه یا PFF (Page Fault Frequency) است. این الگوریتم زمان کاهش یا افزایش تعداد قاب صفحه تخصیص یافته به یک فرآیند را بیان می‌کند، اما کنترلی در مورد این که کدام صفحه باید در هنگام نقص صفحه جایگزین شود، انجام نمی‌دهد. این الگوریتم فقط اندازه مجموعه تخصیص را کنترل می‌کند. که بهتر است، این مجموعه تخصیص برابر مجموعه کاری (working set) باشد. روال کار بدین صورت است که اگر تعداد نقص‌های صفحه برای یک فرآیند افزایش یافت، باید تعدادی قاب حافظه به آن اختصاص یابد و اگر تعداد نقص‌های صفحه برای یک فرآیند از یک حد پایین کم‌تر شد، باید تعدادی قاب از آن فرآیند پس گرفته شود. نکته مهم اینجاست که اگر تعداد نقص‌های صفحه یک فرآیند بالا رفت، ولی قاب آزاد حافظه وجود نداشت، باید درجه چند برنامگی سیستم را کاهش داد و یک یا چند فرآیند را به حالت معلق در آورد تا قاب‌هایی که در اختیار دارد، آزاد شود.

مدل مجموعه کاری (Working Set Model) هر فرآیند، عبارت است از صفحاتی از آن فرآیند که اگر در حافظه قرار داشته باشند، فرآیند موردنظر کارایی و سرعت قابل قبولی دارد. در واقع اگر مجموعه کاری یک فرآیند در حافظه باشد، فرآیند با تعداد معقول و مناسبی وقفه نقص صفحه به کار خود ادامه می‌دهد. ایده مدل مجموعه کاری در واقع یک رهیافت جهت مقابله با پدیده Thrashing است. در این مدل (راهکار) از مفهوم مجموعه کاری به خوبی استفاده می‌شود. به طور خلاصه مدل مجموعه کاری بیان می‌کند که قبل از دادن نوبت اجرا به یک فرآیند، باید مجموعه کاری آن فرآیند به درون حافظه بار شود. می‌توان گفت با انتقال مجموعه کاری یک فرآیند قبل از اجرای آن به درون حافظه، نرخ خطای نقص صفحه کاهش می‌یابد.

نکته: به بار کردن صفحات یک فرآیند قبل از اجرای آن، پیش صفحه‌بندی (Prepaging) گویند.

صورت سوال به این شکل است:

khalilifar.ir

در یک سیستم عامل در کدام حالت فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد؟

۱) اندازه صفحه کوچک شود.

گزینه اول پاسخ سوال نیست، زیرا اگر اندازه صفحه کوچک شود، آنگاه تعداد صفحات یک فرآیند بیشتر می‌شود و به تبع فرکانس نقص صفحه افزایش می‌یابد. آنچه در کاهش فرکانس نقص صفحه موثر است، افزایش اندازه حافظه اصلی، کاهش درجه چندبرنامگی، افزایش اندازه صفحه است. که البته پس از وقوع Trashing، فقط راه حل‌های افزایش اندازه حافظه اصلی و کاهش درجه چندبرنامگی **موثر** هستند. بالا بودن paging به دلیل کمبود حافظه اصلی و بالا بودن درجه چندبرنامگی اتفاق افتاده است. که در این حالت کوبیدگی (Trashing) رخ داده است و به تبع آن میزان بهره‌وری CPU پایین است. بنابراین CPU کنونی اغلب بی‌کار است و حتی استفاده از CPU سریعتر هم سودی نخواهد داشت. برای کاهش پدیده کوبیدگی و به تبع افزایش بهره‌وری CPU یا باید اندازه حافظه اصلی را افزایش داد و یا درجه چندبرنامگی را کاهش داد.

۲) پردازه CPU-bound باشد.

گزینه دوم پاسخ سوال نیست، زیرا CPU-bound بودن یک پردازه ارتباطی به افزایش یا کاهش فرکانس نقص صفحه ندارد. فرآیندهای CPU-bound بیشتر وقتشان صرف پردازش در CPU می‌شود. این فرآیندها محدود به محاسبه (Compute-bound) یا محدود به CPU هستند. اغلب فرآیندها به تناوب نیازمند فوران‌های (Bursts) محاسباتی و در لابه لای آن درخواست‌های IO هستند. معمولا هر فرآیند در تکه زمان‌های کوتاه به CPU نیاز دارد و در انتهای هر یک از این تکه زمان‌ها، در انتظار وقوع یک رویداد مانند تکمیل عملیات IO بلوکه می‌شود و CPU را به طور موقت رها می‌کند و پس از وقوع رویداد مورد نظر آماده می‌شود تا CPU را برای یک تکه زمانی دیگر اخذ کند. به هر یک از این تکه‌های زمانی، CPU Burst می‌گویند. در انتهای آخرین CPU Burst، فرآیند خاتمه می‌یابد.

۳) پردازه IO-bound باشد.

گزینه سوم پاسخ سوال نیست، زیرا IO-bound بودن یک پردازه ارتباطی به افزایش یا کاهش فرکانس نقص صفحه ندارد. فرآیندهای IO-bound بیشتر وقتشان صرف انتظار در IO می‌شود. این فرآیندها محدود به IO هستند. دقت کنید که آنچه که عامل تعیین کننده CPU-bound بودن یا IO-bound بودن است، CPU Burst است و نه طول زمان IO، یعنی فرآیندهای محدود به IO به این دلیل محدود به IO هستند که آنها در بین درخواست‌های IO محاسبات چندانی را انجام نمی‌دهند و علت آن این نیست که درخواست‌های IO طولانی دارند. نکته حائز اهمیت دیگر این است که هرچه قدر سرعت CPU بیشتر شود، کار پردازش سریعتر انجام می‌شود، که در این حالت فرآیندها محدود به IO می‌شوند.

khalilifar.ir

۴) محلی بودن ارجاع‌ها در پردازه بیشتر شود.

گزینه چهارم پاسخ سوال است، زیرا اگر محلی بودن ارجاع‌ها در پردازه بیشتر شود، آنگاه فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد. زیرا هر صفحه‌ای که به حافظه اصلی آورده می‌شود، به دلیل محلی بودن ارجاع‌ها اطلاعات زیادی از همان صفحه قابل استخراج است که نتیجه آن می‌شود نقص صفحه کمتر و عدم مراجعه مکرر به صفحات بعدی که نتیجه آن افزایش نقص صفحه باشد.

به طور کلی محلیت در دو نوع، (۱) محلیت مکانی (Spatial Locality) و (۲) محلیت زمانی (Temporal Locality) وجود دارد.

محلی بودن مکانی را مانند شعاع **متغیر** یک دایره در نظر بگیرید، هرچه قدر این شعاع کوچکتر شود محلی بودن مکانی در **کناره‌ها و اطراف** بیشتر و بیشتر می‌شود و هرچه قدر این شعاع بزرگتر شود محلی بودن مکانی در **کناره‌ها و اطراف** کمتر و کمتر می‌شود. شما با هم محله‌ای‌های خود محلی‌تر مکانی هستید، اما هرچه قدر از محله و مکان خود دور می‌شوید، اهالی محله‌های دیگر با شما کمتر محلی‌تر مکانی هستند. به عبارت دیگر محلی بودن مکانی خوب می‌گوید مراجعه بعدی، در **نزدیکی** همین مراجعه فعلی است.

محلی بودن زمانی را مانند شعاع **ثابت** یک دایره در نظر بگیرید، که محدوده این شعاع هیچگاه تغییر نکند. شما با خانواده خود محلی زمانی هستید. به عبارت دیگر محلی بودن زمانی **خوب** می‌گوید مراجعه بعدی، در **محل** همین مراجعه فعلی است.

مثال: درباره **ویژگی محلیت (Locality) برنامه زیر، کدام مورد درست است؟**

```
int sum (int v[n]){
    int  i,  sum = 0
    for(i = 0;i < n;i + +)
        sum+ = v[i]
}
```

۱) متغیر sum دارای ویژگی محلیت زمانی (Temporal Locality) خوب و متغیر v دارای ویژگی محلیت زمانی بد و محلیت مکانی (Spatial Locality) خوب می‌باشد.

۲) متغیر sum دارای ویژگی محلیت زمانی (Temporal Locality) خوب و متغیر v دارای ویژگی محلیت زمانی بد و محلیت مکانی (Spatial Locality) بد می‌باشد.

۳) متغیر sum دارای ویژگی محلیت مکانی (Spatial Locality) خوب و متغیر v دارای ویژگی محلیت زمانی(Temporal Locality) بد و محلیت مکانی خوب می‌باشد.

۴) متغیر sum دارای ویژگی محلیت مکانی (Spatial Locality) خوب و متغیر v دارای ویژگی محلیت زمانی(Temporal Locality) خوب و محلیت مکانی بد می‌باشد.

khalilifar.ir

پاسخ: گزینه (۱) صحیح است.

آرایه مطرح شده در صورت سوال یعنی v[i] دارای خاصیت محلی بودن مکانی **خوب** است. عناصر آرایه v[i] در داخل یک حلقه for از اندیس صفر تا n به ترتیب و پشت سرهم خوانده می‌شوند، یعنی مراجعه بعدی، مدام در نزدیکی همین مراجعه فعلی است.

متغیر مطرح شده در صورت سوال یعنی sum دارای خاصیت محلی بودن زمانی **خوب** است. متغیر sum در داخل یک حلقه for از اندیس صفر تا n خوانده می‌شود، یعنی مراجعه بعدی، مدام در محل همین مراجعه فعلی است.

همانطور که گفتیم آرایه v[i] دارای خاصیت محلی بودن مکانی **خوب** است، اما از آنجاکه به هریک از عناصر آرایه v[i] در حرکت حلقه فقط و فقط یکبار مراجعه می‌شود، بنابراین آرایه v[i] دارای خاصیت محلی بودن زمانی **بد** است. همچنین همانطور که گفتیم متغیر sum دارای خاصیت محلی بودن زمانی **خوب** است، اما از آنجاکه به کناره‌ها و اطراف متغیر sum در حرکت حلقه مراجعه نمی‌شود، بنابراین متغیر sum دارای خاصیت محلی بودن مکانی **بد** است.

توجه: رعایت اصول ساده برنامه‌نویسی می‌تواند در افزایش سرعت اجرای فرآیندها و کاهش تعداد نقص صفحه تأثیر مستقیم داشته باشد. به عنوان مثال قطعه برنامه زیر را در نظر بگیرید که سعی دارد همه عناصر یک آرایه دو بعدی 100× 100 را با صفر پر کند:

```
for i := 1 to 100 do
    for j := 1 to 100  do
      a[j,i] := 0;
```

فرض کنید هریک از عناصر این آرایه، 2 بایتی هستند و اندازه هر صفحه در حافظه نیز 200 بایت باشد. در این صورت این آرایه 1000 عنصری، جمعاً 100 صفحه را اشغال می‌کند و از آنجا که عناصر آرایه در زبان‌های برنامه‌نویسی C و پاسکال به صورت سطری در حافظه ذخیره می‌شوند، در واقع هر سطر این آرایه در یک صفحه قرار می‌گیرد. یعنی در صفحه اول، عناصر a[1,1] تا a[1,100]، در صفحه دوم عناصر a[2,1] تا a[2,100]، تا صفحه صدم که عناصر a[100,1] تا a[100,100] قرار می‌گیرند.

با دقت در قطعه برنامه نوشته شده، مشاهده می‌شود که پردازش به صورت ستونی انجام می‌شود زیرا از اندیس حلقه بیرونی به عنوان اندیس ستون در آرایه استفاده کرده است (اندیس i). به این ترتیب اگر فقط یک صفحه برای داده‌ها در اختیار این برنامه باشد، برای انجام عملیات خود، دقیقاً 10000 خطای نقص صفحه رخ می‌دهد. زیرا هنگامی که یک صفحه به حافظه آورده شد، فقط یکی از عناصر آن پردازش می‌شود و برای عنصر بعدی یک خطای نقص صفحه رخ می‌دهد، زیرا این عنصر در صفحه بعدی قرار دارد.

به این ترتیب به ازای هر عنصر، یک خطای نقص صفحه رخ می‌دهد.

khalilifar.ir

شکل زیر گویای مطلب است:

```
for i:=1 to 100 do
    for j:=1 to 100 do
    a[ j , i ]:=0
    ستون سطر
```

توجه: هر یک از عناصر آرایه، 2 بایتی هستند و اندازه هر صفحه در حافظه نیز 200 بایت است.

	1	2	3	...	100
1	1,1	1,2	1,3	...	1,100
2	2,1	2,2	2,3	...	2,100
3	3,1	3,2	3,3	...	3,100
⋮	⋮	⋮	⋮	⋮	⋮
100	100,1	100,2	100,3	...	100,100

توجه: اندازه هر سطر ماتریس 2×100 یعنی 200 بایت است که در یک صفحه حافظه اصلی 200 بایتی قرار می‌گیرد.

i	j	
1	1	a[1,1]=0
	2	a[2,1]=0
	⋮	⋮
	100	a[100,1]=0
2	1	a[1,2]=0
	2	a[2,2]=0
	⋮	⋮
	100	a[100,2]=0
⋮	⋮	⋮
100	1	a[1,100]=0
	2	a[2,100]=0
	⋮	⋮
	100	a[100,100]=0

اما اگر برنامه به صورت زیر نوشته شود، شرایط تغییر می‌کند:

```
for i := 1 to 100 do
    for j := 1 to 100 do
        a[i, j] := 0;
```

khalilifar.ir

در این حالت هنگامی که یک صفحه به حافظه آورده می‌شود، هر 100 عنصر آن به ترتیب پردازش می‌شوند و نیازی به نقص صفحه نیست. در واقع جمعاً 100 نقص صفحه رخ می‌دهد به ازای 100 سطر آرایه.

شکل زیر گویای مطلب است:

```
for i:=1 to 100 do
    for j:=1 to 100 do
    a[ i , j ]:=0
    ستون  سطر
```

توجه: هر یک از عناصر آرایه، 2 بایتی هستند و اندازه هر صفحه در حافظه نیز 200 بایت است.

	1	2	3	...	100
1	1,1	1,2	1,3	...	1,100
2	2,1	2,2	2,3	...	2,100
3	3,1	3,2	3,3	...	3,100
⋮	⋮	⋮	⋮	⋮	⋮
100	100,1	100,2	100,3	...	100,100

توجه: اندازه هر سطر ماتریس 2×100یعنی 200 بایت است که در یک صفحه حافظه اصلی 200 بایتی قرار می‌گیرد.

i	j	
1	1	a[1,1]=0
	2	a[1,2]=0
	⋮	⋮
	100	a[1,100]=0
2	1	a[2,1]=0
	2	a[2,2]=0
	⋮	⋮
	100	a[2,100]=0
⋮	⋮	⋮
100	1	a[100,1]=0
	2	a[100,2]=0
	⋮	⋮
	100	a[100,100]=0

بنابراین همانطور که گفتیم اگر محلی بودن ارجاع‌ها در پردازه بیشتر شود، آنگاه فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد. زیرا هر صفحه‌ای که به حافظه اصلی آورده

khalilifar.ir

می‌شود، به دلیل محلی بودن ارجاع‌ها اطلاعات زیادی از همان صفحه قابل استخراج است که نتیجه آن می‌شود نقص صفحه کمتر و عدم مراجعه مکرر به صفحات بعدی که نتیجه آن افزایش نقص صفحه باشد. مثال اخیر محلیت مکانی (Spatial Locality) خوب بود.

۸- گزینه (۳) صحیح است.

یک فرآیند می‌تواند، چندین فرآیند جدید را از طریق یک فراخوان سیستمی ایجاد فرآیند در طول اجرا، ایجاد نماید. فرآیند ایجاد کننده، فرآیند پدر (Parent Process) و فرآیند ایجاد شده، فرآیند فرزند (Child Process) نامیده می‌شود. هر یک از این فرآیندهای جدید نیز می‌توانند فرآیندهای دیگر را بوجود آورند و **درختی** از فرآیندها را تشکیل دهند.

توجه: بیشتر سیستم عامل‌ها (یونیکس، لینوکس و ویندوز) فرآیندها را توسط یک مشخصه فرآیند(process identifier) یا pid به صورت یکتا که معمولا یک عدد صحصیح است، مشخص می‌سازند.

توجه: افزون بر منابع فیزیکی و منطقی که یک فرآیند فرزند پس از ایجاد بدست می‌آورد، داده و مقداردهی اولیه از فرآیند پدر به فرآیند فرزند کپی و پاس داده می‌شود.

توجه: سیستم عامل Unix و Linux برای ایجاد یک فرآیند فرزند (جدید) از فراخوان سیستمی fork استفاده می‌کند. در این سیستم عامل جهت پیاده‌سازی مفهوم حافظه مجازی و همچنین **صرفه‌جویی در مصرف حافظه**، تکنیک Copy-On-Write **می‌تواند** مورد استفاده قرار بگیرد.

توجه: Copy-On-Write یکی از فیلدهای جدول صفحه، به طول یک بیت است و هنگامی که بیش از یک فرآیند در یک صفحه باشد، این فیلد براساس تعداد فرآیندهای موجود در یک صفحه مقدار می‌گیرد.

توجه: هنگامی که فرآیند فرزند (جدید) ایجاد می‌شود، در مورد **زمان‌بندی پردازنده** و به تبع اجرای فرآیندهای پدر و فرزند، دو حالت ممکن است رخ دهد:

۱-فرآیند پدر بطور همروند در سیستم تک پردازنده‌ای و بطور موازی در سیستم چند پردازنده‌ای با فرآیند فرزند زمان‌بندی و به تبع اجرا شود.

۲-فرآیند پدر منتظر می‌ماند تا کار چند و یا همه فرزندانش تمام شود.

توجه: به طور کلی مستقل از اینکه فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** (تکنیک Copy-On-Write مورد استفاده قرار نگیرد) و یا **مشترک** (تکنیک Copy-On-Write مورد استفاده قرار بگیرد) **باشد، محتوای فرآیند فرزند** در ابتدا از نظر داده‌ها، مقدارها و کد شامل Stack، Data، Heap ، Register ، PCB و Code، یک کپی **کاملا، دقیقا و یکسان** از فرآیند پدر است، چون داده و مقداردهی اولیه از فرآیند پدر به فرآیند فرزند کپی و پاس داده می‌شود. حتی مقادیر PCB فرآیند پدر در PCB فرآیند فرزند کپی می‌شود و تنها چیزی که در محتوای PCB فرآیند پدر و PCB

khalilifar.ir

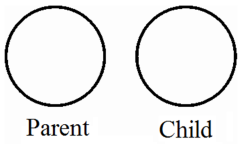
فرآیند فرزند تفاوت دارد، مقدار pid است چون هر فرآیند pid مختص به خودش را دارد، در واقع pid فرآیند پدر با pid فرآیند فرزند متفاوت است. همچنین دقت کنید که فضای آدرس PCB فرآیند پدر از فضای آدرس PCB فرآیند فرزند مستقل است، اما همانطور که گفتیم بعد از اجرای fork محتوای PCB فرآیند پدر در PCB فرآیند فرزند کپی می‌شود.

توجه: در مفهوم fork برای ایجاد یک فرآیند فرزند (جدید) می‌توان تکنیک Copy-On-Write را مورد استفاده قرار داد یا نداد. اگر تکنیک Copy-On-Write مورد استفاده قرار بگیرد، پس از دستور fork جهت صرفه‌جویی در مصرف حافظه، به جای آنکه صفحات حافظه فرآیند پدر برای فرآیند فرزند کپی شود، صفحات حافظه فرآیند پدر با فرآیند فرزند به اشتراک گذاشته می‌شود.

توجه: به تفاوت **فضای آدرس فرآیند** (ظرف فرآیند و محل ذخیره‌سازی فرآیند) و **محتوای فرآیند** (مقادیر فرآیند، داده و کد) دقت داشته باشید.

توجه: هنگامی که فرآیند فرزند (جدید) ایجاد می‌شود، در مورد **فضای آدرس** و **محتوای فرآیند پدر و فرزند**، چهار حالت ممکن است رخ دهد:

۱- اگر فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار نگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد استفاده قرار نگیرد، یعنی محتوای فرآیند پدر و فرآیند فرزند بسته به شرایط، **فقط در حد تغییر مقدار متغیرها و نه تغییر برنامه و کد تغییر کند**، آنگاه فضای آدرس فرآیند پدر و فرآیند فرزند شامل Stack ، Data ، Heap و Code کاملا مستقل و جدا از هم خواهد بود و تغییرات در محتوای فرآیند پدر و فرآیند فرزند در دو فضای مستقل و جدا از هم انجام می‌گردد. شکل زیر گویای مطلب است:

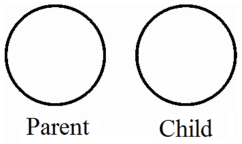


توجه: بنابراین رابطه $\text{card}(\text{Parent} \cap \text{Child}) = 0$ به معنی 0 صفحه مشترک، برای **فضای آدرس مستقل** برقرار است.

۲- اگر فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار نگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد استفاده قرار بگیرد و یک برنامه و قطعه کد جدید به درون خود بار کند، یعنی محتوای فرآیند فرزند بسته به شرایط، **در حد تغییر ساختار و تغییر مقدار متغیرها و حتی تغییر برنامه و کد تغییر کند**، آنگاه فضای آدرس فرآیند پدر و فرآیند فرزند شامل Stack ، Data ، Heap و Code به تبع اجرای دستور exec و ساخت محتوای کاملا جدید در فرآیند فرزند کاملا مستقل و

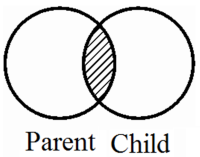
khalilifar.ir

جدا از هم خواهد بود و تغییرات در محتوای فرآیند پدر و فرآیند فرزند در دو فضای مستقل و جدا از هم انجام می‌گردد. شکل زیر گویای مطلب است:



توجه: بنابراین رابطه $card(Parent \cap Child) = 0$ به معنی 0 صفحه مشترک، برای **فضای آدرس مستقل** برقرار است.

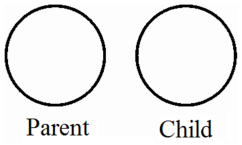
۳- اگر فضای آدرس فرآیند پدر و فرآیند فرزند در ابتدا **مشترک** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار بگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد استفاده قرار نگیرد، یعنی محتوای فرآیند پدر و فرآیند فرزند بسته به شرایط، **فقط در حد تغییر مقدار متغیرها و نه تغییر برنامه و کد تغییر کند،** آنگاه فضای آدرس فرآیند پدر و فرآیند فرزند شامل Stack ، Data ، Heap و Code در ابتدا به اشتراک گذاشته می‌شود و جهت صرفه‌جویی در مصرف حافظه، فرآیندهای پدر و فرزند در ابتدای کار از صفحات کد و داده به صورت مشترک استفاده می‌کنند، البته تا زمانی که فقط عمل خواندن (Read) بین فرآیندها مدنظر باشد، این صفحات به صورت مشترک استفاده می‌گردد. اما بر طبق تکنیک Copy-On-Write هرگاه یکی از دو فرآیند پدر یا فرزند بخواهد محتوای صفحه‌ای از صفحات مشترک داده و نه کد را تغییر دهد (مثلاً چیزی بنویسد) یک کپی جداگانه از آن صفحه برای آن فرآیند ساخته می‌شود و فرآیند از آن به بعد از آن صفحه استفاده می‌کند (به جای صفحه مشترک) و فرآیندهای دیگر از صفحات اصلی (مشترک) استفاده می‌کنند. بدین ترتیب محرمانگی داده‌ها حفظ شده و تغییرات صورت گرفته توسط یک فرآیند بر روی سایر فرآیندها اثر نخواهد داشت. در تکنیک Copy-On-Write فقط صفحاتی کپی می‌شوند که توسط فرآیندی تغییر یابند، و تمام صفحات بدون تغییر می‌تواند بین فرآیندهای پدر و فرزند بصورت مشترک استفاده شود. به عبارت دیگر پس از fork فرآیند پدر و فرآیند فرزند، مسیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرها بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. شکل زیر گویای مطلب است:



khalilifar.ir

توجه: بنابراین رابطه $card(Parent \cap Child) = k$ به معنی k صفحه مشترک، برای **فضای آدرس مشترک** برقرار است.

۴- اگر فضای آدرس فرآیند پدر و فرآیند فرزند در ابتدا **مشترک** باشد یعنی تکنیک Copy-On-Write مورد استفاده قرار بگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید برای فرآیند فرزند مورد استفاده قرار بگیرد و یک برنامه و قطعه کد جدید به درون خود بار کند، یعنی محتوای فرآیند فرزند بسته به شرایط، **در حد تغییر ساختار و تغییر مقدار متغیرها و حتی تغییر برنامه و کد تغییر کند،** آنگاه فضای آدرس فرآیند پدر و فرآیند فرزند شامل Stack ، Data ، Heap و Code به تبع اجرای دستور exec و ساخت محتوای کاملا جدید در فرآیند فرزند کاملا مستقل و جدا از هم خواهد بود و تغییرات در محتوای فرآیند پدر و فرزند در دو فضای مستقل و جدا از هم انجام می‌گردد. شکل زیر گویای مطلب است:

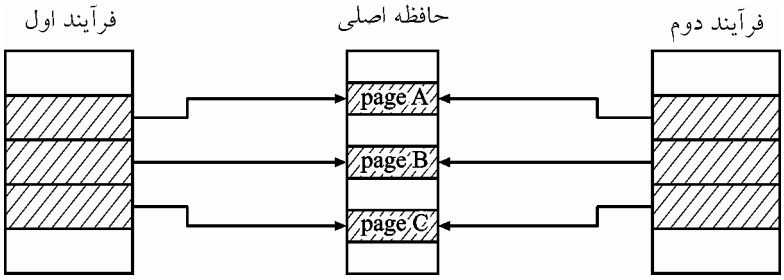


توجه: بنابراین رابطه $card(Parent \cap Child) = 0$ به معنی 0 صفحه مشترک، برای **فضای آدرس مستقل** برقرار است.

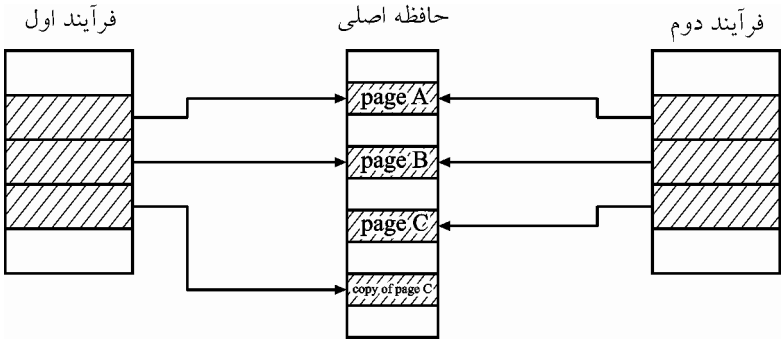
توجه: در سیستم عامل Unix یک فرآیند فرزند پس از ایجاد توسط فراخوانی fork، می‌تواند بلافاصله از فراخوانی سیستمی ()exec استفاده کند و کل فضای آدرس حافظه‌ی خود را جایگزین کند و از ادامه شراکت با پدر صرفه نظر کند. بنابراین، با این کار، حافظه‌ی جداگانه‌ای برای فرآیند فرزند ایجاد می‌شود. با توجه به این مطلب اگر بعد از اجرای fork و ایجاد فرآیند فرزند، ابتدا فرآیند پدر اجرا گردد، ممکن است فرآیند پدر بخواهد بطور خصوصی در یکی از صفحات چیزی بنویسد و باعث ایجاد یک کپی از صفحه بر اساس تکنیک Copy-On-Write شود. حال اگر در ادامه فرآیند فرزند اجرا گردد و در همان ابتدا از فراخوان سیستمی ()exec استفاده کند و راه خود را از پدر جدا کند و شراکت را برهم زند، صفحاتی که پدر به دلیل تغییرات خود ایجاد کرده بود، سربار به حساب می‌آیند و کار بیهوده تلقی می‌گردد، مانند پدری که پس از فرزنددار شدن برای صرفه‌جویی در هزینه‌ها، از خانه‌ی خود به شکل اشتراکی استفاده می‌کند. اما این پدر بعدها به دلیل کارهای شخصی خود خانه‌ی دیگری را نیز تهیه می‌کند و بعد از تهیه خانه‌ی دوم متوجه می‌شود، که فرزند راه خود را جدا کرده است، و شراکت را بر هم زده است، و فرزند نیز خانه‌ای برای خود تهیه کرده است، حال خانه‌ی دوم پدر برای رسیدگی به امور شخصی بلااستفاده می‌ماند و این سربار است، زیرا دیگر شراکتی در کار نیست، فرزندی نیست، همان خانه‌ی اول برای پدر کافی بود. بهتر بود پدر صبر می‌کرد، تا اول فرزند تصمیم بگیرد، سپس بر اساس تصمیم فرزند، پدر نیز تصمیم خود را می‌گرفت.

khalilifar.ir

نتیجه: حال مجدداً برگردید به وادی کامپیوتر، در صورتی که اگر بعد از اجرای فراخوانی سیستمی fork، ابتدا فرآیند فرزند فراخوانی و اجرا شود، ممکن است در همان ابتدای اجرایش دستور ()exec را اجرا کند و در نتیجه از آن به بعد، از فضای حافظه‌ی شخصی خود استفاده کند، که در این صورت اگر فرآیند پدر بخواهد چیزی بر روی صفحات مشترک شده بنویسد، دیگر نیازی به کپی‌کردن آن صفحه نخواهد بود و این یعنی حذف سربار و افزایش کارایی.
شکل زیر گویای مطالب می‌باشد:



قبل از اینکه فرآیند اول صفحه C را تغییر دهد.



بعد از اینکه فرآیند اول صفحه C را تغییر دهد.

توجه: در این تکنیک زمانی فرآیند پدر می‌تواند خاتمه یابد که یک کپی از صفحات آن برای هر یک از فرزندانش ایجاد شده باشد (یعنی صفحات تمامی فرآیندهای فرزند تغییر کرده باشند، به عبارت دیگر همه فرزندان همه صفحه‌های مربوط به خود را تغییر داده باشند.) و دیگر نیازی به صفحات فرآیند پدر نباشد.

توجه: همانطور که گفتیم، سیستم عامل Unix و Linux برای ایجاد یک فرآیند فرزند (جدید) از فراخوان سیستمی fork استفاده می‌کند، بنابراین fork برای ایجاد یک فرآیند فرزند مورد استفاده قرار می‌گیرد، هدف fork ایجاد یک فرآیند فرزند برای فرآیند فراخوانی کننده آن یعنی فرآیند پدر است، به عبارت دیگر fork برای یک پدر، به شکل طبیعی یک فرزند به دنیا می‌آورد، fork

khalilifar.ir

متخصص زایمان است. فراخوان سیستمی fork هیچ آرگومان ورودی ندارد، اما مقدار بازگشتی دارد، fork در حالت اجرای موفق دو مقدار برمی‌گرداند که یکی برای فرآیند فرزند برابر مقدار صفر که به آن پاس داده می‌شود و یکی دیگر هم برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است، وقتی فرزندی به دنیا می‌آید، علاوه بر آینکه خودش صاحب pid می‌شود، pid آنرا تحویل پدرش هم می‌دهند. مانند وقتی که فرزندی به دنیا می‌آید، علاوه بر اینکه خودش صاحب شماره شناسنامه می‌شود، شماره شناسنامه آنرا تحویل پدرش هم می‌دهند. همچنین عدد صفر پاس داده شده به فرآیند فرزند هم به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد. نوع مقدار بازگشتی fork از جنس pid_t است که در کتابخانه sys/types.h زبان C و C++ تعریف شده است، البته به طور معمول در سایر زبان‌ها از نوع integer است. همچنین یک فرآیند پدر یا فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع ()getpid استفاده نماید.

توجه: با استفاده از مقدار حاصل از بازگشت اجرای تابع fork، می‌توان متوجه شد که در ادامه و پس از به دنیا آمدن فرزند نو رسیده، فرآیند پدر در چه مسیری اجرا شود و فرآیند فرزند در چه مسیری اجرا شود.

توجه: فراخوانی fork در حالت عدم اجرای موفق یک عدد صحیح کوچکتر از صفر برمی‌گرداند.

توجه: مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود.

توجه: اینکه فرآیند پدر یا فرآیند فرزند به چه ترتیبی اجرا شوند، بستگی به زمان‌بند پردازنده و شرایط درون خود فرآیندها دارد. بسته به شرایط ممکن است اول فرآیند پدر اجرا شود و بعد فرآیند فرزند و یا اول فرآیند فرزند اجرا شود و بعد فرآیند پدر و یا هر دو باهم به طور همروند در سیستم تک پردازنده‌ای یا موازی در سیستم چند پردازنده‌ای اجرا شوند.

توجه: فرآیند فرزند یک شماره pid یکتا دارد که با فرآیند پدر متفاوت است. همچنین فرآیند پدر نیز یک شماره pid یکتا دارد که با فرآیند فرزند متفاوت است.

توجه: ppid نشان دهنده pid فرآیند پدر است. بنابراین شماره ppid فرآیند فرزند، برابر pid فرآیند پدر است.

توجه: هر برنامه هنگامی که در حافظه قرار می‌گیرد تا اجرا شود، حاوی سه قسمت اصلی کد (CS: Code Segment)، داده (DS: Data Segment) و پشته (SS: Stack Segment) است. مابقی حافظه که در اختیار برنامه نبوده و آزاد می‌باشد به حافظ Heap یا حافظه پویا اختصاص داده می‌شود. دستورالعمل‌های برنامه در قسمت کد، متغیرهای سراسری در قسمت داده و متغیرهای محلی در قسمت پشته ساخته می‌شوند. متغیرهای سراسری ابتدای برنامه و بیرون همه توابع ساخته

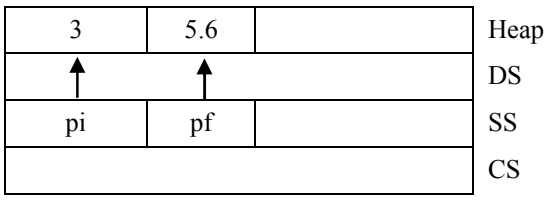
khalilifar.ir

شده و تا انتهای برنامه فضای آنها حفظ می‌گردد. متغیرهای محلی بـه محـض ورود بـه زیربرنامـه ساخته شده و هنگام اتمام زیربرنامه از بین می‌روند. به کمک مفهـوم اشـاره‌گرهـا و حافظـه Heap می‌توان متغیرهایی پویا در حافظه Heap پدید آورد (حداکثر به اندازه حافظه Heap) و همچنین هر وقت که دیگر نیازی به آنها نباشد، می‌توان آنها را از بین برده و فضای آنها را بـه Heap برگردانـد. در زبان C برای گرفتن فضا در حافظـه Heap از تـابع malloc و بـرای رهاسـازی آن از تـابع free استفاده می‌شود. معرفی این توابع در فایل stdlib.h قرار دارد. فرم کلی این توابـع بـه صـورت زیـر است:

```
void * malloc (اندازه فضای متغیر بر حسب بایت) ;
void free (void *p) ;
```

مثال: قطعه کد زیر فرم استفاده از دستورات malloc و free و نحوه تعریف متغیرهای پویا را نشان می‌دهد:

```
#include  <stdio.h> /* printf */
#include  <stdlib.h> /* malloc , free */
int main(void)
{
    int *pi;
    float *pf;
    pi = (int *) malloc (sizeof (int));
   *pi = 3;
    pf = (float *) malloc (sizeof (float));
   *pf = 5.6;
    printf("%f",*pi +*pf); /* 8.6 */
   free(pi);
   free(pf);
    return 0;
}
```



khalilifar.ir

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که یک متغیر را که مقدارش برابر با 5 است را در خروجی نمایش می‌دهد:

```
//gcc 5.4.0
#include  <stdio.h> /* printf */
int main(void)
{
    int i = 5;
    printf("i=%d",i);
    return 0;
}
```

توجه: این برنامه در سیستم عامل UNIX و Linux به زبان C تحت کامپایلر gcc نوشته شده است.

خروجی نهایی برنامه به صورت زیر است:

```
i = 5
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که یک متغیر را که مقدارش برابر با 5 است را پس از اجرای دستور fork در خروجی نمایش می‌دهد:

```
//gcc 5.4.0
#include <stdio.h>   /*printf */
#include <unistd.h>   /*fork */
int main ()
{
    int i = 5;
    printf("Hello!");

   /*fork a child process*/
   fork();
    printf("i=%d",i);
    return 0;
}
```

توجه: این برنامه در سیستم عامل UNIX و Linux به زبان C تحت کامپایلر gcc نوشته شده است.

توجه: در سیستم عامل ویندوز ایجاد فرآیند توسط دستور ()fork انجام نمی‌شود، در ویندوز ایجاد فرآیند توسط create process routines انجام می‌شود.

توجه: برنامه اجرا می‌شود و در اولین خط مقدار متغیر محلی i برابر با 5 می‌شود. دقت کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می‌شود که داخل Stack Segment تعریف و مقداردهی می‌شود. در خط بعد کلمه‌ی !Hello توسط دستور printf در

khalilifar.ir

خروجی نمایش داده می‌شود، تا اینجا همه چیز عادی و طبق روال معمول است. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. همانطور که گفتیم محتوای فرآیند فرزند در ابتدا از نظر داده‌ها، مقدارها و کد شامل Stack ، Data ، Heap ، Register ، PCB و Code، یک کپی **کاملا**، **دقیقا** و **یکسان** از فرآیند پدر است، چون داده و مقداردهی اولیه از فرآیند پدر به فرآیند فرزند کپی و پاس داده می‌شود. حتی مقادیر PCB فرآیند پدر در PCB فرآیند فرزند کپی می‌شود و تنها چیزی که در محتوای PCB فرآیند پدر و PCB فرآیند فرزند تفاوت دارد، مقدار pid است چون هر فرآیند pid مختص به خودش را دارد، در واقع pid فرآیند پدر با pid فرآیند فرزند متفاوت است. همچنین دقت کنید که فضای آدرس PCB فرآیند پدر از فضای آدرس PCB فرآیند فرزند مستقل است. **در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دستور fork را اجرا می کنند، زیرا فیلد شمارنده برنامه PC موجود در PCB فرآیند پدر و فرزند که پس از دستور fork مقادیر یکسانی نیز دارند به دستور پس از fork اشاره می‌کند.** از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هستند، مقدار متغیر i در هر دو فرآیند برابر مقدار 5 است. دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می‌شود دستور printf به معنای نمایش مقدار متغیر i در خروجی است، هر کدام از فرآیندهای پدر و فرزند پس از رسیدن به دستور printf مقدار متغیر i را به طور **مستقل** در خروجی نمایش می‌دهند. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، مسیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرهایی که مقداردهی اولیه شده‌اند بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضای آدرس مختص به خود را دارند، هرگونه تغییر، مستقل از سایرین خواهد بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسان و برابر i و برابر مقدار 5 است، اما فضای آدرس فرآیند پدر و فرزند متفاوت است.

خروجی نهایی برنامه به صورت زیر است:

khalilifar.ir

Parent :
```
#include< stdio.h >
#include  < unistd.h >
int main ()
{
  int i  =  5;
  printf ("Hello!");
   / * fork a child process * /
  fork();
 → printf ("i = %d",i);
  return  0;
}
```

Child :
```
#include< stdio.h >
#include  < unistd.h >
int main ()
{
  int i  =  5;
  printf ("Hello!");
   / * fork a child process * /
  fork();
  → printf ("i = %d",i);
  return  0;
}
```

```
Hello!
i = 5
i = 5
```

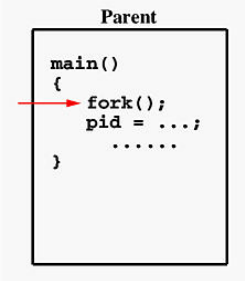
توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال مقدماتی فوق مسیر فرآیند پدر و فرزند به دلیل نبود شرط if از هم متمایز نشده بودند.

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که مقدار pid را توسط دستور printf داخل یک حلقه‌ی for پس از اجرای دستور fork در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include <stdio.h>   /* printf  */
#include <unistd.h>   /* fork */
#include  <sys/types.h>   /* pid_t */
int  main()
{
    pid_t  pid;
    int    i = 0;
    fork();
    pid = getpid() ;
    for (i = 1 ; i < 4 ; i++) {
            printf("pid=%d \n" , pid);
      }
    return 0;
}
```

khalilifar.ir

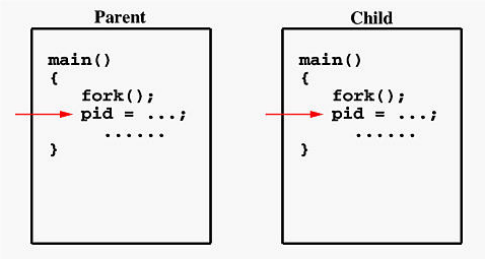
توجه: برنامه اجرا می‌شود و در اولین خط مقدار متغیر محلی i برابر با 0 می‌شود. دقت کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می‌شود که داخل Stack Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود.



توجه: در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دستور fork را اجرا می کنند. از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هستند، مقدار اولیه متغیر i در هر دو فرآیند برابر مقدار 0 است. دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می‌شود دستور pid = getpid()، یک فرآیند پدر یا فرزند می‌تواند جهت بازیابی مقدار process ID یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید. هر کدام از فرآیندهای پدر و فرزند پس از رسیدن به دستور pid = getpid () مقدار متغیر pid را به طور مستقل بر اساس تابع () getpid مقداردهی می‌کنند. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، مسیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرهایی که مقداردهی اولیه شده‌اند بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضای آدرس مختص به خود را دارند، هرگونه تغییر، مستقل از سایرین خواهد بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسان و برابر i و برابر مقدار 0 است، اما فضای آدرس فرآیند پدر و فرزند متفاوت است.

دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می‌شود دستور printf داخل یک حلقه for است که 3 بار تکرار می‌شود. هر کدام از فرآیندهای پدر و فرزند پس از رسیدن به دستور printf داخل حلقه for مقدار متغیر pid را به طور مستقل بر اساس تابع () getpid مقداردهی می‌کنند و در خروجی نمایش می‌دهند.

khalilifar.ir



خروجی نهایی برنامه به صورت زیر است:

Parent :

```
#include < stdio.h >
#include < unistd.h >
#include< sys / types.h >
int main()
{
  pid _ t pid;
  int  i = 0;
  fork();
  → pid  =  getpid();
  for (i  =  1 ; i  <  4 ; i++){
        printf ("pid = %d  \n", pid);
    }
  return 0;
}
```

Child :

```
#include < stdio.h >
#include < unistd.h >
#include< sys / types.h >
int main()
{
  pid _ t pid;
  int  i = 0;
  fork();
  → pid  =  getpid();
  for (i  =  1 ; i  <  4 ; i++){
        printf ("pid = %d  \n", pid);
    }
  return 0;
}
```

```
pid = 2600
pid = 2600
pid = 2600
pid = 2603
pid = 2603
pid = 2603
```

توجه: اینکه فرآیند پدر یا فرآیند فرزند به چه ترتیبی اجرا شوند، بستگی به زمان‌بند پردازنده و شرایط درون خود فرآیندها دارد. بسته به شرایط، ممکن است اول فرآیند پدر اجرا شود و بعد فرآیند فرزند و یا اول فرآیند فرزند اجرا شود و بعد فرآیند پدر و یا هر دو باهم به طور همروند در سیستم تک پردازنده‌ای یا موازی در سیستم چند پردازنده‌ای اجرا شوند. برای مثال یک فرم دیگر خروجی بر اساس زمان‌بندی همروند پردازنده در سیستم تک پردازنده‌ای یا زمان‌بندی موازی پردازنده در سیستم چند پردازنده‌ای می‌تواند به صورت زیر باشد:

khalilifar.ir

```
pid = 2600
pid = 2603
pid = 2600
pid = 2603
pid = 2600
pid = 2603
```

توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال مقدماتی فوق مسیر فرآیند پدر و فرزند به دلیل نبود شرط if از هم متمایز نشده بود.

توجه: این مدل استفاده از fork کاربرد خاصی ندارد. وقتی fork و ایجاد فرآیند فرزند(جدید)، کارآمد است که بتوان بعد از اجرای fork دو محتوای متفاوت از فرآیند پدر و فرآیند فرزند را اجرا کرد و نه این که دقیقا همان کد قبلی را اجرا کرد. برای اینکه بتوان بعد از اجرای fork دو محتوای متفاوت را اجرا کرد یک راه بیشتر نداریم و آن هم استفاده از خروجی و مقدار بازگشتی دستور fork است. دستور fork در فرآیند پدر و فرآیند فرزند دو خروجی متفاوت ایجاد می‌کند. هر فرآیند در سیستم عامل یک شماره ی مختص به خود دارد که سیستم عامل برای شناسایی و کار با فرآیندها از آن استفاده می‌کند که به آن process id یا pid گفته می‌شود. در فرآیند پدر، خروجی و مقدار بازگشتی fork شناسه‌ی pid فرآیند فرزند است، در حالی که خروجی و مقدار بازگشتی fork در فرآیند فرزند برابر با صفر است. به این ترتیب با استفاده از تفاوت خروجی و مقدار بازگشتی fork در فرآیند پدر و فرآیند فرزند، می‌توان کاری کرد که فرآیند پدر و فرآیند فرزند بعد از دستور fork کارهای متفاوتی انجام دهند، که در ادامه، مثالی از این مورد را بررسی می‌کنیم.

توجه: دقت کنید که اگر خروجی fork یک عدد منفی بود، بدین معنی است که برنامه موفق به ایجاد یک فرآیند فرزند(جدید) نشده است.

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که مقدار i، مقدار pid حاصل از بازگشت fork و مقدار Process id حاصل از بازگشت getpid را در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include <stdio.h>   /* printf */
#include <unistd.h>  /*fork */
#include <sys/types.h>  /* pid_t */
int main()
{
    pid_t pid;
    int i = 0;
```

khalilifar.ir

```
    pid = fork (); /*fork a child process*/

    if (pid > 0) { /*Parent Process:*/
     /*When fork() returns a positive number, we are in the parent process*/
     /*the fork return value is the PID of the newly created child process*/

       printf ("*** Parent Process Begin *** \n");

       i = i + 1 ;
       printf ("i = %d \n" , i);
       printf ("Process id = %d \n", getpid () );
       printf ("pid = %d \n" , pid);

       printf ("*** Parent Process End *** \n");
  }
    else if (pid == 0) { /*Child Process:*/
     /*When fork() returns 0, we are in the child process.*/

       printf ("*** Child Process Begin *** \n");

       i = i - 1 ;
       printf ("i = %d \n" , i);
       printf ("Process id = %d \n", getpid () );
       printf ("pid = %d \n" , pid);

       printf ("*** Child Process End *** \n");
    }


    else { /*error occurred*/
    /*When fork() returns a negative number, an error happened*/
     printf ("fork creation failed!!! \n ");
    }


  return 0;
}
```

توجه: برنامه اجرا می‌شود و در اولین خط مقدار متغیر محلی i برابر با 0 می‌شود. دقت کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می‌شود که داخل

khalilifar.ir

Stack Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دستور fork را اجرا می‌کنند. از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هستند، مقدار اولیه متغیر i در هر دو فرآیند برابر مقدار 0 است. دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می‌شود دستور if و else if است. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، مسیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرهایی که مقداردهی اولیه شده‌اند بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضای آدرس مختص به خود را دارند، هرگونه تغییر، مستقل از سایرین خواهد بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسان و برابر i و برابر مقدار 0 است، اما فضای آدرس فرآیند پدر و فرزند متفاوت است.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
i = i + 1 ;
printf ("i = %d \n" , i);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور i = i + 1 برابر 1 می‌شود و این تغییر روی فرآیند فرزند اثری ندارد.

i = i + 1; **printf ("i = %d \n" , i);**	**i = 1**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

khalilifar.ir

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 2603**

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
i = i - 1 ;
printf ("i = %d \n" , i);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور i = i - 1 برابر 1- می‌شود و این تغییر روی فرآیند پدر اثری ندارد.

i = i - 1; **printf ("i = %d \n" , i);**	**i = -1**

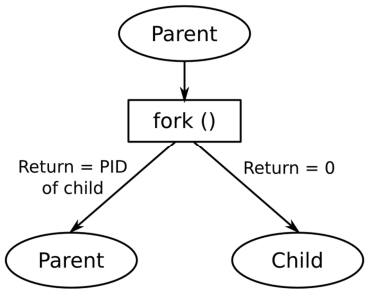
توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2603**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

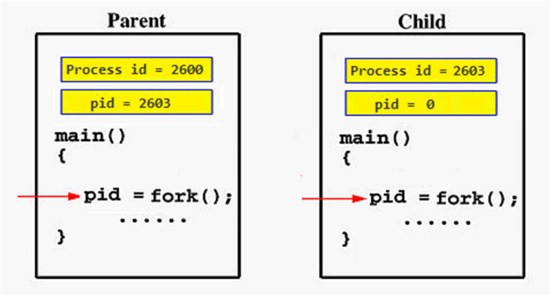
pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 0**

شکل زیر گویای مطلب است:



شکل زیر گویای مطلب است:

khalilifar.ir



خروجی نهایی برنامه به صورت زیر است:

```
*** Parent Process Begin ***
i = 1
Process id = 2600
pid = 2603
*** Parent Process End ***
*** Child Process Begin ***
i = -1
Process id = 2603
pid = 0
*** Child Process End ***
```

توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال فوق مسیر فرآیند پدر و فرزند به دلیل وجود شرط if از هم متمایز شده بود.

توجه: هر برنامه هنگامی که در حافظه قرار می‌گیرد تا اجرا شود، حاوی سه قسمت اصلی کد (CS: Code Segment)، داده (DS: Data Segment) و پشته (SS: Stack Segment) است. مابقی حافظه که در اختیار برنامه نبوده و آزاد می‌باشد به حافظ Heap یا حافظه پویا اختصاص داده می‌شود. دستورالعمل‌های برنامه در قسمت کد، متغیرهای سراسری در قسمت داده و متغیرهای محلی در قسمت پشته ساخته می‌شوند. متغیرهای سراسری ابتدای برنامه و بیرون همه توابع ساخته شده و تا انتهای برنامه فضای آنها حفظ می‌گردد. متغیرهای محلی به محض ورود به زیربرنامه ساخته شده و هنگام اتمام زیربرنامه از بین می‌روند. به کمک مفهوم اشاره‌گرها و حافظه Heap می‌توان متغیرهایی پویا در حافظه Heap پدید آورد (حداکثر به اندازه حافظه Heap) و همچنین هر وقت که دیگر نیازی به آنها نباشد، می‌توان آنها را از بین برده و فضای آنها را به Heap برگرداند. در زبان C برای گرفتن فضا در حافظه Heap از تابع malloc و برای رهاسازی آن از تابع free استفاده می‌شود. معرفی این توابع در فایل stdlib.h قرار دارد. فرم کلی این توابع به صورت زیر است:

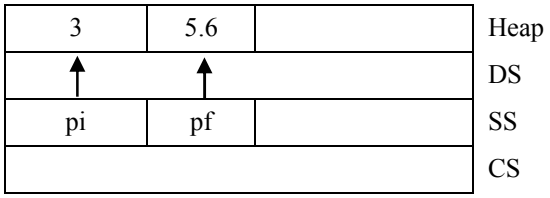
void * malloc (اندازه فضای متغیر بر حسب بایت) ;

khalilifar.ir

void free (void *p) ;

مثال: برنامه زیر دستورات و نحوه تعریف متغیرهای پویا را نشان می‌دهد:

```
#include  <stdio.h> /* printf */
#include  <stdlib.h> /* malloc , free */
int main(void)
{
    int *pi;
    float *pf;
    pi = (int *) malloc (sizeof (int));
   *pi = 3;
    pf = (float *) malloc (sizeof (float));
   *pf = 5.6;
    printf("%f",*pi +*pf); /* 8.6 */
   free(pi);
   free(pf);
    return 0;
}
```



توجه: pi* می‌گوید در آدرسی که توسط pi مشخص شده است مقدار 3 را قرار دهد یا بخواند و همچنین pf* می‌گوید در آدرسی که توسط pf مشخص شده است مقدار 5.6 را قرار دهد یا بخواند. به تفاوت pi و pi* دقت نمایید. در pi توسط دستور malloc آدرس یک متغیر پویا رزرو می‌شود، اما توسط pi* آن آدرس موجود در pi مقداردهی می‌شود و یا مقدار آن خوانده می‌شود.

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که مقدار pi* تعریف شده در حافظه Heap، مقدار pid حاصل از بازگشت fork و مقدار Process id حاصل از بازگشت getpid را در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0

#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork */
#include  <sys/types.h>  /* pid_t */
```

khalilifar.ir

```
int  main()
{
    pid_t  pid;
    int *pi;
    pi = (int *) malloc (sizeof (int));
   *pi=0;

       pid = fork (); /*fork a child process*/

       if (pid > 0) { /*Parent Process:*/
        /*When fork() returns a positive number, we are in the parent process*/
        /*the fork return value is the PID of the newly created child process*/

          printf ("*** Parent Process Begin *** \n");

          *pi = *pi + 1 ;
          printf ("*pi = %d \n" , *pi);
          printf ("Process id = %d \n", getpid () );
          printf ("pid = %d \n" , pid);

          printf ("*** Parent Process End *** \n");
     }
       else if (pid == 0) { /*Child Process:*/
        /*When fork() returns 0, we are in the child process.*/

          printf ("*** Child Process Begin *** \n");

          *pi = *pi - 1 ;
          printf ("*pi = %d \n" , *pi);
          printf ("Process id = %d \n", getpid () );
          printf ("pid = %d \n" , pid);

          printf ("*** Child Process End *** \n");
       }

       else { /*error occurred*/
       /*When fork() returns a negative number, an error happened*/
        printf ("fork creation failed!!! \n ");
       }

   return 0;
}
```

توجه: برنامه اجرا می‌شود و در خط pi=0* مقدار متغیر پویا در آدرس pi تعریف شده در حافظه Heap برابر با 0 می‌شود. دقت کنید که متغیر پویا در آدرس pi داخل تابع main تعریف شده است و یک متغیر پویا محسوب می‌شود که داخل Heap تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی

khalilifar.ir

فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند. از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هستند، مقدار اولیه متغیر پویا pi* در هر دو فرآیند برابر مقدار 0 است. دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می‌شود دستور if و else if است. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، مسیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرهایی که مقداردهی اولیه شده‌اند بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضای آدرس مختص به خود را دارند، هرگونه تغییر، مستقل از سایرین خواهد بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسان و برابر pi* و برابر مقدار 0 است، اما فضای آدرس فرآیند پدر و فرزند متفاوت است.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
*pi = *pi + 1 ;
printf ("*pi = %d \n" , *pi);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر pi* برابر 0 است و پس از اجرای دستور *pi = *pi + 1 برابر 1 می‌شود و این تغییر روی فرآیند فرزند اثری ندارد.

***pi = *pi + 1;** **printf ("*pi = %d \n" , *pi);**	***pi = 1**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

khalilifar.ir

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 2603**

دستورات بعدی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
*pi = *pi - 1 ;
printf ("*pi = %d \n" , *pi);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر pi* برابر 0 است و پس از اجرای دستور *pi = *pi - 1 برابر 1- می‌شود و این تغییر روی فرآیند پدر اثری ندارد.

***pi = *pi - 1;** **printf ("*pi = %d \n" , *pi);**	***pi = -1**

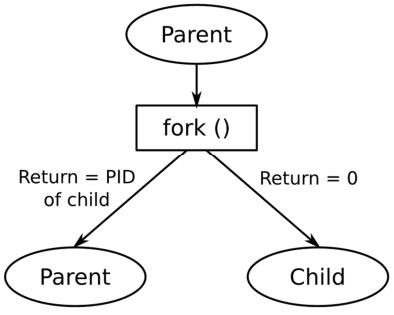
توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2603**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

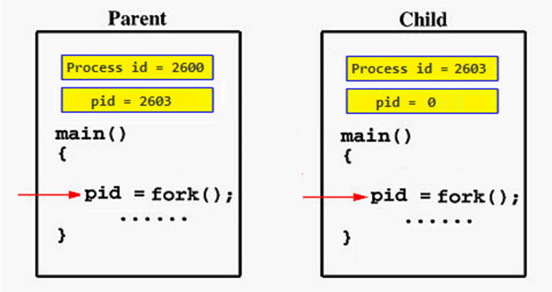
pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 0**

شکل زیر گویای مطلب است:



شکل زیر گویای مطلب است:

khalilifar.ir



خروجی نهایی برنامه به صورت زیر است:

```
*** Parent Process Begin ***
*pi = 1
Process id = 2600
pid = 2603
*** Parent Process End ***
*** Child Process Begin ***
*pi = -1
Process id = 2603
pid = 0
*** Child Process End ***
```

توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال فوق مسیر فرآیند پدر و فرزند به دلیل وجود شرط if از هم متمایز شده بود.

مثال: وقتی یک فرآیند فرزند توسط فرآیند پدر و دستور ()fork ایجاد می‌شود، کدام بخش‌های زیر مابین فرآیند پدر و فرزند به اشتراک گذاشته می‌شود؟

الف) Stack

ب) Heap

ج) Shared memory segments

پاسخ: فقط Shared memory segments مابین فرآیند پدر و فرزند به اشتراک گذاشته می‌شود، اما محتوای داده‌های Stack و Heap فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرهایی که مقداردهی اولیه شده‌اند بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است.

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، مقدار value در خطوط LINE A و LINE B برابر کدام گزینه است؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

khalilifar.ir

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/

int value = 20;

int  main()
{
    pid_t  pid , pidChild;

        pid = fork (); /*fork a child process*/

        if (pid > 0) { /*Parent Process*/

          printf ("*** Parent Process Begin *** \n");

            pidChild = wait (NULL);
            printf ("***Child Complete*** \n");
            value = value + 15 ;
            printf ("Parent: value = %d \n",value); /*LINE A*/
            printf ("Child Process id wait = %d \n", pidChild);
            printf ("Parent Process id = %d \n", getpid () );

           printf ("*** Parent Process End *** \n");

       }
          else if (pid == 0) { /*Child Process:*/

         printf ("***Child Process Begin *** \n");

             value = value - 15 ;
             printf ("Child: value = %d \n",value); /*LINE B*/
             printf ("Child Process id = %d \n", getpid () );

       printf ("***Child Process End *** \n");

       }


       else { /*error occurred*/
         printf ("fork creation failed!!! \n ");
       }


   return 0;
}
```

۱) LINE A = 5 , LINE B = 35　　۲) LINE A = 35 , LINE B = 5

۳) LINE A = 5 , LINE B = 0　　۴) LINE A = 0 , LINE B = 5

پاسخ- گزینه (۲) صحیح است.

khalilifar.ir

توجه: برنامه اجرا می‌شود و در اولین خط مقدار متغیر سراسری value برابر با 20 می‌شود. دقت کنید که متغیر value بالای تابع main تعریف شده است و یک متغیر سراسری محسوب می‌شود که داخل Data Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
pidChild = wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند.

توجه: فراخوان سیستمی ()wait مقدار Process id فرآیند فرزندی که پایان یافته است را در خروجی بر می‌گرداند.

توجه: فرآیند فرزند از طریق فراخوان سیستمی ()wait می‌تواند با فرآیند پدرش ارتباط برقرار کند، به عبارت دیگر مقدار Process id فرآیند فرزند توسط فراخوان سیستمی ()wait به فرآیند پدرش پاس داده می‌شود.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
value = value - 15 ;
printf ("Child: value = %d \n",value); /*LINE B*/
printf ("Child Process id = %d \n", getpid ());
```

توجه: مقدار اولیه متغیر سراسری value برابر 20 است و پس از اجرای دستور value=value-15 برابر 5 می‌شود و این تغییر روی فرآیند پدر اثری ندارد. هر چند که متغییر value سراسری است.

value = value - 15; **printf ("Child: value = %d \n",value); /*LINE B*/**	**value = 5**

توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Child Process id = %d \n", getpid ());	**Process id = 2603**

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
pidChild = wait (NULL);
printf ("***Child Complete*** \n");
value = value + 15 ;
printf ("Parent: value = %d \n",value); /*LINE A*/
```

khalilifar.ir

```
printf ("Child Process id wait = %d \n", pidChild);
printf ("Parent Process id = %d \n", getpid () );
```

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

توجه: دقت کنید که عمل انتساب pidChild = wait(NULL) باقی‌مانده از قبل ابتدا تکمیل می‌شود و سپس دستور زیر اجرا می‌شود.

printf ("*Child Complete \n***");**	*****Child Complete*****

توجه: مقدار اولیه متغیر سراسری value برابر 20 است و پس از اجرای دستور value=value+15 برابر 35 می‌شود و این تغییر روی فرآیند فرزند اثری ندارد. هر چند که متغییر value سراسری است. و هرچند که قبلا مقدار متغیر سراسری value توسط فرآیند فرزند برابر 5 شده است.

value = value + 15; printf ("Parent: value = %d \n",value); /*LINE A*/	**value = 35**

توجه: فراخوان سیستمی ()wait مقدار Process id فرآیند فرزندی که پایان یافته است را در خروجی بر می‌گرداند.

printf ("Child Process id wait = %d \n", pidChild);	**Child Process id wait = 2603**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Child Process id = %d \n", getpid ());	**Process id = 2600**

خروجی نهایی برنامه به صورت زیر است:

```
***Child Process Begin ***
Child: value = 5
Child Process id = 2603
***Child Process End ***
*** Parent Process Begin ***
***Child Complete***
Parent: value = 35
Child Process id wait = 2603
Parent Process id = 2600
*** Parent Process End ***
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، مقدار pid و pid1 در خطوط LINE A ، LINE B ، LINEC و LINE D برابر کدام گزینه است؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

khalilifar.ir

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/

int  main()
{
    pid_t  pid , pid1;

        pid = fork (); /*fork a child process*/

        if (pid > 0) { /*Parent Process*/

            pid1 = getpid ();
            printf ("Parent : pid = %d \n" , pid); /*LINE C*/
            printf ("Parent : pid1 = %d \n" , pid1); /*LINE D*/
       }

        else if (pid == 0) { /*Child Process*/
            pid1 = getpid ();
            printf ("Child : pid = %d  \n" , pid); /*LINE A*/
            printf ("Child : pid1 = %d  \n" , pid1); /*LINE B*/
      }
        else { /*error occurred*/
         printf ("fork creation failed!!! \n ");
{
   return 0;
}
```

۱) LINE A = 2603 , LINE B = 0 , LINE C = 2603 , LINE D = 2600

۲) LINE A = 0 , LINE B = 2603 , LINE C = 2600 , LINE D = 2603

۳) LINE A = 2600 , LINE B = 2603 , LINE C = 2603 , LINE D = 0

۴) LINE A = 0 , LINE B = 2603 , LINE C = 2603 , LINE D = 2600

پاسخ- گزینه (۴) صحیح است.

توجه: در اولین خط، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
pid1 = getpid ();
printf ("Parent : pid = %d \n" , pid); /*LINE C*/
printf ("Parent : pid1 = %d \n" , pid1); /*LINE D*/
```

khalilifar.ir

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

printf ("Parent : pid = %d \n" , pid); /*LINE C*/	**pid = 2603**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

pid1 = getpid (); **printf ("Parent : pid1 = %d \n" , pid1); /*LINE D*/**	**Parent : pid1 = 2600**

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
pid1 = getpid ();
printf ("Child : pid = %d \n" , pid); /*LINE A*/
printf ("Child : pid1 = %d \n" , pid1); /*LINE B*/
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

printf ("Child : pid = %d \n" , pid); /*LINE A*/	**pid = 0**

توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

pid1 = getpid (); **printf ("Child : pid1 = %d \n" , pid1); /*LINE B*/**	**Child : pid1 = 2603**

خروجی نهایی برنامه به صورت زیر است:

```
Parent: pid = 2603
Parent: pid1 = 2600
Child: pid = 0
Child: pid1 = 2603
```

مثال: با اجرای قطعه کد زیر در نهایت چند فرآیند خواهیم داشت؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600، pid واقعی فرآیند فرزند اول برابر مقدار 2601، pid واقعی فرآیند فرزند دوم برابر مقدار 2602 و pid واقعی فرآیند فرزند سوم برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
```

khalilifar.ir

```
#include <sys/types.h> /*pid_t*/
int main()
{
    if (fork())
        if(fork())
fork();
   printf ("Process id = %d \n", getpid ());
   return 0;
}
```

۱) 0 ۲) 2 ۳) 4 ۴) 8

پاسخ- گزینه (۳) صحیح است.

توجه: در اولین خط، دستور if (fork()) قرار دارد، وقتی که دستور fork() داخل دستور if اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (2601)
if (fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است.

if (2603)	TRUE

توجه: در ابتدا با فراخوانی دستور fork() یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی if (2601) برابر مقدار TRUE خواهد بود. که منجر به این می‌شود که دستورات بعدی فرآیند پدر مورد بررسی قرار گیرد.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
if (0)
if (fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود.

khalilifar.ir

if (0)	FALSE

توجه: از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی if (0) برابر مقدار FALSE خواهد بود. که منجر به این می‌شود که دستورات fork دوم و fork سوم مورد بررسی قرار نگیرد. اما از آنجا که دستور printf مستقل و خارج از بدنه دستورات شرطی است، در انتهای فرآیند فرزند اول اجرا می‌شود.

توجه: دقت داشته باشید که if (fork()) دوم داخل بدنه if (fork()) اول است و fork() سوم داخل بدنه if (fork()) دوم است. بنابراین اگر if (fork()) اول برقرار نباشد، if (fork()) دوم و fork() سوم به تبع آن اجرا نخواهد شد.

توجه: یک فرآیند فرزند **(C1)** می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

توجه: دقت داشته باشید که یکی از بهترین شیوه‌های شمارش تعداد کل فرآیندها، قرار دادن یک دستور printf جهت نمایش Process id توسط دستور getpid() در انتهای قطعه کد پایه است.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (2602)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

if (2604)	TRUE

توجه: در ابتدا با فراخوانی دستور fork() یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی if (2602) برابر مقدار TRUE خواهد بود. که منجر به این می‌شود که دستورات بعدی فرآیند پدر مورد بررسی قرار گیرد.

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
if (0)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

khalilifar.ir

if (0)	FALSE

توجه: از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی if (0) برابر مقدار FALSE خواهد بود. که منجر به این می‌شود که دستور fork سوم مورد بررسی قرار نگیرد. اما از آنجا که دستور printf مستقل و خارج از بدنه دستورات شرطی است، در انتهای فرآیند فرزند دوم اجرا می‌شود.

توجه: دقت داشته باشید که ()fork سوم داخل بدنه if (fork()) دوم است. بنابراین اگر if (fork()) دوم برقرار نباشد، ()fork سوم به تبع آن اجرا نخواهد شد.

توجه: یک فرآیند فرزند (C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2602

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2603=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: وقتی که دستور()fork اجرا می‌شود یک فرآیند فرزند سوم (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند سوم است.

توجه: یک فرآیند پدر (P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2600

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

```
0
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر است که به آن پاس داده می‌شود.

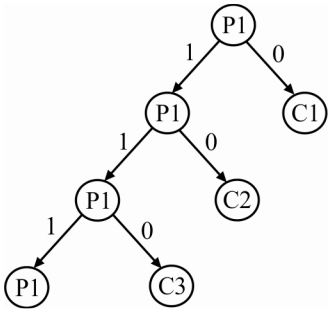
توجه: یک فرآیند فرزند **(C3)** می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2603

khalilifar.ir

نتیجه: در حالت کلی دستور fork داخل if یعنی (()if (fork باعث می‌شود که فرزند، نازا باشد، یعنی خود فرزند توسط پدر به دنیا می‌آید، اما فرزند، توان زاییدن و زاد و ولد را ندارد.

شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2601
Process id = 2602
Process id = 2603
Process id = 2600
```

مثال: با اجرای قطعه کد زیر در نهایت چند فرآیند خواهیم داشت؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600، pid واقعی فرآیند فرزند اول برابر مقدار 2601، pid واقعی فرآیند فرزند دوم برابر مقدار 2602 و pid واقعی فرآیند فرزند سوم برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include <sys/types.h>  /*pid_t*/
int  main()
{
  if (fork())
     if(!fork())

fork();

   printf ("Process id = %d \n", getpid ());

   return 0;
}
```

۱) 0 ۲) 2 ۳) 4 ۴) 8

پاسخ- گزینه (۳) صحیح است.

khalilifar.ir

توجه: در اولین خط، دستور (if (fork()) قرار دارد، وقتی که دستور ()fork داخل دستور if اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (2601)
if (!fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است.

if (2601)	TRUE

توجه: در ابتدا با فراخوانی دستور ()fork یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی (2601) if برابر مقدار TRUE خواهد بود. که منجر به این می‌شود که دستورات بعدی فرآیند پدر مورد بررسی قرار گیرد.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
if (0)
if (!fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود.

if (0)	FALSE

توجه: از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی (0) if برابر مقدار FALSE خواهد بود. که منجر به این می‌شود که دستورات fork دوم و fork سوم مورد بررسی قرار نگیرد. اما از آنجا که دستور printf مستقل و خارج از بدنه دستورات شرطی است، در انتهای فرآیند فرزند اول اجرا می‌شود.

khalilifar.ir

توجه: دقت داشته باشید که if (!fork()) دوم داخل بدنه if (fork()) اول است و fork() سوم داخل بدنه if (!fork()) دوم است. بنابراین اگر if (fork()) اول برقرار نباشد، if (!fork()) دوم و fork() سوم به تبع آن اجرا نخواهد شد.

توجه: یک فرآیند فرزند (C1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (!2602) = if(0)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

if (!2602) = if(0)	FALSE

توجه: در ابتدا با فراخوانی دستور fork() یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی if (!2602) = if(0) برابر مقدار FALSE خواهد بود. که منجر به این می‌شود که دستور fork سوم مورد بررسی قرار نگیرد. اما از آنجا که دستور printf مستقل و خارج از بدنه دستورات شرطی است، در انتهای فرآیند پدر اجرا می‌شود.

وجه: یک فرآیند پدر (P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2600

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
if (!0) = if(1)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

if (!0) = if(1)	TRUE

khalilifar.ir

توجه: از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی if (!0) = if(1) برابر مقدار TRUE خواهد بود. که منجر به این می‌شود که دستورات بعدی فرآیند فرزند دوم مورد بررسی قرار گیرد.

توجه: دقت بسیار زیاد داشته باشید که ()fork سوم داخل بدنه if (!fork()) دوم است. بنابراین اگر if (!fork()) دوم برقرار باشد، ()fork سوم به تبع آن اجرا خواهد شد.

دستوراتی که توسط فرآیند پدر(C2) اجرا می‌شود، به صورت زیر است:

2603=fork();
printf ("Process id = %d \n", getpid ());

توجه: وقتی که دستور()fork سوم اجرا می‌شود یک فرآیند فرزند سوم (جدید) از روی فرآیند پدر(فرآیند فرزند دوم) ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر (فرآیند فرزند دوم) و فرزند (فرآیند فرزند سوم) خط بعد از دستور fork را اجرا می کنند.

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر (فرآیند فرزند دوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر (فرآیند فرزند دوم) در واقع pid یعنی Process id فرآیند فرزند سوم است.

توجه: یک فرآیند پدر (C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2602**

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

0
printf ("Process id = %d \n", getpid ());

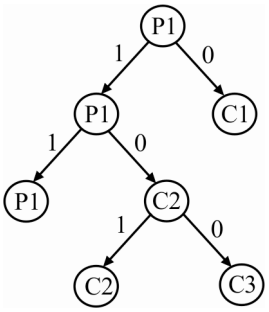
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C3) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2603**

شکل زیر گویای مطلب است:

khalilifar.ir



خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2601
Process id = 2600
Process id = 2602
Process id = 2603
```

با اجرای کد زیر در نهایت چند پردازه خواهیم داشت؟

```
main ()
{
      for(i = 1 ;i < 4 ;i++)
           fork();
}
```

توجه: مطابق قواعد زبان‌های برنامه‌نویسی، قطعه کد فوق معادل و هم ارز قطعه کد زیر است:

```
main ()
{
    fork();
    fork();
    fork();
}
```

توجه: دقت داشته باشید که یکی از بهترین شیوه‌های شمارش تعداد کل فرآیندها، قرار دادن یک دستور printf جهت نمایش Process id توسط دستور ()getpid در انتهای قطعه کد پایه است، به همین جهت به قطعه کد فوق یک دستور printf به صورت زیر اضافه شده است:

khalilifar.ir

```
main ()
{
    fork();
    fork();
    fork();
    printf ("Process id = %d \n", getpid ());
}
```

توجه: در اولین خط، دستور ()fork قرار دارد، وقتی که دستور()fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2601=fork();
fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
0
fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود.

دستورات که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2602=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

khalilifar.ir

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
0=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

```
2603=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند اول) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند سوم است.

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

```
0=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر است که به آن پاس داده می‌شود.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2604=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند چهارم است.

توجه: یک فرآیند پدر (P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

دستوراتی که توسط فرآیند فرزند چهارم (C4) اجرا می‌شود، به صورت زیر است:

```
0
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند چهارم برابر مقدار صفر است که به آن پاس داده می‌شود.

khalilifar.ir

توجه: یک فرآیند فرزند (C4) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2604

دستوراتی که توسط فرآیند پدر (C2) اجرا می‌شود، به صورت زیر است:

2605=fork();
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند دوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند پنجم است.

توجه: یک فرآیند پدر(C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2602

دستوراتی که توسط فرآیند فرزند پنجم (C5) اجرا می‌شود، به صورت زیر است:

0
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند پنجم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C5) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2605

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

2606=fork();
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند اول) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند ششم است.

توجه: یک فرآیند پدر (C1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

دستوراتی که توسط فرآیند فرزند ششم (C6) اجرا می‌شود، به صورت زیر است:

khalilifar.ir

0
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند ششم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C6) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2606**

دستوراتی که توسط فرآیند پدر (C3) اجرا می‌شود، به صورت زیر است:

2607=fork();
printf ("Process id = %d \n", getpid ());

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند سوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند هفتم است.

توجه: یک فرآیند پدر(C3) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2603**

دستوراتی که توسط فرآیند فرزند هفتم (C7) اجرا می‌شود، به صورت زیر است:

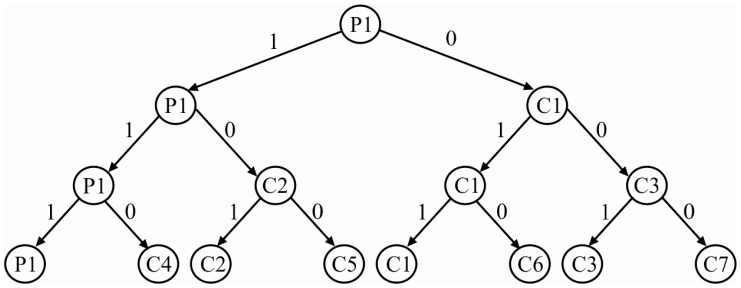
0
printf ("Process id = %d \n", getpid ());

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند هفتم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C7) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2607**

شکل زیر گویای مطلب است:



khalilifar.ir

خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2600
Process id = 2604
Process id = 2602
Process id = 2605
Process id = 2601
Process id = 2606
Process id = 2603
Process id = 2607
```

توجه: یک کار مهم دیگری هم که می‌توان انجام داد این است که به جای اینکه در فرآیند فرزند کد متفاوتی که خودمان نوشته‌ایم اجرا شود، می‌توان کاری کرد که یک برنامه‌ی دیگر که به صورت فایل اجرایی است اجرا شود. برای این کار می‌توان از دستورات متفاوتی استفاده کرد که یکی از آن‌ها دستور execvp برای جایگزین کردن تصویر حافظه فرآیند است که از خانواده دستورات exec می‌باشد. این دستور کل فضای حافظه‌ی فرآیند فرزند را پاک می‌کند و در آن برنامه‌ی جدیدی که در execvp آمده است را می‌ریزد و هیچ اثری از کدها و داده‌های قبلی که در آن بوده نمی‌ماند. این تابع نام فایل اجرایی و آرگومان‌هایی که باید به برنامه پاس داده شود را می‌گیرد و آن را در فضای حافظه‌ی فرآیند فرزند بارگذاری می‌کند و آن را اجرا می‌کند.

مثال: در کد زیر یک نمونه از استفاده‌ی execvp() را می‌بینید که در آن برنامه‌ی ls لینوکس را اجرا می‌کند، که فایل‌های موجود در یک دایرکتوری را نمایش می‌دهد و همین طور فرآیند پدر اعداد 1 تا 99 را در خروجی نمایش می‌دهد:

```
//gcc 5.4.0
#include <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include <sys/types.h>  /*pid_t*/
int main()
{
    pid_t pid;
/*fork a child process*/
    pid = fork();
    if (pid < 0) {
        printf("Fork Faild!");
}
   else if(pid == 0){  /*child process/*
      char *arg[] = {"/bin/ls" , "-l","-a", NULL};
      execvp(arg[0],arg);
}
```

khalilifar.ir

```
    else { /*parent process/*
        wait(NULL);
        printf ("***Child Complete*** \n");

       for (int i = 0; i < 5; ++i) {
          printf ("Parent process counter :%d \n",i);
}

}
   return 0;
}
```

توجه: در اولین خط، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
char *arg[] = {"/bin/ls" , "-l","-a", NULL};
execvp(arg[0],arg);
```

توجه: پس از فراخوانی تابع ()execvp، کل فضای آدرس فرآیند فرزند پاک می‌شود و در آن برنامه‌ی جدیدی که در ()execvp آمده است قرار می‌گیرد و هیچ اثری از کدها و داده‌های قبلی که در آن بوده نمی‌ماند. این تابع نام فایل اجرایی و آرگومان‌هایی که باید به برنامه پاس داده شود را می‌گیرد و آن را در فضای حافظه‌ی فرآیند فرزند بارگذاری می‌کند و آن را اجرا می‌کند. بنابراین پس از اجرای تابع ()execvp، قطعه کد و برنامه جدید اجرا می‌شود و کنترل اجرای برنامه دیگر هیچ وقت به دستور قبل و بعد تابع ()execlp باز نمی‌گردد.

دستورات بعدی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
printf ("***Child Complete*** \n");
```

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

printf ("*Child Complete \n***");**	*****Child Complete*****

khalilifar.ir

توجه: مقادیر متغیر محلی i در فرآیند پـدر توسـط حلقـه for از 0 تـا 4 در خروجـی نمـایش داده می‌شود.

```
for (int i = 0; i < 5; ++i) {
        printf ("Parent process counter :%d \n",i);
}
```

printf ("Parent process counter :%d \n",i);	**Parent process counter :0** **Parent process counter :1** **Parent process counter :2** **Parent process counter :3** **Parent process counter :4**

خروجی نهایی برنامه به صورت زیر است:

```
total 88
drwxr-xr-x 22 root              root              4096 Jun 27 18:52 .
drwxrwxr-x  3 root              ren               4096 May 21 16:27 ..
drwxr-xr-x  2 rextester_user31  rextester_user31  4096 Jun 22 05:52 1311737820
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 1358939562
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 1422882116
drwxr-xr-x  2 root              root              4096 Jun 22 05:50 1471488187
drwxr-xr-x  2 rextester_user127 rextester_user127 4096 Jun 27 18:52 1516783416
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 1521451137
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 1642473737
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 1790473723
drwxr-xr-x  2 root              root              4096 Jun 22 05:51 1857273623
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 2098476182
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 213214471
drwxr-xr-x  2 rextester_user295 rextester_user295 4096 Jun 27 18:52 404136654
drwxr-xr-x  2 root              root              4096 Jun 22 05:51 521610754
drwxr-xr-x  2 root              root              4096 Jun 22 05:50 703843859
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 7215752
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 793881491
drwxr-xr-x  2 root              root              4096 Jun 22 05:50 815181955
drwxr-xr-x  2 rextester_user184 rextester_user184 4096 Jun 21 18:53 861836283
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 945697616
drwxr-xr-x  2 root              root              4096 Jun 22 05:51 963143096
Parent process counter :0
Parent process counter :1
Parent process counter :2
Parent process counter :3
Parent process counter :4
***Child Complete***
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، خط printf("LINE J") پـس از اجـرای دسـتور execlp() چندبار اجرا می‌شود؟

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/
```

khalilifar.ir

```
int  main()
{
    pid_t  pid;

/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}
else if (pid == 0) { /* child process*/

    execlp("/bin/ls" , "ls" , NULL);
    printf("LINE J");
}
else { /* parent process */
   /* parent will wait for the child to complete */
   wait(NULL);
   printf ("***Child Complete*** \n");
 }
  return 0;
}
```

۱) 0 بار ۲) 1 بار ۳) 2 بار ۴) 3 بار

پاسخ- گزینه (۱) صحیح است.

توجه: در اولین خط، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
execlp("/bin/ls" , "ls" , NULL);
printf("LINE J");
```

توجه: پس از فراخوانی تابع ()execlp، کل فضای آدرس فرآیند فرزند پاک می‌شود و در آن برنامه‌ی جدیدی که در ()execlp آمده است قرار می‌گیرد و هیچ اثری از کدها و داده‌های قبلی که در آن بوده نمی‌ماند. این تابع نام فایل اجرایی و آرگومان‌هایی که باید به برنامه پاس داده شود را می‌گیرد و آن را در فضای حافظه‌ی فرآیند فرزند بارگذاری می‌کند و آن را اجرا می‌کند. بنابراین

khalilifar.ir

پس از اجرای تابع ()execlp، قطعه کد و برنامه جدید اجرا می‌شود و کنترل اجرای برنامه دیگر هیچ وقت به دستور قبل و بعد تابع ()execlp باز نمی‌گردد، در این حالت پس از اجرای تابع ()execlp، خط printf("LINE J") دیده نخواهد شد و به تبع هرگز اجرا هم نخواهد شد. اما اگر اجرای تابع ()execlp موفقیت آمیز نباشد، آنگاه کنترل برنامه به خط بعد از تابع ()execlp باز می‌گردد و خط printf("LINE J") اجرا و چاپ می‌شود.

دستورات بعدی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

printf ("***Child Complete*** \n");

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

printf ("***Child Complete \n***");	***Child Complete***

خروجی نهایی برنامه به صورت زیر است:

```
7215752
793881491
815181955
861836283
945697616
963143096
***Child Complete***
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، مقدار خروجی در خطوط LINE X و LINE Y برابر کدام گزینه است؟

```
//gcc 5.4.0
#include <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include <sys/types.h>  /*pid_t*/

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int  main()
{
int i;
pid_t  pid;
```

khalilifar.ir

```
   pid = fork();

   if (pid == 0) {
        for (i = 0 ; i < SIZE ; i++){
            nums[i] *= -i;
            printf("CHILD: %d \n" , nums[i]); /* LINE X*/
      }
}
else if (pid > 0) {
    wait(NULL);
    printf ("***Child Complete*** \n");
    for (i = 0; i < SIZE; i++)
         printf("PARENT: %d \n", nums[i]); /* LINE Y*/
 }
   return 0;
}
```

۱) LINE X = 0, -1, -4, -9, -16 , LINE Y = 0, 1, 2, 3, 4

۲) LINE X = -16, -9, -4, -1, 0 , LINE Y = 0, 1, 2, 3, 4

۳) LINE X = 0, -1, -4, -9, -16 , LINE Y = 4, 3, 2, 1, 0

۴) LINE X = -16, -9, -4, -1, 0 , LINE Y = 4, 3, 2, 1, 0

پاسخ- گزینه (۱) صحیح است.

توجه: برنامه اجرا می‌شود و در اولین خط مقادیر آرایه سراسری nums[5] برابر با مقادیر {0,1,2,3,4} می‌شود. دقت کنید که آرایه سراسری nums[5] بالای تابع main تعریف شده است و یک آرایه سراسری محسوب می‌شود که داخل Data Segment تعریف و مقداردهی می‌شود. همچنین دقت کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می‌شود که داخل Stack Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
for (i = 0 ; i < 5 ; i++){
     nums[i] *= -i;
     printf("CHILD: %d \n" , nums[i]); /* LINE X*/
 }
```

khalilifar.ir

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور ++i برابر مقادیر {0,1,2,3,4} می‌شود و این تغییرات روی فرآیند پدر اثری ندارد.

توجه: مطابق قوانین کوتاه‌نویسی دستورات در زبان برنامه‌نویسی C دستور nums[i] *= -i معادل دستور nums[i] =nums[i] *(-i) است.

توجه: مقادیر اولیه آرایه سراسری nums[5] برابر {0,1,2,3,4} است و پس از اجرای دستور nums[i] =nums[i] *(-i) برابر مقادیر {0,-1,-4,-9,-16} می‌شود و این تغییر روی فرآیند پدر اثری ندارد. هر چند که آرایه nums[5] سراسری است.

nums[i] =nums[i] *(-i) **printf("CHILD: %d \n" , nums[i]); /* LINE X*/**	**CHILD: 0** **CHILD: -1** **CHILD: -4** **CHILD: -9** **CHILD: -16**

دستورات بعدی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
for (i = 0 ; i < 5 ; i++){
    printf("PARENT: %d \n" , nums[i]); /* LINE Y*/
}
```

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور ++i برابر مقادیر {0,1,2,3,4} می‌شود و این تغییرات روی فرآیند فرزند اثری ندارد.

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

printf ("*Child Complete \n***");**	*****Child Complete*****

توجه: مقادیر اولیه آرایه سراسری nums[5] در فرآیند پدر برابر {0,1,2,3,4} است. هرچند که قبلا مقادیر آرایه سراسری nums[5] توسط فرآیند فرزند برابر مقادیر {0,-1,-4,-9,-16} شده است.

printf("PARENT: %d \n" , nums[i]); /* LINE Y*/	**PARENT:0** **PARENT:1** **PARENT:2** **PARENT:3** **PARENT:4**

خروجی نهایی برنامه به صورت زیر است:

khalilifar.ir

```
CHILD:0
CHILD:-1
CHILD:-4
CHILD:-9
CHILD:-16
PARENT:0
PARENT:1
PARENT:2
PARENT:3
PARENT:4
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، با اجرای کد زیر در نهایت چند فرآیند و چند نخ خواهیم داشت؟

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/
int  main()
{
    pid_t  pid;

   pid = fork();
   if (pid == 0) { /* child process*/

         fork ();
         thread_create(…);
}

fork();

printf ("Process id = %d \n", getpid ());

    return 0;
}
```

۱) 2 فرآیند و 6 نخ
۲) 6 فرآیند و 0 نخ
۳) 6 فرآیند و 2 نخ
۴) 0 فرآیند و 2 نخ

پاسخ- گزینه (۳) صحیح است.

توجه: در اولین خط، دستور ()fork قرار دارد، وقتی که دستور()fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می‌کنند.

khalilifar.ir

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2601=pid=fork();
if (pid == 0) { /* child process*/
        fork ();
        thread_create(…);
}
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است. با توجه به شرط if (pid == 0) در فرآیند پدر (P1) باعث می‌شود ()fork دوم و thread_create(…) اجرا نشود و فقط در ادامه ()fork سوم مورد بررسی قرار بگیرد.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
0=pid=fork();
if (pid == 0) { /* child process*/
        fork ();
        thread_create(…);
}
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود. و با توجه به شرط if (pid == 0) در فرآیند فرزند اول (C1) باعث می‌شود ()fork دوم و thread_create(…) اجرا شود و همچنین در ادامه ()fork سوم هم مورد بررسی قرار بگیرد.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2602=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

khalilifar.ir

توجه: یک فرآیند پدر (P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2600

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
0=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2602

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

```
2603=fork();
thread_create(…);
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند اول) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند سوم است.

توجه: توسط دستور thread_create(…) نخ اول ایجاد می‌شود.

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

```
0=fork();
thread_create(…);
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: توسط دستور thread_create(…) نخ دوم ایجاد می‌شود.

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

```
2604=fork();
printf ("Process id = %d \n", getpid ());
```

khalilifar.ir

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند چهارم است.

توجه: یک فرآیند پدر (C1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

دستوراتی که توسط فرآیند فرزند چهارم (C4) اجرا می‌شود، به صورت زیر است:

0
printf ("Process id = %d \n", getpid ());

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند چهارم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C4) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2604

دستوراتی که توسط فرآیند پدر (C3) اجرا می‌شود، به صورت زیر است:

2605=fork();
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند سوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند پنجم است.

توجه: یک فرآیند پدر(C3) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2603

دستوراتی که توسط فرآیند فرزند پنجم (C5) اجرا می‌شود، به صورت زیر است:

0
printf ("Process id = %d \n", getpid ())

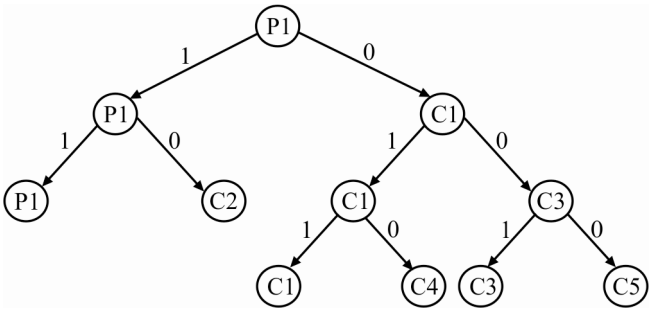
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند پنجم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C5) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2605

khalilifar.ir

شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2600
Process id = 2602
Process id = 2601
Process id = 2604
Process id = 2603
Process id = 2605
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که مقدار i، مقدار pid حاصل از بازگشت fork و مقدار Process id حاصل از بازگشت getpid را در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0

#include <stdio.h>   /* printf */
#include <unistd.h>  /*fork */
#include <sys/types.h>  /* pid_t */
int  main()
{
    pid_t  pid;
    int i = 0;

        pid = fork (); /*fork a child process*/

        if (pid > 0) { /*Parent Process:*/

          printf ("*** Parent Process Begin *** \n");

          printf ("&i = %d \n" , &i);

          i = i + 1 ;
          printf ("i = %d \n" , i);
```

khalilifar.ir

```
        printf ("&i = %d \n" , &i);

        printf ("Process id = %d \n", getpid () );
        printf ("pid = %d \n" , pid);

        printf ("*** Parent Process End *** \n");
    }
     else if (pid == 0) { /*Child Process:*/

        printf ("*** Child Process Begin *** \n");

        printf ("&i = %d \n" , &i);

        i = i - 1 ;
        printf ("i = %d \n" , i);

        printf ("&i = %d \n" , &i);

         printf ("Process id = %d \n", getpid () );
         printf ("pid = %d \n" , pid);

         printf ("*** Child Process End *** \n");
     }

      else { /*error occurred*/
       printf ("fork creation failed!!! \n ");
      }

   return 0;
}
```

توجه: برنامه اجرا می‌شود و در اولین خط مقدار متغیر محلی i برابر با 0 می‌شود. دقت کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می‌شود که داخل Stack Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دستور fork را اجرا می‌کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
printf ("&i = %d \n" , &i);

i = i + 1 ;
printf ("i = %d \n" , i);

printf ("&i = %d \n" , &i);

printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

khalilifar.ir

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور i = i + 1 برابر 1 می‌شود و این تغییر روی فرآیند فرزند اثری ندارد.

توجه: به تفاوت آدرس مجازی و آدرس فیزیکی در این قطعه کد توجه نمایید، عبارت &i دسترسی به آدرس مجازی را ایجاد می‌کند، و آدرس مجازی فرآیند پدر و فرآیند فرزند کاملا **یکسان** است. اما آدرس فیزیکی متغیر i در فرآیند پدر و فرآیند فرزند کاملا **متفاوت** است.

printf ("&i = %d \n" , &i); **i = i + 1;** **printf ("i = %d \n" , i);** **printf ("&i = %d \n" , &i);**	**&i = 10000** **i=1** **&i = 10000**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 2603**

دستورات بعدی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
printf ("&i = %d \n" , &i);

i = i - 1 ;
printf ("i = %d \n" , i);

printf ("&i = %d \n" , &i);

printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور i = i - 1 برابر 1- می‌شود و این تغییر روی فرآیند پدر اثری ندارد.

توجه: به تفاوت آدرس مجازی و آدرس فیزیکی در این قطعه کد توجه نمایید، عبارت &i دسترسی به آدرس مجازی را ایجاد می‌کند، و آدرس مجازی فرآیند پدر و فرآیند فرزند کاملا **یکسان** است. اما آدرس فیزیکی متغیر i در فرآیند پدر و فرآیند فرزند کاملا **متفاوت** است.

khalilifar.ir

printf ("&i = %d \n" , &i); i = i + 1; printf ("i = %d \n" , i); printf ("&i = %d \n" , &i);	&i = 10000 i=-1 &i = 10000

توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2603

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

pid = fork; printf ("pid = %d \n" , pid);	pid = 0

خروجی نهایی برنامه به صورت زیر است:

```
*** Parent Process Begin ***
&i = 10000
i = 1
&i = 10000
Process id = 2600
pid = 2603
*** Parent Process End ***
*** Child Process Begin ***
&i = 10000
i = -1
&i = 10000
Process id = 2603
pid = 0
*** Child Process End ***
```

۹- گزینه (۱) صحیح است.

به طور کلی برای تشخیص اتمام انتقال داده‌ها دو روش در کامپیوترها قابل استفاده است. یکی روش سرکشی یا نمونه‌برداری (Polling) و دیگری روش وقفه (Interupt) است.

در روش سرکشی (Polling) هر دستگاه دارای ثباتی کنترلی است که CPU با بررسی آن می‌تواند متوجه شود که آیا عملیات انتقال آن دستگاه تمام شده است یا خیر. لذا در این تکنیک CPU

khalilifar.ir

می‌بایست در یک پریود زمانی، مرتبا ثبات‌های کنترلی دستگاه‌های مختلف را به صورت سرکشی بررسی کند که باعث اتلاف وقت CPU و به تبع کاهش بهره‌وری CPU است. با آنکه پیاده‌سازی این روش ساده است و به امکانات سخت‌افزاری خاصی نیاز ندارد ولی سرعت آن پایین است. این روش مانند کلاس درسی است که در آن استاد هر چند دقیقه یکبار از تک تک دانشجویان به ترتیب بپرسد که آیا سوالی دارند یا خیر، همچنین روش سرکشی باعث اتلاف وقت دستگاه‌های IO و به تبع کاهش بهره‌وری دستگاه‌های IO می‌شود، چون برای مثال اگر کار IO یک دستگاه ورودی و خروجی تمام شود باید آنقدر معطل بماند تا نوبت سرکشی آن توسط CPU فرا برسد. اما در روش وقفه (Interupt) هر دستگاه دارای سیگنال کنترلی مخصوص به خود است که به نحوی به CPU ارتباط دارد. هرگاه انتقال داده‌ها توسط آن دستگاه تمام شود، سیگنالی را به نام وقفه به سمت CPU می‌فرستد تا آن را از این موضوع مطلع سازد. در این حالت پردازنده، پردازش جاری را متوقف ساخته و به سرویس‌دهی این وقفه می‌پردازد. این روش مانند کلاس درسی است که در آن هر دانشجویی که سوال دارد در هر زمان با بالا بردن دست خود این موضوع را به اطلاع استاد می‌رساند. در این حالت استاد در اولین زمان مناسب صحبت‌های جاری خود را قطع کرده و پاسخ آن دانشجو را می‌دهد. سرعت این روش از روش سرکشی بیشتر است. همچنین روش وقفه باعث افزایش بهره‌وری CPU و دستگاه‌های IO می‌شود. تمام کامپیوترها راهکاری را فراهم می‌کنند تا قسمت‌های مختلف کامپیوتر (مانند ورودی و خروجی) روند اجرای دستورالعمل‌ها توسط پردازنده را جهت سرویس‌دهی‌های دیگری قطع کنند. در واقع مکانیزمی را فراهم می‌کند تا اجرای دستورالعمل‌های جاری پردازنده موقتاً متوقف شده و دستورات سرویس‌دهی دیگری اجرا شوند، پس از آن دوباره کنترل به همان برنامه باز می‌گردد. سیستم عامل جهت تسریع در پاسخ دادن به وقفه‌ها، آدرس روال‌های سرویس‌دهنده به آنها را در جدولی به نام جدول توصیف وقفه **(Interrupt Descriptor Table)** نگهداری می‌کند که به ازای هر وقفه یک درایه در این جدول وجود دارد که به آن بردار وقفه **(Interrupt Vector)** می‌گویند.

صورت سوال به این شکل است:

عامل اصلی برای استفاده سیستم عامل از وقفه کدام است؟

۱) افزایش بهره‌وری

گزینه اول پاسخ سوال است، زیرا عامل اصلی برای استفاده سیستم عامل از وقفه، افزایش بهره‌وری است. روش وقفه باعث افزایش بهره‌وری CPU و دستگاه‌های IO می‌شود.

۲) سادگی در پیاده‌سازی

گزینه دوم پاسخ سوال نیست، زیرا روش سرکشی پیاده‌سازی ساده‌تری نسبت به روش وقفه دارد.

khalilifar.ir

۳) کاهش زمان ارتباطات

گزینه سوم پاسخ سوال نیست، زیرا کاهش زمان ارتباطات مرتبط با بحث شبکه‌های کامپیوتری یعنی کاهش زمان تاخیر پردازش (T_F)، کاهش زمان تاخیر انتشار (T_{Prob})، کاهش زمان تاخیر صف (T_{queue}) و کاهش زمان تاخیر پردازش ($T_{Process}$) است.

۴) جلوگیری از اتلاف در IO

گزینه چهارم پاسخ سوال نیست، زیرا روش وقفه باعث افزایش بهره‌وری CPU و دستگاه‌های IO می‌شود، اما باعث جلوگیری 100 درصد از اتلاف و به تبع بهره‌وری 100 درصد در CPU و دستگاه‌های IO نمی‌شود.

۱۰- گزینه (۱) صحیح است.

الگوریتم SJF (Shortest Job First)

در این روش ابتدا کاری برای اجرا انتخاب می‌شود که از همه کوتاهتر باشد (زمان اجرای کمتری داشته باشد).

توجه: این الگوریتم، **SPN** (Shortest Process Next) و **SPT** (Shortest Process Time) نیز نامیده می‌شود.

توجه: SJF یک الگوریتم **انحصاری** (Non Preemptive) است. در سایر متون فارسی به الگوریتم انحصاری، الگوریتم «**غیرقبضه‌ای**» یا «**غیرقابل پس گرفتن**» یا «**غیرقابل تخلیه پیش هنگام**» نیز گفته می‌شود.

توجه: یک نقص عمده الگوریتمSJF این است که ممکن است باعث قحطی‌زدگی فرآیندهای طولانی شود. به این ترتیب که اگر همواره تعدادی فرآیند کوچک وارد سیستم شوند، اجرای فرآیندهای بزرگ به طور متناوب به تعویق می‌افتد. این روال حتی می‌تواند تا بینهایت ادامه یابد و هیچگاه نوبت به فرآیندهای بزرگ نرسد!!!!!

توجه: در این روش اگر دو فرآیند مدت زمان اجرای برابر داشته باشند، بر اساس FCFS زمان‌بندی می‌شوند.

توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسخ و میانگین زمان گردش کار (بازگشت) فرآیندهاست.

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآیندها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجرای فرآیندها را فقط حدس زده و به طور تقریبی بدست آورد.

khalilifar.ir

الگوریتم SRT (Shortest Remaining Time)

این الگوریتم نسخه **غیرانحصاری** (Preemptive) الگوریتم SJF است. در سایر متون فارسی به الگوریتم غیرانحصاری، الگوریتم «**قبضه‌ای**» یا «**قابل پس گرفتن**» یا «**قابل تخلیه پیش هنگام**» نیز گفته می‌شود.

در این الگوریتم اگر حین اجرای یک فرآیند، فرآیندی وارد شود که زمان اجرای کوتاه‌تری داشته باشد، پردازنده را در اختیار می‌گیرد.

توجه: این الگوریتم، **SRPT** (Shortest Remaining Processing Time)،

SRTF (Shortest Remaining Time First) و

SRTN (Shortest Remaining Time Next) نیز نامیده می‌شود.

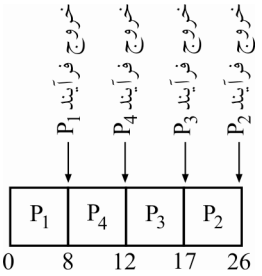
توجه: اگر لحظه ورود همه فرآیندها یکی باشد، الگوریتم SRT مشابه SJF عمل می‌کند.

توجه: در الگوریتم **SRT** نیز همانند الگوریتم SJF، احتمال وقوع **قحطی‌زدگی برای کارهای بزرگ** وجود دارد.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	0	8		
P_2	1	9		
P_3	2	5		
P_4	3	4		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



توجه: در الگوریتم‌های انحصاری HRRN و SJF لحظه خروج اول یک فرآیند با لحظه خروج کامل یک فرآیند برابر است، بنابراین در حل این سوال زمان پاسخ و زمان بازگشت یکسان در نظر گرفته شده است.

زمان ورود فرآیند – زمان خروج اول فرآیند = زمان پاسخ فرآیند

زمان ورود فرآیند – زمان خروج کامل فرآیند = زمان بازگشت فرآیند

khalilifar.ir

P_1 زمان بازگشت $= 8 - 0 = 8$

P_2 زمان بازگشت $= 26 - 1 = 25$

P_3 زمان بازگشت $= 17 - 2 = 15$

P_4 زمان بازگشت $= 12 - 3 = 9$

میانگین زمان بازگشت $= \text{ATT} = \frac{8+25+15+9}{4} = \frac{57}{4} = 14.25$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 8 - 8 = 0$

P_2 زمان انتظار $= 25 - 9 = 16$

P_3 زمان انتظار $= 15 - 5 = 10$

P_4 زمان انتظار $= 9 - 4 = 5$

میانگین زمان انتظار $= \text{AWT} = \frac{0+16+10+5}{4} = \frac{31}{4} = 7.75$

میانگین زمان اجرا $= \text{AST} = \frac{8+9+5+4}{4} = \frac{26}{4} = 6.5$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

14.25= 6.5 + 7.75

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	0	8	0	8
P_2	1	9	16	25
P_3	2	5	10	15
P_4	3	4	5	9

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

14.25 7.75 6.5

khalilifar.ir

در الگوریتم HRRN

برای تعیین اولویت یک فرآیند در الگوریتم HRRN از فرمول زیر استفاده می‌شود:

$$\text{اولویت} = \frac{\text{زمان انتظار + زمان اجرا}}{\text{زمان اجرا}} = \frac{\text{زمان انتظار}}{\text{زمان اجرا}} + 1$$

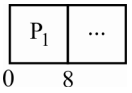
نکته: HRRN یک الگوریتم **انحصاری** است.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان پاسخ =
P_1	0	8		
P_2	1	9		
P_3	2	5		
P_4	3	4		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:

در این الگوریتم در زمان صفر فقط فرآیند P_1 قرار دارد که به شکل انحصاری اجرا می‌گردد.



در لحظه 8 فرآیندهای P_2، P_3 و P_4 در صف آماده قرار دارند. انتظار هر یک از فرآیندهای P_2، P_3 و P_4 تا لحظه 8 به صورت زیر است:

$$P_2 \text{انتظار} = 8 - 1 = 7$$

$$P_3 \text{انتظار} = 8 - 2 = 6$$

$$P_4 \text{انتظار} = 8 - 3 = 5$$

در ادامه براساس رابطه اولویت داریم:

$$\text{اولویت}\,(P_2) = \frac{7}{9} + 1 = 1.7$$

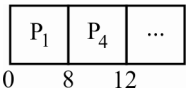
$$\text{اولویت}\,(P_3) = \frac{6}{5} + 1 = 2.2$$

$$\text{اولویت}\,(P_4) = \frac{5}{4} + 1 = 2.25$$

$$\text{اولویت}\,(P_4) > \text{اولویت}\,(P_3) > \text{اولویت}\,(P_2)$$

واضح است که P_4 بالاترین اولویت را دارد، پس در زمان 8 فرآیند P_4 انتخاب می‌شود که به شکل انحصاری اجرا می‌گردد.

khalilifar.ir



در لحظه 12 فرآیندهای P_2 و P_3 در صف آماده قرار دارند. انتظار هر یک از فرآیندهای P_2 و P_3 تا لحظه 12 به صورت زیر است:

$$P_2 \text{انتظار} = 12 - 1 = 11$$

$$P_3 \text{انتظار} = 12 - 2 = 10$$

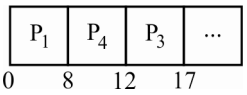
در ادامه براساس رابطه اولویت داریم:

$$\text{اولویت}(P_2) = \frac{11}{9} + 1 = 2.2$$

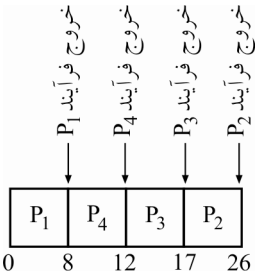
$$\text{اولویت}(P_3) = \frac{10}{5} + 1 = 3$$

$$\text{اولویت}(P_3) > \text{اولویت}(P_2)$$

واضح است که P_3 بالاترین اولویت را دارد، پس در زمان 12 فرآیند P_3 انتخاب می‌شود که به شکل انحصاری اجرا می‌گردد.



در لحظه 17 فقط فرآیند P_2 در صف آماده قرار دارد. پس در زمان 17 فرآیند P_2 انتخاب می‌شود که به شکل انحصاری اجرا می‌گردد.



با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	0	8	0	8
P_2	1	9	16	25
P_3	2	5	10	15
P_4	3	4	5	9

khalilifar.ir

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا
6.5 7.75 14.25

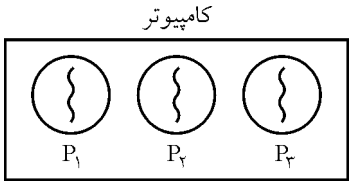
توجه: واضح است که زمان‌بندی‌های دو الگوریتم SJF و HRRN یکسان است، بنابراین پُرواضح است که گزینه اول پاسخ سوال است.

۱۰۰- گزینه (۴) صحیح است.

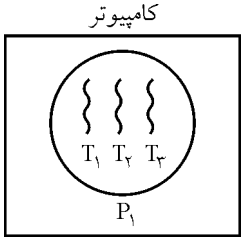
نخ (Thread)

در سیستم‌های قدیمی‌تر، به ازای هر فرآیند یک رشته نخ یا رشته اجرایی و به تبع یک شمارنده برنامه (PC) وجود داشت اما در سیستم عامل‌های امروزی به ازای هر فرآیند می‌توان چند نخ یا رشته اجرایی داشت.

شکل زیر سه فرآیند معمولی را نشان می‌دهد که هریک برای خودشان یک رشته اجرایی و یک حافظه مختص به خود را دارند.

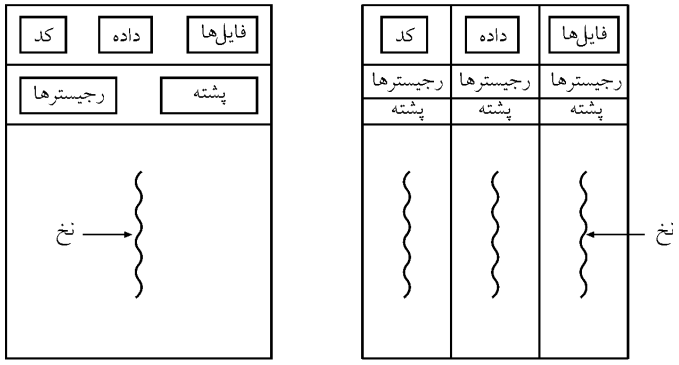


ولی در شکل زیر یک فرآیند، سه رشته اجرایی دارد که هر یک رجیستر، پشته و شمارنده برنامه (PC) مجزای خود را دارند و مانند فرآیندها می‌توانند همروند (در سیستم‌های تک‌پردازنده‌ای) و موازی (در سیستم‌های چندپردازنده‌ای) اجرا شوند.



توجه: نخ‌های همتا که در یک فرآیند قرار دارند و از کد، داده، heap و منابع مشترک استفاده می‌کنند اما هر نخ، شمارنده برنامه، مجموعه رجیستر و فضای پشته جداگانه‌ای در اختیار دارد. در واقع هر نخ، TCB مجزایی دارد.

khalilifar.ir



توجه: از آن‌جا که نخ‌های همتا در یک فرآیند قرار داشته و اشتراکات زیادی با هم دارند، عمل تعویض متن بین آنها به راحتی و با هزینه کمتری صورت می‌گیرد، در واقع TCB مربوط به نخ‌ها، محتوی کمتری نسبت به PCB فرآیندها دارد، برای مثال لیست فایل‌های باز مربوط به فرآیندها است، بنابراین این لیست به هنگام تعویض متن فرآیندها باید داخل PCB مربوط به فرآیند ذخیره گردد، در حالی که به هنگام تعویض متن بین نخ‌ها نیازی به ذخیره‌سازی لیست فایل‌های باز مربوط به یک فرآیند در TCB یک نخ نیست. بنابراین تعویض متن بین نخ‌ها نسبت به فرآیندها ارزان‌تر است.

چند نخی در زبان #C

#C پیاده سازی چند نخی را پشتیبانی می‌کند. در زبان #C، هر برنامه به طور پیش فرض از یک نخ تشکیل شده است و درصورت ایجاد نخ‌های دیگر، مفهوم چندنخی پیاده‌سازی می‌گردد.

توجه: نخ اول به صورت پیش‌فرض وجود دارد و برنامه با نخ اول شروع به اجرا می‌کند.

مثال: در قطعه کد زیر نخ T_1 به طور پیش‌فرض وجود دارد و نخ T_2 ایجاد می‌گردد:

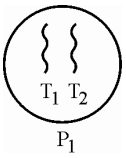
```
static void main ()
    {
Thread    T2=new    Thread (Go);
→   T2.Start( )
→   Go ( );
        }
static void Go( )
{
for (int i = 0 ; i<5 ; i++)
        console.write("*");
}
```

در این بخش نخ T2 ایجاد می‌گردد و فعالیت Go داخل این نخ قرار می‌گیرد.

نخ T2 اجرا می‌گردد.

فعالیت Go داخل نخ T1 قرار داده شده است. می‌شود فعالیت دیگری تعریف شود و در این بخش فراخوانی شود.

khalilifar.ir



توجه: فعالیت Go داخل نخ T_1 قرار دارد، اما فعالیت Go در نخ T_2 هم قرار داده شده است. در دو نخ T_1و T_2 که فعالیت Go داخل آن قرار دارد، متغیر محلی i در داخل پشته مربوط به هر نخ ایجاد می‌گردد.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

****

چاپ 10 عدد ستاره به دلیل اجرای همروند (سیستم تک‌پردازنده‌ای) یا موازی (سیستم چندپردازنده‌ای) دو نخ T_1 و T_2 است!

توجه: چاپ 10 عدد ستاره نشانه این است که هر نخ پشته مختص به خود را دارد.

توجه: نخ از دو بخش ساختاری (ظرف) و محتوایی تشکیل شده است، ساختار نخ در زبان C# توسط دستور new ایجاد می‌شود، و محتوای نخ توسط قرار دادن یک قطعه کد به درون آن پُر می‌شود. در مثال بالا دو ساختار و ظرف نخ T_1و T_2 توسط دستور new ایجاد شده است و قطعه کد Go به عنوان محتوای نخ T_1و T_2 در نظر گرفته شده است. بنابراین نخ‌های یک پردازه(فرآیند)، دارای برنامه (قطعه کد) مخصوص به خود هستند.

مثال: در قطعه کد زیر نخ T_1 به طور پیش‌فرض وجود دارد و نخ T_2 ایجاد می‌گردد:

```
static void main ()
   {
     Thread    T2=new    Thread (func);
T2.Start( ) ;                          ↓
console.write("X");                در این بخش نخ T2 ایجاد
   }                               می‌گردد و فعالیت func
static void func( )                داخل این نخ قرار می‌گیرد.
{
console.write("Y");
}
```

→ نخ T2 اجرا می‌گردد.

→ فعالیت چاپ "X" داخل نخ T1 قرار داده شده است.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

XY

توجه: نخ از دو بخش ساختاری (ظرف) و محتوایی تشکیل شده است، ساختار نخ در زبان C# توسط دستور new ایجاد می‌شود، و محتوای نخ توسط قرار دادن یک قطعه کد به درون آن پُر می‌شود. در مثال بالا دو ساختار و ظرف نخ T_1و T_2 توسط دستور new ایجاد شده است و قطعه

khalilifar.ir

کد write("X") به عنوان محتوای نخ T_1 و قطعه کد write("Y") به عنوان محتوای نـخ T_2 در نظـر گرفته شده است. بنابراین نخ‌های یک پردازه(فرآیند)، دارای برنامه(قطعه کد) مخصـوص بـه خـود هستند.

توجه: اشتراکات نخ‌های داخل یک فرآینـد شـامل سـگمنت داده (داده سراسـری)، فضـای آدرس، heap، فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامـه (PC)، رجیسـترها و پشته می‌باشد.

صورت سوال به این شکل است:

کدام عبارت در مورد نخ‌ها درست نیست؟

۱) نخ‌های یک پردازه، دارای برنامه مخصوص به خود هستند.

گزینه اول گزاره درستی است، زیرا نخ‌های یک پردازه(فرآیند)، دارای برنامه(قطعه کد) مخصـوص به خود هستند.

۲) نخ‌های یک پردازه، از فضای heap مشترک استفاده می‌کنند.

گزینه دوم گزاره درستی است، زیرا نخ‌های یک پردازه، از فضای heap مشترک استفاده می‌کنند.

۳) نخ‌های یک پردازه، از فضای آدرس یکسان استفاده می‌کنند.

گزینه سوم گزاره درستی است، زیرا نخ‌های یک پردازه، از فضای آدرس یکسان استفاده مـی‌کننـد. کل ساختار و محتوای یـک فرآینـد داخـل یـک فضـای آدرس قـرار مـی‌گیـرد، نـخ یـک مفهـوم برنامه‌نویسی است که بخشی از یک فرآیند محسوب می‌شود.

۴) نخ‌های یک پردازه، از یک پشته مشترک استفاده می‌کنند.

گزینه چهارم گزاره درستی نیست، زیرا نخ‌های یک پردازه، از یک پشته مشترک استفاده نمـی‌کننـد. هر نخ پشته مختص به خود را دارد.

khalilifar.ir

khalilifar.ir

تست‌های کنکور کارشناسی ارشد سال ۱۳۹۹

۱- الگوریتم زیر ساختار فرآیند P_i برای حل مسئله ناحیه بحرانی (Critical-Problem) در حالتی که n فرآیند وجود داشته باشد، است. در خصوص این الگوریتم کدام گزینه صحیح است؟

(مهندسی کامپیوتر – دولتی ۹۹)

```
do {
waiting[i] = true;
key = true;
while(waiting[i] && key)
   key = test_and_set(&lock);
waiting[i] = false;
/* Critical Section */
j = (i + 1)  % n;
while (!waiting[j])
  j = (j + 1) % n;
if (j == i)
   lock = false;
/* Remainder Section */
} while (true);
```

۱) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را به ازای مقدار دلخواه n برآورده می‌کند.

۲) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را تنها به ازای مقدار n=2 برآورده می‌کند.

۳) شرط پیشرفت را تنها به ازای مقدار n بزرگتر از 2 برآورده نمی‌کند.

۴) شرط پیشرفت را هرگز برآورده نمی‌کند.

۲- الگوریتم بانکدار (Banker) برای حل کدام مسئله به کار می‌رود؟ **(مهندسی کامپیوتر – دولتی ۹۹)**

۱) دوری از بن‌بست ۲) تشخیص بن‌بست

۳) جلوگیری از بن‌بست ۴) ترمیم (Recovery) بن‌بست

۳- کدام گزینه درباره ریسه‌های (Threads) سطح کاربر و سطح هسته درست است؟

(مهندسی کامپیوتر – دولتی ۹۹)

۱) زمان‌بندی ریسه‌های سطح هسته سریعتر از ریسه‌های سطح کاربر است.

۲) ریسه‌های سطح کاربر و سطح هسته از طریق فراخوانی سیستمی (System Calls) به هم سرویس می‌دهند.

khalilifar.ir

۳) ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس هم دسترسی داشته و می‌توانند در فضای آدرس هم بنویسند.
۴) ریسه‌های سطح هسته به فضای آدرس ریسه‌های سطح کاربر دسترسی دارند، اما ریسه‌های سطح کاربر به فضای آدرس ریسه‌های سطح هسته دسترسی ندارند.

۴- در یک سیستم عامل از صفحه‌بندی وارون (Inverted Paging) استفاده می‌شود. کدام گزینه در مورد جدول صفحه درست است؟ (مهندسی کامپیوتر – دولتی ۹۹)

۱) یک جدول صفحه عمومی که بر اساس شماره قاب مرتب شده است.
۲) یک جدول صفحه عمومی که بر اساس شماره پردازه مرتب شده است.
۳) یک جدول صفحه عمومی که بر اساس شماره آدرس مجازی مرتب شده است.
۴) هر پردازه دارای یک جدول صفحه اختصاصی است که بر اساس شماره قاب مرتب شده است.

۵- در یک سیستم که تخصیص حافظه در آن بر اساس صفحه‌بندی (Paging) انجام می‌شود، اندازه هر فریم (Frame) برابر 2048 بایت است. شکل زیر، حافظه اصلی سیستم است، که قسمت‌های خاکستری فریم‌های تخصیص داده شده به یک پردازه هستند. اگر Internal Fragmentation برابر 900 بایت باشد، اندازه پردازه و External Fragmentation چند بایت است؟

(مهندسی کامپیوتر – دولتی ۹۹)



۱) اندازه پردازه برابر 12288 بایت و اندازه External Fragmentation برابر 14336 بایت است.
۲) اندازه پردازه برابر 12288 بایت و اندازه External Fragmentation برابر صفر است.
۳) اندازه پردازه برابر 11388 بایت و اندازه External Fragmentation برابر صفر است.
۴) اندازه پردازه برابر 6 بایت و اندازه External Fragmentation برابر7 بایت است.

۶- در یک سیستم صفحه‌بندی (Paging)، طول آدرس منطقی 19 بیت است. اگر تعداد صفحات موجود در فضای آدرس منطقی 129 صفحه باشد و قرار باشد به یک فضای آدرس فیزیکی 1 مگابایتی نگاشت صورت گیرد، هر مدخل (entry) از جدول صفحه (Page Table) باید چند بیت باشد؟ (بدون در نظر گرفتن valid یا invalid در page table) (مهندسی کامپیوتر – دولتی ۹۹)

۱) 8 ۲) 9 ۳) 17 ۴) 20

khalilifar.ir

۷- در یک سیستم زمان‌بندی MLFQ دو صف RR با برش زمانی 3 و ∞ وجود دارد. میانگین زمان بازگشت برای پردازش‌های زیر کدام است؟ (مهندسی کامپیوتر – دولتی ۹۹)

	Arrival time	CPU Burst1	I/O Burst1	CPU Burst2
P1	0	3	6	2
P2	3	7	5	1
P3	4	2	4	4
P4	11	5	3	1

۱) 21.5 ۲) 17 ۳) 16.5 ۴) 14

۸- یک سامانه مدیریت حافظه را در نظر بگیرید، که تنها چهار قاب به پردازه اختصاص داده شده است که در ابتدا خالی هستند. حال پردازه صفحات را به ترتیب از چپ به راست 1, 2, 3, 4, 3, 2, 2, 1, 6, 7 ارجاع می‌کند. اگر این سیستم مدیریت حافظه از روش LRU برای جایگزینی صفحه استفاده کند، چه تعداد نقص صفحه خواهیم داشت؟ (مهندسی کامپیوتر – دولتی ۹۹)

۱) 7 ۲) 8 ۳) 9 ۴) 10

۹- در یک دستگاه دیسک‌خوان، بازو روی سیلندر 35 قرار دارد و جهت حرکت آن به سمت شیارهای بزرگتر با شماره بزرگتر است. اگر زمان انتقال از یک سیلندر به بعدی 5ms باشد و از الگوریتم آسانسور برای دستیابی به سیلندرها استفاده شود، کل زمان جستجو برای دستیابی به سیلندرهای زیر (به ترتیب از چپ به راست) چند میلی ثانیه است؟ (مهندسی کامپیوتر – دولتی ۹۹)

34, 45, 39, 80, 12, 25, 44

۱) 565 ۲) 575 ۳) 665 ۴) 675

۱۰- کدام عبارت ذیل نمی‌تواند خروجی اجرای هم‌روند فرآیندهای P_1 و P_2 باشد؟ (S_1 متغیر سراسری با مقدار اولیه صفر می‌باشد.) (مهندسی کامپیوتر – دولتی ۹۹)

P1	P2
Print "A"	Print "C"
wait (S1)	signal (S1)
Print "B"	Print "D"

۱) ABCD ۲) ACDB ۳) CABD ۴) CADB

۱۱- کدام گزینه از مزایای ساختار سیستم عامل لایه‌ای (Layered) نسبت به ساختار سیستم عامل یکپارچه (Monolithic) نیست؟ (مهندسی IT – دولتی ۹۹)

۱) قابلیت گسترش بیشتر
۲) خطایابی ساده‌تر
۳) مدیریت ساده‌تر
۴) سرعت بیشتر

khalilifar.ir

۱۲- کدام مورد از مزایای ساختار ریزهسته (Micro Kernel) در طراحی سیستم عامل نیست؟
(مهندسی IT – دولتی ۹۹)

۱) کارایی سیستم را افزایش می‌دهد.
۲) برای سیستم‌های توزیع شده مناسب است.
۳) اضافه‌کردن سرویس جدید نیازی به اصلاح هسته سیستم عامل ندارد.
۴) در صورت بروز خرابی در سرویس خارج از هسته، کل سیستم عامل از کار نمی‌افتد.

۱۳- با توجه به جدول زیر، متوسط زمان پاسخ‌دهی (Response Time) و متوسط زمان انتظار (Waiting Time) پردازه‌ها برای الگوریتم Preemptive Shortest Remaining Job First چند واحد زمانی است؟
(مهندسی IT – دولتی ۹۹)

پردازه	زمان محاسبات	زمان ورود پردازه
P1	5	3
P2	8	1
P3	6	2

۱) متوسط زمان پاسخ‌دهی برابر 6 و متوسط زمان انتظار برابر صفر است.
۲) متوسط زمان پاسخ‌دهی برابر 6 و متوسط زمان انتظار برابر 1.6 است.
۳) متوسط زمان پاسخ‌دهی برابر 6 و متوسط زمان انتظار برابر 6 است.
۴) متوسط زمان پاسخ‌دهی برابر 6.6 و متوسط زمان انتظار برابر 6 است.

۱۴- در یک سیستم، 50 پردازه (Proccess) با کد زیر به صورت هم‌روند (Concurrent) در حال اجرا هستند. اگر مقدار اولیه سمافورها x=20 ، y=15 و z=1 باشند، حداکثر چند پردازه ممکن است پشت سمافور z در حالت انتظار بلوکه شوند؟
(مهندسی IT – دولتی ۹۹)

```
wait (x);
 wait (y);
   wait (z);
      a= a+1;
   signal (z);
 signal (y);
signal (x);
```

۱) 14 ۲) 30 ۳) 49 ۴) 50

۱۵- یک سامانه داری 64 صفحه مجازی (Virtual Pages) است که به 16 قاب فیزیکی (Physical Frames) بر اساس رابطه زیر نگاشت داده می‌شود. طول هر صفحه یک کیلو کلمه (1 K Words) است. اگر آدرس مجازی برابر 1010101000111101 باشد، کدام گزینه آدرس فیزیکی را نشان می‌دهد؟
(مهندسی IT – دولتی ۹۹)

khalilifar.ir

۱) 10101010001111 ۲) 10111000111101
۳) 10101000111101 ۴) 11001000111101

۱۶- کدام گزینه معیار (Criterion) یک زمان‌بند پردازنده نیست؟ (مهندسی IT – دولتی ۹۹)

۱) زمان پاسخ ۲) بهره‌وری پردازنده
۳) گذردهی (Throughput) ۴) زمان Burst (Burst time)

۱۷- برای خواندن از دیسک، در کدام لایه نرم‌افزاری محاسبات مربوط به شیار (Track)، قطاع (Sector) و هد دیسک صورت می‌پذیرد؟ (مهندسی IT – دولتی ۹۹)

۱) لایه Device Driver
۲) لایه روتین سرویس‌دهی به وقفه
۳) لایه مدیریت دستگاه‌های سیستم عامل
۴) لایه نخ سطح هسته که برنامه سطح کاربر را اجرا می‌کند.

khalilifar.ir

پاسخ تست‌های کنکور کارشناسی ارشد سال ۱۳۹۹

۱- گزینه (۴) صحیح است.

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

```
P0:
 {
(1) waiting [i]= TRUE;
(2) key = TRUE;
(3) while (waiting [i] && key)
(4) key = test_and_set (&lock);
(5)  waiting [i] = FALSE;
/*critical_section*/
(6) j = (i + 1) % n;
(7) while (!waiting [i])
(8) j = (j + 1) % n;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
}
```

```
P1:
 {
(1) waiting [i]= TRUE;
(2) key = TRUE;
(3) while (waiting [i] && key)
(4) key = test_and_set (&lock);
(5)  waiting [i] = FALSE;
/*critical_section*/
(6) j = (i + 1) % n;
(7) while (!waiting [i])
(8) j = (j + 1) % n;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
}
```

توجه: عملگر % که به mod معروف است، باقی مانده تقسیم را در خروجی قرار می‌دهد.

توجه: معادل عبارت j = (i + 1) % n در خط (6) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P0 به صورت زیر است:

```
j = (i + 1) % 2
j = (0 + 1) % 2
j = 1
```

توجه: معادل عبارت j = (j + 1) % n در خط (8) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P0 به صورت زیر است:

```
j = (j + 1) % 2
j = (1 + 1) % 2
j = 0
```

توجه: معادل عبارت j = (i + 1) % n در خط (6) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P1 به صورت زیر است:

```
j = (i + 1) % 2
j = (1 + 1) % 2
j = 0
```

khalilifar.ir

توجه: معادل عبارت j = (j + 1) % n در خط (8) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P1 به صورت زیر است:

j = (j + 1) % 2
j = (0 + 1) % 2
j = 1

توجه: مقادیر اولیه به صورت زیر است:

key =FALSE , lock = FALSE, waiting [0] = FALSE, waiting [1] = FALSE

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

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

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

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

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

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

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

P_0:
(1) waiting [0]= TRUE;
(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

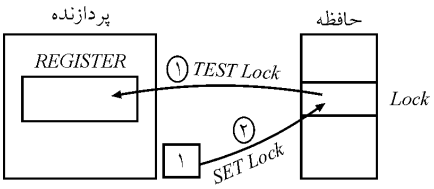
(4) key = test_and_set (&lock);

توجه: بسیاری از پردازنده‌ها، دستورالعمل دو بخشی اما اتمیک خاصی دارند به نام TSL (TEST and SET Lock)، بدین نحو که این دستورالعمل به شکل تجزیه ناپذیر، عملیات زیر را انجام می‌دهد:

khalilifar.ir

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

به شکل زیر توجه کنید:



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

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:
(5) waiting [0] = FALSE;
/*critical_section*/

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

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

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

khalilifar.ir

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

P_1:

(1) waiting [1]= TRUE;

(2) key = TRUE;

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

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

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

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

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

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

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

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

khalilifar.ir

```
P0:
 {
(1) waiting [0]= TRUE;
(2) key = TRUE;
(3) while (waiting [0] && key)
(4) key = test_and_set (&lock);
(5)  waiting [0] = FALSE;
/*critical_section*/
(6) j = 1;
(7) while (!waiting [1])
(8) j = 0;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
}
```

```
P1:
 {
(1) waiting [1]= TRUE;
(2) key = TRUE;
(3) while (waiting [1] && key)
(4) key = test_and_set (&lock);
(5)  waiting [1] = FALSE;
/*critical_section*/
(6) j = 0;
(7) while (!waiting [0])
(8) j = 1;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
}
```

توجه: مقادیر اولیه به صورت زیر است:

key =FALSE , lock = FALSE, waiting [0] = FALSE, waiting [1] = FALSE

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

```
P0:
(1) waiting [0]= TRUE;
(2) key = TRUE;
```

همچنین فرض کنید فرآیند P_1 نیز به شکل همروند یا موازی با فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
(1) waiting [1]= TRUE;
(2) key = TRUE;
```

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

دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند. مقدار اولیه متغیر lock برابر FALSE است، هر فرآیندی که زودتر دستور key = test_and_set (&lock) را اجرا کند، داخل ناحیه بحرانی می‌شود و فرآیندی که دیرتر دستور key = test_and_set (&lock) را اجرا کند، باید شرط حلقه را چک کند.

khalilifar.ir

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به اجرای دستور test_and_set (&lock) کنند، بنابراین فرآیند P_0 وارد ناحیه بحرانی می‌شود، اما فرآیند P_1 باید شرط حلقه را چک کند. به صورت زیر:

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:
(5) waiting [0] = FALSE;
/*critical_section*/

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

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

توجه: در اجرای هم‌روند قبلا از دو خط اول فرآیند P1 عبور کردیم و key = TRUE قبلا اجرا شده است و مجدد اجرا نمی‌شود، پس در حال حاضر مقدار key برابر FALSE است که قبلا در فرآیند P0 مقداردهی شده است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P1:
(5) waiting [1] = FALSE;
/*critical_section*/

khalilifar.ir

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفق شدند. فرآیند اول و دوم نتوانستند هر دو باهم وارد ناحیه بحرانی خودشان بشوند. بنابراین شرط دوم انحصار متقابل برقرار نیست.

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

قانون سوم ارسطو (آزمون شرط پیشرفت)

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

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

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

P_0:

(1) waiting [0]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

khalilifar.ir

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:
(5) waiting [0] = FALSE;
/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P0 مشغول حرکت است.

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

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

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

P0:
/*critical_section*/
(6) j = 1;
(7) while (!waiting [1])

توجه: هم اکنون خروجی حلقه while (!FALSE) برابر TRUE است. پس بدنه حلقه یعنی دستور j = 0 اجرا می‌شود.

(8) j = 0;
(9) if (j = = i)

توجه: هم اکنون خروجی دستور شرطی if (0 = = 0) برابر TRUE است. پس بدنه دستور شرطی یعنی دستور lock = FALSE اجرا می‌شود.

(10) lock = FALSE;

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

P0:
/*remainder_section*/

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

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

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

khalilifar.ir

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

P_1:

(1) waiting [1]= TRUE;

(2) key = TRUE;

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

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

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

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P1:

(5) waiting [1] = FALSE;

/*critical_section*/

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

توجه: هم اکنون waiting [0] = FALSE و key = FALSE و lock = TRUE است.

(گام ۴): همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

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

P1:

/*critical_section*/

(6) j = 0;

(7) while (!waiting [0])

توجه: هم اکنون خروجی حلقه while (!FALSE) برابر TRUE است. پس بدنه حلقه یعنی دستور j = 1 اجرا می‌شود.

khalilifar.ir

(8) j = 1;
(9) if (j = = i)

توجه: هم اکنون خروجی دستور شرطی if (1 = = 1) برابر TRUE است. پس بدنه دستور شرطی یعنی دستور lock = FALSE اجرا می‌شود.

(10) lock = FALSE;

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

P1:
/*remainder_section*/

همانطور که در (گام ۴) گفتیم قرار شد که همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۵) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P1 مشغول حرکت است.

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

فرض کنید فرآیند P1 نیز قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:
(1) waiting [1]= TRUE;
(2) key = TRUE;

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

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

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

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

khalilifar.ir

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P1:
(5) waiting [1] = FALSE;
/*critical_section*/

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

قانون چهارم ارسطو (آزمون گرسنگی)

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

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

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

P_0:
(1) waiting [0]= TRUE;
(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

khalilifar.ir

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

P0:
(5) waiting [0] = FALSE;
/*critical_section*/

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

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

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

P_1:
(1) waiting [1]= TRUE;
(2) key = TRUE;

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

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنـه حلقـه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

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

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنـه حلقـه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

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

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قـرار بـدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۳) می‌شویم. هم اکنون پردازنده پشت ناحیه بحرانی فرآیند P1 در یک حلقه انتظار، دچار انتظار مشغول است.

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

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

khalilifar.ir

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

P0:

/*critical_section*/

(6) j = 1;

(7) while (!waiting [1])

توجه: هم اکنون خروجی حلقه while (!TRUE) برابر FALSE است. پس بدنه حلقه یعنی دستور j = 0 اجرا نمی‌شود.

(8) j = 0;

(9) if (j = = i)

توجه: هم اکنون خروجی دستور شرطی if (1 = = 0) برابر FALSE است. پس بدنه دستور شرطی یعنی دستور lock = FALSE اجرا نمی‌شود.

(10) lock = FALSE;

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

P0:

/*remainder_section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۴) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

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

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

P_0:

(1) waiting [0]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = TRUE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = TRUE است.

khalilifar.ir

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P0 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P0 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۴) گفتیم قرار شد که فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب نشد، فرآیند اول نتوانست مجددا وارد ناحیه بحرانی خودش بشود، دقت کنید که در سناریوی فوق فرآیند P_0 از ناحیه بحرانی خارج شد، اما مقدار lock را نتوانست FALSE کند و مقدار lock در همان مقدار TRUE باقی ماند. فرآیند P1 از قبل در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود مانده است و می‌چرخد و در ادامه فرآیند P0 هم در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود مانده است و این یعنی «**بن بست**» رخ داده است.

اگر فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند که در این حالت بن‌بست رخ داده است. خب هر دو باهم پشت ناحیه بحرانی خودشان مسدود شدند. فرآیند اول نتوانست وارد ناحیه بحرانی خودش شود، همچنین فرآیند دوم هم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین «**بن بست**» رخ داده است.

قانون دوم ارسطو(آزمون بن بست)

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

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور

khalilifar.ir

همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خـواب رفته‌اند که در این حالت بن‌بست رخ داده است.

```
P0:                                   P1:
 {                                     {
(1) waiting [0]= TRUE;                (1) waiting [1]= TRUE;
(2) key = TRUE;                       (2) key = TRUE;
(3) while (waiting [0] && key)        (3) while (waiting [1] && key)
(4) key = test_and_set (&lock);       (4) key = test_and_set (&lock);
(5)  waiting [0] = FALSE;             (5)  waiting [1] = FALSE;
/*critical_section*/                  /*critical_section*/
(6) j = 1;                            (6) j = 0;
(7) while (!waiting [1])              (7) while (!waiting [0])
(8) j = 0;                            (8) j = 1;
(9) if (j == i)                       (9) if (j == i)
(10) lock = FALSE;                    (10) lock = FALSE;
/*remainder_section*/                 /*remainder_section*/
}                                     }
```

توجه: مقادیر اولیه به صورت زیر است:

key =FALSE , lock = FALSE, waiting [0] = FALSE, waiting [1] = FALSE

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

```
P0:
(1) waiting [0]= TRUE;
(2) key = TRUE;
```

همچنین فرض کنید فرآیند P_1 نیز به شکل همروند یا موازی با فرآینـد P_0 قصـد دارد وارد ناحیـه بحرانی خودش شود، به صورت زیر:

```
P2:
(1) waiting [1]= TRUE;
(2) key = TRUE;
```

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود بـه ناحیـه‌ی بحرانـی را دارنـد. بنـابراین آرایه فرآیندها هر دو waiting [0]= TRUE و waiting [1]= TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند.

دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قـرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند. مقدار اولیـه متغیـر lock برابـر FALSE است، هر فرآیندی کـه زودتـر دسـتور key = test_and_set (&lock) را اجـرا کنـد، داخـل ناحیـه

khalilifar.ir

بحرانی می‌شود و فرآیندی که دیرتر دستور key = test_and_set (&lock) را اجرا کند، باید شرط حلقه را چک کند.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به اجرای دستور test_and_set (&lock) کنند، بنابراین فرآیند P_0 وارد ناحیه بحرانی می‌شود، اما فرآیند P_1 باید شرط حلقه را چک کند. به صورت زیر:

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:

(5) waiting [0] = FALSE;

/*critical_section*/

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

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

توجه: در اجرای هم‌روند قبلا از دو خط اول فرآیند P1 عبور کردیم و key = TRUE قبلا اجرا شده است و مجدد اجرا نمی‌شود پس در حال حاضر مقدار key برابر FALSE است که قبلا در فرآیند P0 مقداردهی شده است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

khalilifar.ir

P1:
(5) waiting [1] = FALSE;
/*critical_section*/

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

توجه: برای برقرار بودن شرط انتظار محدود باید **قانون دوم ارسطو** (آزمون بن بست) و **قانون چهارم ارسطو**(آزمون گرسنگی) هر دو باهم برقرار باشند. بنابراین شرط انتظار محدود در سوال مطرح شده به دلیل وجود بن بست در سناریوی گرسنگی برقرار نیست.

صورت سوال به این شکل است:

الگوریتم زیر ساختار فرآیند P_i برای حل مسئله ناحیه بحرانی (Critical-Problem) در حالتی که n فرآیند وجود داشته باشد، است. در خصوص این الگوریتم کدام گزینه صحصیح است؟

۱) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را به ازای مقدار دلخواه n برآورده می‌کند.

گزینه اول **نادرست** است، زیرا شرط انحصار متقابل برقرار نیست و برآورده نمی‌کند، شرط پیشرفت برقرار است و برآورده می‌کند و شرط انتظار محدود برقرار نیست و برآورده نمی‌کند.

۲) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را تنها به ازای مقدار n=2 برآورده می‌کند.

گزینه دوم **نادرست** است، زیرا شرط انحصار متقابل برقرار نیست و برآورده نمی‌کند، شرط پیشرفت برقرار است و برآورده می‌کند و شرط انتظار محدود برقرار نیست و برآورده نمی‌کند.

۳) شرط پیشرفت را تنها به ازای مقدار n بزرگتر از 2 برآورده نمی‌کند.

گزینه سوم **نادرست** است، زیرا شرط پیشرفت برقرار است و برآورده می‌کند.

۴) شرط پیشرفت را هرگز برآورده نمی‌کند.

گزینه چهارم **نادرست** است، زیرا شرط پیشرفت برقرار است و برآورده می‌کند.

توجه: همانطور که واضح و مشخص هست، همه گزینه‌ها نادرست هستند و هیچ یک از گزینه‌ها نمی‌تواند پاسخ سوال باشد.

توجه: هنگامی که در راه حلی برای کنترل شرایط رقابتی شرط انحصار متقابل برقرار نیست و برآورده نمی‌کند و همچنین بن بست رخ داده است یعنی شرط انتظار محدود هم برقرار نیست و برآورده نمی‌کند، بررسی شرط پیشرفت برای داشتن یک راه حل اخلاقی برای کنترل شرایط رقابتی

khalilifar.ir

نیاز نیست.
توجه: سازمان سنجش آموزش کشور، در کلید اولیه و نهایی خود، گزینه چهارم را به عنوان پاسخ اعلام کرده بود. ولی همانطور که دیدید پاسخ سوال در گزینه‌ها نبود.

۲- گزینه (۱) صحیح است.

مقابله با بن‌بست

در حالت کلی برای مقابله با بن‌بست به چهار روش می‌توان عمل کرد:
۱- الگوریتم Ostrich (شتر مرغ)
۲- تشخیص و ترمیم بن‌بست (Deadlock Detection and Recovery)
۳- پیشگیری (جلوگیری) از بن‌بست (Deadlock Prevention)
۴- اجتناب (دوری) از بن‌بست (Deadlock Avoidance)

الگوریتم Ostrich

در این روش، سیستم عامل وقوع بن‌بست را کاملاً نادیده می‌گیرد!

تشخیص و ترمیم بن‌بست

در این روش هیچ هزینه‌ای برای پیشگیری و اجتناب از بن‌بست پرداخت نمی‌شود و اجازه می‌دهیم بن‌بست رخ دهد. پس از کشف بن‌بست، باید سیستم از بن‌بست خارج شود.
توجه: الگوریتم کافمن (Coffman)، جهت کشف بن بست مورد استفاده قرار می‌گیرد.

پیشگیری از بن‌بست

برای وقوع بن‌بست باید هر چهار شرط کافمن برقرار باشند. برای پیشگیری از بن‌بست باید کاری کنیم که یکی از این شروط رخ ندهد. بنابراین هر چهار شرط را بررسی می‌کنیم:
الف- انحصار متقابل
ب- انحصاری بودن
ج- نگهداری و انتظار
د- انتظار چرخشی

اجتناب از بن‌بست

تا اینجا برای مقابله با بن‌بست یا از روش تشخیص و بازیابی استفاده می‌کردیم و یا از روش پیشگیری. روش تشخیص و بازیابی این نقص را دارد که هنگام بازیابی، تعدادی از فرآیندها باید به عقب برگردند و سربار تحمیل شده به سیستم بسیار بالاست. روش پیشگیری از منابع استفاده‌ی بهینه نمی‌کند و در کل بهره‌وری سیستم را کاهش می‌دهد.
اما ایده‌ی کلی موجود در روش اجتناب از بن‌بست این است که هنگامی که یک فرآیند منبعی را

khalilifar.ir

درخواست کرد، حتی اگر منبع مورد نیاز وی کاملاً آزاد است، فـوراً آن منبـع را در اختیـار فرآینـد قرار ندهیم، بلکه ابتدا بررسی کنیم با تخصیص دادن این منبع به فرآیند، احتمال وقوع بن‌بسـت در آینده وجود دارد یا خیر. در صورتی که احتمال وقوع بن‌بست در آینده وجود داشته باشـد، فرآینـد مورد نظر باید منتظر بماند و منبع مورد نیاز وی، فعلاً به وی تخصیص داده نمی‌شود.

الگوریتم بانکدار جهت اجتناب (دوری) از بن‌بست (Bankers Algorithm)

این الگوریتم جهت اجتناب از بن‌بست به کار می‌رود و همانند یک بانکدار عمل می‌کند که هیچ‌گاه همه‌ی سرمایه‌ی خود را به مشتریانش تخصیص نمی‌دهد و همواره طوری عمل می‌کند کـه بتوانـد نیازهای مشتریانش را برآورده کند.

توجه: هنگامی که مشخص نباشد کـه رفتـار فرآینـدها نگهـداری و درخواسـت و یـا رهاسـازی و درخواست است، و ماتریس Need مورد استفاده قرار گیرد، در این شرایط برای اجتناب (دوری) از بن بست، الگوریتم بانکداران (Bankers) مورد استفاده قرار می‌گیرد.

توجه: هنگامی که رفتار فرآیندها نگهداری و درخواست باشد، و ماتریس Request مـورد اسـتفاده قرار گیرد، در این شرایط برای کشف بن‌بست، الگوریتم کشف بن‌بست کـافمن (Coffman) مـورد استفاده قرار می‌گیرد.

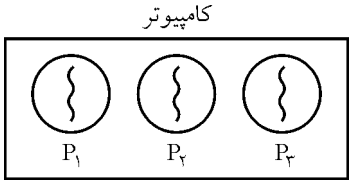
توجه: الگوریتم بانکداران، جهت بررسی امن یا ناامن بودن مورد استفاده قرار می‌گیـرد و در مـورد بن‌بست نظر نمی‌دهد.

توجه: الگوریتم کافمن (Coffman)، جهت کشف بن‌بست مورد استفاده قرار می‌گیرد.

۳- گزینه (۳) صحیح است.

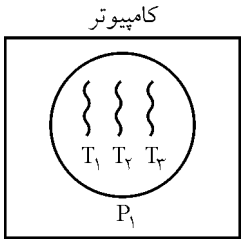
در سیستم‌های قدیمی‌تر، به ازای هر فرآیند یک رشته نخ یا رشته اجرایی و به تبـع یـک شـمارنده برنامه (PC) وجود داشت اما در سیستم عامل‌های امروزی به ازای هر فرآیند می‌توان چنـد نـخ یـا رشته اجرایی داشت.

شکل زیر سه فرآیند معمولی را نشان می‌دهد که هریک برای خودشان یک رشـته اجرایـی و یـک حافظه مختص به خود را دارند.

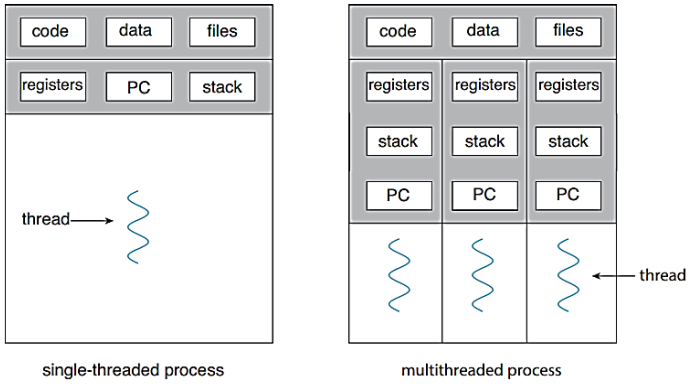


ولی در شکل زیر یک فرآیند، سه رشته اجرایی دارد که هر یک رجیستر، پشـته و شـمارنده برنامـه (PC) مجزای خود را دارند و مانند فرآیندها می‌توانند همروند (در سیستم‌های تک‌پردازنده‌ای) و موازی (در سیستم‌های چندپردازنده‌ای) اجرا شوند.

khalilifar.ir



نکته: نخ‌های همتا که در یک فرآیند قرار دارند و از کد، داده و منابع مشترک استفاده می‌کنند اما هر نخ، شمارنده برنامه، مجموعه رجیستر و فضای پشته جداگانه‌ای در اختیار دارند. در واقع هر نخ، TCB مجزایی دارد.



توجه: از آن‌جا که نخ‌های همتا در یک فرآیند قرار داشته و اشتراکات زیادی با هم دارند، عمل تعویض متن بین آنها به راحتی و با هزینه کمتری صورت می‌گیرد، در واقع TCB مربوط به نخ‌ها، محتوی کمتری نسبت به PCB فرآیندها دارد، مثلاً لیست فایل‌های باز مربوط به فرآیندها است، بنابراین این لیست به هنگام تعویض متن فرآیندها باید داخل PCB مربوط به فرآیند ذخیره گردد، در حالی که به هنگام تعویض متن بین نخ‌ها نیازی به ذخیره‌سازی لیست فایل‌های باز مربوط به یک فرآیند در TCB یک نخ نیست. **بنابراین تعویض متن بین نخ‌ها نسبت به فرآیندها ارزان‌تر است.**

توجه: گاهی از نخ به عنوان Light Weight Process (فرآیند سبک وزن) نیز یاد می‌کنند و به کل یک فرآیند، Heavy Weight Process (فرآیند سنگین وزن) نیز گویند.

توجه: نخ‌ها هم مانند فرآیندها می‌توانند حالت‌های مختلفی را تجربه کنند مانند آماده، در حال اجرا یا منتظر. در واقع پردازنده می‌تواند بین نخ‌ها به اشتراک گذاشته شود.

مزایای فرآیندهای چندنخی

۱- ساختار بسیاری از برنامه‌های کاربری ذاتاً از بخش‌های کاملاً مستقل تشکیل می‌شوند که جدا

khalilifar.ir

نکردن آن‌ها باعث پیچیدگی بالا و کاهش خوانایی در برنامه می‌گردد. مهندسی نرم‌افزار نیز بر ساخت برنامه‌های کاربردی توسط پیمانه‌های مستقل نیز تأکید دارد. برای مثال، خطاست اگر بیندیشید که برنامه شبیه‌سازی ۱۱ بازیکن یک تیم فوتبال در یک نخ قرار گیرد. به عنوان مثالی دیگر یک برنامه واژه‌پرداز می‌تواند از نخ‌های مستقلی مانند کنترل املا و گرامر، صف‌آرایی، مدیریت ورودی‌های کاربر و غیره تشکیل شده باشد.

۲- در فرآیند تک‌نخی، هرگاه فراخوان سیستمی مسدود کننده‌ای اجرا شود، کل فرآیند مسدود می‌گردد. در حالی‌که در فرآیندهای چندنخی در صورتی که سیستم عامل زمان‌بندی چند نخی را پشتیبانی کند، فقط نخی که فراخوان سیستمی مسدودکننده دارد، مسدود می‌گردد و مابقی نخ‌های یک فرآیند می‌توانند پس از در اختیار گرفتن پردازنده، اجرا گردند. فرآیند تک‌نخی مانند قانونی می‌باشد که اگر یک نفر در خانواده خطا کند، همه خانواده محکوم می‌گردند و فرآیند چندنخی مانند قانونی می‌باشد که اگر یک نفر در خانواده خطا کند، فقط همان یک نفر محکوم می‌گردد و بقیه خانواده می‌توانند به زندگی طبیعی خود ادامه دهند.

۳- ایجاد هم‌روندی (در سیستم تک‌پردازنده‌ای) و توازی (در سیستم چندپردازنده‌ای) در نخ‌های یک فرآیند و فرآیندهای دیگر.

مثال: کاربرد چندنخی در فرآیند سمت سرویس‌دهنده.

در این مدل، فرآیند سرویس‌دهنده از چندین نخ جهت پاسخ به درخواست‌های کاربر یعنی سرویس‌گیرنده تشکیل شده است. پاسخ هر کاربر می‌تواند توسط یک نخ از سمت سرویس‌دهنده داده شود. چنانچه نخی در فرآیند سرویس‌دهنده جهت تبادل داده از روی دیسک به سمت سرویس‌گیرنده مسدود گردد، نخ‌های دیگر فرآیند سرویس‌دهنده می‌توانند به درخواست‌های دیگر، پاسخ دهند. زیرا کارکرد آن‌ها وابسته به نخ مسدود شده نیست.

توجه: شاید بگویید به جای قرار دادن کارهای مختلف یک سرویس‌دهنده در داخل نخ‌های یک فرآیند، می‌شد هریک از کارها را در داخل یک فرآیند قرار داد و چند فرآیندی را در مقابل چندنخی ابداع کرد. اما به دلایل زیر استفاده از چندنخی معقولانه‌تر به‌نظر می‌رسد:

- هزینه زمانی ایجاد (بارگذاری TCB) و پایان دادن (ذخیره‌سازی TCB) به یک نخ در یک فرآیند به مراتب کمتر از ایجاد (بارگذاری PCB) و پایان دادن (ذخیره‌سازی PCB) یک فرآیند است. نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند، در حالی که فرآیندها، منابع مختص به خود را در اختیار می‌گیرند.
- نخ‌های همتا در یک فرآیند، اشتراکات زیادی باهم دارند، بنابراین عمل تعویض متن بین آنها با هزینه کمتری انجام می‌گردد. در حالی که تعویض متن بین فرآیندها به دلیل عدم اشتراکات با هزینه بیشتری انجام می‌گردد.

توجه: اشتراکات نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس،

khalilifar.ir

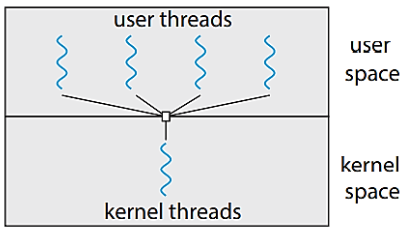
فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد. از آنجاکه نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند. بنابراین ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس مشترک هم دسترسی داشته و می‌توانند در فضای آدرس مشترک هم بنویسند.

اداره نخ‌ها توسط بسته نخ (Thread Package) به سه روش زیر وجود دارد:

نخ‌های سطح کاربر: در این روش اداره نخ شامل ایجاد، حذف و زمان‌بندی نخ در مد کاربر و توسط کتابخانه نخ (Thread Library) انجام می‌شود. در این روش هسته سیستم عامل از وجود نخ اطلاع ندارد. در این روش فقط زمان‌بند پردازنده و زمان‌بند چندنخی در سطح کاربر وجود دارد و زمان‌بند چند نخی در سطح هسته در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل فقط فرآیندها را می‌شناسد و هیچ اطلاعاتی از نخ‌ها ندارد. در واقع اولویت‌بندی نخ‌ها، مدیریت نخ‌ها و زمان‌بند چندنخی در سطح کاربر و توسط یک بسته نرم‌افزاری انجام می‌گردد. بدین معنی که نخ‌ها را برنامه‌نویس مشخص می‌کند و مدیریت آن‌ها را نیز بر عهده می‌گیرد. بنابراین زمان‌بند پردازنده، براساس الگوریتم مشخصی مثلاً نوبت چرخشی پردازنده را در اختیار یکی از فرآیندهای آماده قرار می‌دهد، سپس زمان‌بند چند نخی در سطح کاربر، متناسب با کاربردی که در آن فرآیند به کار گرفته می‌شود، الگوریتم زمان‌بندی را انتخاب کرده و تصمیم می‌گیرد که پردازنده در اختیار کدام‌یک از نخ‌های آماده در فرآیند موردنظر قرار گیرد و تا زمانی که پردازنده در تملک فرآیند باشد و یا تا قبل از پایان برش زمانی مربوط به فرآیند، نخ‌های یک فرآیند از پردازنده بهره‌مند می‌شوند و به محض مسدود شدن یک نخ و یا پایان برش زمانی یک فرآیند، یا اتمام فرآیند، پردازنده به فرآیند بعدی تعلق می‌گیرد.

توجه: در این روش نخ ماهیت منطقی دارد و از دید کاربر فقط وجود دارد، در واقع از نظر سیستم عامل ماهیت فیزیکی ندارد، بنابراین نخ کاربر در این روش همانند یک تابع در فرآیند می‌باشد که از رجیستر و پشته مختص به خود نیز بهره‌مند می‌باشد.

توجه: در این روش چند نخ سطح کاربر به یک فرآیند تک نخی نگاشت می‌شود که به آن مدل چند به یک (Many to one model) گفته می‌شود. شکل زیر گویای مطلب است:



Many-to-one model.

khalilifar.ir

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، بین پدران خانواده‌ها تقسیم گردد و این پدران خانواده‌ها باشند که تصمیم بگیرند به هر عضو خانواده چه مقدار نقدینگی تعلق بگیرد. در این روش فقط پدران شماره حساب مختص به خود را دارند. اما اگر یکی از اعضای خانواده خطایی انجام دهد و محکوم گردد، آنکه تمام اعضای خانواده برای مدتی از خدمات دولت محکوم می‌گردند، زیرا در این مدل، دولت فقط پدران خانواده‌ها را می‌شناسد و از اعضای خانواده اطلاعی ندارد. بنابراین حساب پدر خانواده برای مدتی مسدود می‌گردد.

توجه: عمل تعویض متن مابین نخ‌های یک فرآیند کاربر، کاملاً در سطح کاربر و با سربار بسیار ناچیز (در حد فراخوانی نخ بعدی) و بدون تغییر حالت پردازنده به مد هسته پردازنده، توسط زمان‌بند چندنخی در سطح کاربر (برنامه‌های کاربر) انجام می‌گردد. فقط کافیست محتوای ثبات‌های CPU برای نخ فعلی، ذخیره شده و سپس مقادیر قبلا ذخیره شده مربوط به نخ بعدی که به آن سوییچ می‌شود، بارگذاری مجدد شوند. و نیاز به دخالت هسته و تله‌های سنگین (سویچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مد هسته به مد کاربر شامل تغییر MMU و TLB) ندارد.

توجه: نخ‌های سطح کاربر، رابطه و تبادل داده سریع‌تر و سبک‌تری نسبت به نخ‌های سطح هسته دارند. چون نیاز به به دخالت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مد هسته به مد کاربر شامل تغییر MMU و TLB) ندارد.

توجه: به دلیل آنکه نخ‌های یک فرآیند کاربر، تماماً در فضای کاربر مدیریت می‌شوند، اگر نخ موجود در یک فرآیند کاربر، یک فرآیند سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل نه تنها آن نخ، بلکه کل فرآیند کاربر را که شامل تمام نخ‌های دیگر می‌باشد، مسدود می‌کند، زیرا هسته سیستم عامل خبری از نخ‌های داخل فرآیند کاربر ندارد. در واقع در این روش هسته سیستم عامل نخ‌ها را همانند توابع داخل یک فرآیند می‌بیند، یعنی هسته سیستم عامل، یک فرآیند چند نخی سطح کاربر را، مانند یک فرآیند تک نخی اما دارای چند تابع مختلف می‌بیند!

توجه: یک نقص صفحه (صفحه‌بندی در حافظه مجازی) مربوط به یک نخ، باعث می‌شود کل فرآیند مسدود شود. زیرا هسته سیستم عامل خبری از نخ‌های داخل فرآیند کاربر ندارد.

توجه: زمان‌بند سطح کاربر برای زمان‌بندی نخ‌ها نمی‌تواند از Timer سیستم استفاده نماید و باید به طور دستی زمان‌بندی شود.

توجه: این راه‌کار، منجر به عدم امکان هم‌روندی (در سیستم‌های تک‌پردازنده‌ای) نخ‌های داخل یک فرآیند کاربر در حالت انسداد یک نخ داخل یک فرآیند کاربر می‌گردد.

البته اگر نخی داخل یک فرآیند کاربر مسدود نگردد، امکان هم‌روندی میان نخ‌های داخل فرآیند

khalilifar.ir

کاربر برقرار است. مانند یک تیم فوتبال ۱۱ نفره که اگر بازیکنی مرتکب خطا گردد، از آن‌جا که داور فقط نام تیم را می‌شناسد و نه تک‌تک بازیکنان تیم را، آن‌گاه کل تیم را جریمه، اخراج و مسدود می‌کند. اما اگر هیچ‌یک از بازیکنان تیم مرتکب خطایی نگردد، واضح است که هم‌روندی برقرار است.

توجه: فرض کنید یک کیک داریم که آن را به چهار قسمت مساوی تقسیم کرده‌ایم، همچنین فرض کنید خوردن هر بخش کیک یک ساعت زمان بخواهد، اگر یک نفر بخواهد تمام این کیک را بخورد، پس چهار ساعت طول می‌کشد اگر چهار نفر بخواهند تمام این کیک را بخورند و به هر نفر یک بخش کیک داده شود، آنگاه خوردن تمام کیک به‌طور موازی ۱ ساعت طول خواهد کشید.

در روش سطح کاربر، یک فرآیند چند نخی کاربر نمی‌تواند از امتیازات چند پردازنده‌ای بهره ببرد، زیرا در روش سطح کاربر، هسته سیستم عامل در هر لحظه فقط یک پردازنده را در اختیار نخ‌های یک فرآیند کاربر قرار می‌دهد. حتی اگر چند پردازنده موجود باشد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر نیست. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند کاربر وجود ندارد.

توجه: نرم‌افزارهای POSIX P-threads و Mach C-thread یه عنوان یک بسته نرم‌افزاری، می‌تواند جهت مدیریت نخ‌ها و زمان‌بند چند نخی در سطح کاربر مورد استفاده قرار گیرند.

توجه: در این روش تخصیص منابع و زمان‌بندی پردازنده، بر روی فرآیندها انجام می‌شود. همچنین زمان‌بندی نخ‌های سطح کاربر، بر عهده زمان‌بند چندنخی سطح کاربر خواهد بود.

توجه: سیستم عامل سولاریس، مدل چند به یک را پیاده سازی می‌کند.

نخ‌های سطح هسته: در این روش اداره نخ شامل ایجاد، حذف و زمان‌بندی نخ در مد هسته و توسط هسته سیستم عامل انجام می‌شود. در این روش هسته از وجود نخ اطلاع دارد. در این روش فقط زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته وجود دارد و زمان‌بند چند نخی در سطح کاربر در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل نه تنها فرآیندها را می‌شناسد بلکه از وجود نخ‌های داخل یک فرآیند نیز در صورت وجود نخ آگاه است. در واقع مدیریت نخ‌ها و زمان‌بند چندنخی (نخ‌های فرآیندهای کاربر و نخ‌های فرآیندهای سیستم عامل) در سطح هسته، توسط هسته سیستم عامل انجام می‌گردد و کاربر هیچ دیدی از این کار ندارد. انگار که اجتماع تمام نخ‌های فرآیندهای سیستم عامل و کاربر را در نظر بگیرید، حال بر روی تک نخ‌ها بر اساس یک الگوریتم خاص حرکت کنید. بنابراین زمان‌بند چند نخی در سطح هسته، براساس الگوریتم مشخصی مثلاً نوبت چرخشی پردازنده را در اختیار یکی از نخ‌های آماده قرار می‌دهد. توجه کنید در این روش پردازنده دیگر در تملک فرآیند نیست بلکه در تملک نخ‌ها است. در واقع تا زمانی که پردازنده در تملک یک نخ باشد و تا قبل از مسدود شدن نخ، و یا تمام شدن نخ و یا پایان برش زمانی مربوط به نخ، نخ می‌تواند از پردازنده بهره ببرد و به محض مسدود شدن یک نخ،

khalilifar.ir

یا پایان برشی زمان مربوط به نخ یا اتمام نخ، پردازنده می‌تواند به نخ بعدی که ممکن است، نخ بعدی، نخی هم‌خانواده با نخ قبلی در یک فرآیند باشد، یا نخی در یک فرآیند دیگر باشد، تعلق بگیرد. در واقع زمان‌بند چند نخی در سطح هسته سیستم عامل تعیین می‌کندکه نخ بعدی که باید شروع به کار کند متعلق به همان فرآیند باشد و یا از یک فرآیند دیگر انتخاب شود.

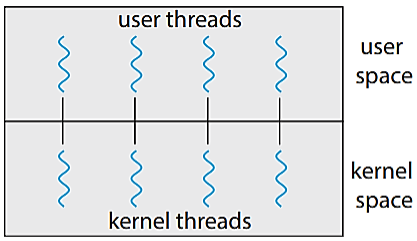
توجه: برای انجام زمان‌بندی چند نخی، هسته سیستم عامل باید علاوه بر جدول فرآیندها، یک جدول نخ (شبیه جدول فرآیند) داشته باشد که اطلاعات تمامی نخ‌های موجود در سیستم را نگهداری کند. مجدداً تأکید می‌کنیم که در این حالت هر نخ TCB خاص خود را دارد. به عبارت دیگر هر نخ رجیستر و پشته مختص به خود را دارد.

توجه: در این روش نخ ماهیت فیزیکی دارد، و از دید کاربر و سیستم عامل وجود دارد، بنابراین در این روش، هر نخ، رجیستر و پشته مختص به خود را دارد، به عبارت دیگر هر نخ TCB مختص به خود را دارد.

توجه: عملیات مرتبط با نخ‌های سطح کاربر، مانند ایجاد (بارگذاری TCB مختص به نخ) و پایان دادن (ذخیره‌سازی TCB مختص به نخ) بر عهده هسته سیستم عامل است و توسط زمان‌بند چند نخی در سطح هسته سیستم عامل انجام می‌گردد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، بدون اعمال هیچ‌گونه اولویت‌بندی بین تک تک اعضاء یک کشور تقسیم گردد (حال این اعضاء شهروند عام باشد یا خاص) در این روش هر یک از اعضای کشور شماره حساب مختص به خود را دارند. اما اگر یکی از افراد کشور خطایی انجام دهد و محکوم گردد، آنگاه فقط همان فرد برای مدتی از خدمات دولت محروم می‌گردد و دولت به بقیه اعضاء خانواده آن فرد همچنان خدمات ارائه می‌دهد. زیرا دولت از مشخصات تک تک اعضاء جامعه آگاه است.

توجه: در این روش هر نخ سطح کاربر به یک نخ سطح هسته نگاشت می‌شود که به آن مدل یک به یک (one to one model) گفته می‌شود. شکل زیر گویای مطلب است:



One-to-one model.

توجه: عمل تعویض متن، مابین نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل، در مد هسته سیستم پردازنده توسط زمان‌بند چندنخی در سطح هسته

khalilifar.ir

سیستم عامل انجام می‌گیرد. بنابراین زمان‌بندی نخ‌های سطح هسته به دلیل دخالت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مد هسته به مد کاربر شامل تغییر MMU و TLB)، کندتر از ریسه‌های سطح کاربر است. در نتیجه نخ‌های هسته به طور کلی فرایند ایجاد، حذف و مدیریت کُندتری نسبت به نخ‌های کاربر دارند.

توجه: نخ‌های سطح هسته رابطه و تبادل داده کندتر و سنگین‌تری نسبت به نخ‌های سطح کاربر دارند. چون نیاز به به دخالت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مد هسته به مد کاربر شامل تغییر MMU و TLB) دارد.

توجه: به دلیل آنکه نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل)، تماماً در فضای هسته سیستم عامل مدیریت می‌شوند و هسته سیستم عامل از وجود نخ‌های یک فرآیند آگاه است، اگر نخ موجود در یک فرآیند، یک فراخوان سیستمی مسدود کننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم‌خانواده با آن نخ مسدود شده، همچنان می‌توانند از پردازنده بهره ببرند.

توجه: یک نقص صفحه (صفحه‌بندی در حافظه مجازی) مربوط به یک نخ، باعث نمی‌شود کل فرآیند مسدود شود. زیرا هسته سیستم عامل از نخ‌های داخل فرآیند کاربر باخبر است.

توجه: زمان‌بند سطح هسته برای زمان‌بندی نخ‌ها می‌تواند از Timer سیستم استفاده نماید.

توجه: این راه کار، منجر به امکان هم‌روندی (در سیستم‌های تک پردازنده‌ای) و امکان توازی (در سیستم‌های چندپردازنده‌ای) میان فرآیندهای مختلف و یا میان نخ‌های داخل یک فرآیند می‌شود. البته اگر نخی مسدود نگردد، درجه همروندی و توازی بالاتر هم خواهد رفت، زیرا در اینصورت همه نخ‌ها به طور هم‌روند یا موازی در حال حرکت هستند. مانند یک تیم فوتبال ۱۱ نفره که اگر بازیکنی مرتکب خطا گردد، همروندی یا توازی همچنان برقرار است، چون داور تک‌تک بازیکنان را می‌شناسد و فقط بازیکن خاطی را مسدود، جریمه و اخراج می‌کند و بقیه بازیکنان تیم به بازی خود ادامه می‌دهند، اما اگر هیچ یک از بازیکنان تیم مرتکب خطا نگردند، واضح است که همروندی و توازی بالاتر هم خواهد بود.

توجه: مثال کیک مطرح شده را مجدداً به یاد آورید، در روش سطح هسته، یک فرآیند چندنخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چندپردازنده‌ای بهره ببرد. در روش سطح هسته، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. بنابراین در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر است. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد.

توجه: در این روش، منابع به فرآیندها اختصاص می‌یابد ولی زمان‌بندی پردازنده، بر روی نخ‌ها

khalilifar.ir

انجام می‌گیرد و زمان‌بندی نخ‌های سطح کاربر و نخ‌های سطح هسته، برعهده زمان‌بند چندنخی سطح هسته می‌باشد.

توجه: سیستم عامل لینوکس، خانواده سیستم عامل ویندوز و سولاریس ۹ مدل یک به یک را پیاده‌سازی می‌کنند.

نخ‌های ترکیبی: در این روش اداره نخ شامل ایجاد، حذف و زمان‌بندی نخ به طور ترکیبی از سطح کاربر و سطح هسته انجام می‌شود.

این روش از اجتماع دو روش سطح کاربر و هسته ابداع گردیده است. در این روش علاوه بر زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته سیستم عامل، زمان‌بند چند نخی نیز در سطح کاربر برای زمان‌بندی و اولویت‌دهی نخ‌های فرآیند کاربر وجود دارد. در واقع زمان‌بندی نخ‌های فرآیندهای سیستم عامل، توسط هسته سیستم عامل و زمان‌بندی نخ‌های فرآیندهای کاربر، توسط برنامه کاربر انجام می‌گردد. که این امر منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد. در این روش، زمان‌بند چند نخی در سطح کاربر، نخ‌های کاندید خود را از میان نخ‌های متعدد در فرآیندهای مختلف کاربر براساس یک الگوریتم خاص انتخاب و تحویل یک زمان‌بند چند نخی در سطح هسته می‌دهد. در واقع انتخاب نخ‌های کاندید (زمان‌بندی) در فرآیندهای کاربر به خود کاربر واگذار شده است که همانطور که گفتیم منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد. حال اجتماع حاصل از نخ‌های کاندید فرآیندهای کاربر و نخ‌های فرآیندهای سیستم عامل توسط زمان‌بند چندنخی هسته سیستم عامل براساس یک الگوریتم خاص زمانبندی می‌شود.

از توضیحات فوق این مهم برداشت می‌شود که هسته سیستم عامل باید این قابلیت را داشته باشد که کاربر بتواند نخ‌های فرآیندهای سطح خود را به هسته سیستم عامل معرفی کند. بنابراین در این روش هسته سیستم عامل نه تنها فرآیندهای کاربر را می‌شناسد، بلکه از وجود نخ‌های کاربر داخل فرآیندهای کاربر نیز در صورت وجود نخ آگاه است.

توجه: برای معرفی نخ‌های کاندید فرآیندهای سطح کاربر به زمانبند چندنخی در سطح هسته سیستم عامل، علاوه بر جایگاه‌های مخصوص نخ‌های فرآیندهای سیستم عامل در زمان‌بند چندنخی در سطح هسته سیستم عامل، تعدادی جایگاه، ویژه نخ‌های سطح کاربر نیز، در زمانبند چندنخی در سطح هسته سیستم عامل در نظر گرفته شده است، به این جایگاه ویژه که **محیط اجرای نخ** نیز نامیده می‌شود، LWP گفته می‌شود. در واقع هر نخ کاندید انتخاب شده توسط زمانبند چندنخی در سطح کاربر، پس از معرفی به زمانبند چندنخی در سطح هسته، در یکی از جایگاه‌های ویژه که همان LWP است، جهت زمان‌بندی توسط زمان‌بند چندنخی در سطح هسته، قرار می‌گیرد.

توجه: LWP سرواژه‌ی عبارت Light Weight Process و به معنی فرآیند سبک وزن است.

توجه: هنگامی که یک نخ به زمان‌بند چندنخی در سطح هسته معرفی می‌گردد و در ادامه در یک

khalilifar.ir

LWP جهت زمان‌بندی توسط زمان‌بند چندنخی در سطح هسته، قرار می‌گیرد، در طول حیات خود ممکن است، در LWP های متفاوتی بخش‌هایی از اجرای خود را طی کند، مثلاً یک نخ در صورت رسیدن به عملیات ورودی و خروجی، جایگاه خود یعنی LWP را واگذار می‌کند و در اجرای بعدی پس از پایان عملیات ورودی و خروجی ممکن است به یک LWP دیگر منتسب شود. توجه کنید که LWP محیط اجرای نخ می‌باشد.

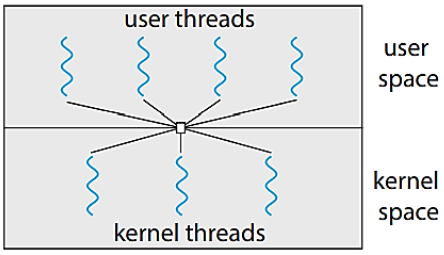
توجه: برای انجام زمان‌بندی چندنخی، هسته سیستم عامل باید علاوه بر جدول فرآیندها، یک جدول نخ (شبیه جدول فرآیند) داشته باشد که اطلاعات تمامی نخ‌های موجود در سیستم را نگهداری کند.

توجه: در این روش نخ ماهیت فیزیکی دارد و از دید کاربر و سیستم عامل وجود دارد، بنابراین در این روش، هر نخ، رجیستر و پشته مختص به خود را دارد، به عبارت دیگر هر نخ TCB مختص به خود را دارد.

توجه: عملیات مرتبط با نخ‌های سطح کاربر، مانند (بارگذاری TCB مختص به نخ) و پایان دادن (ذخیره‌سازی TCB مختص به نخ) بر عهده هسته سیستم عامل است و توسط زمان‌بند چندنخی سطح هسته انجام می‌گردد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، با اعمال نوعی اولویت‌بندی بین برخی از اعضاء یک کشور تقسیم گردد (حال این اعضاء شهروند عام باشند یا خاص) در این روش هریک از اعضاء اولویت‌دار و منتخب کشور شماره حساب مختص به خود را دارند. اما اگر یکی از اعضاء خانواده خطایی انجام دهد و محکوم گردد، آنگاه فقط همان فرد برای مدتی از خدمات دولت محروم می‌گردد و دولت به بقیه اعضاء خانواده آن فرد، همچنان خدمات ارائه می‌دهد، زیرا دولت از مشخصات تک تک اعضاء آن خانواده آگاه است.

توجه: در این روش چند نخ سطح کاربر به تعداد کم‌تر یا مساوی از نخ‌های سطح هسته (LWP ها) نگاشت می‌شود که به آن مدل چند به چند (Many to Many model) گفته می‌شود.



Many-to-many model.

khalilifar.ir

توجه: عمل تعویض متن مابین نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل، در مد هسته پردازنده توسط زمانبند چندنخی در سطح هسته سیستم عامل انجام می‌گردد و عمل تعویض متن مابین فرآیندها (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل و در مد هسته پردازنده توسط زمانبند پردازنده انجام می‌گردد.

توجه: به دلیل آنکه نخ‌های کاندید فرآیندهای سطح کاربر، به هسته سیستم عامل معرفی می‌گردد و هسته سیستم عامل از وجود نخ‌های کاندید فرآیندهای سطح کاربر آگاه است، اگر یک نخ کاندید موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم‌خانواده با آن نخ مسدود شده، همچنان می‌توانند از پردازنده بهره ببرند. این راه‌کار، منجر به امکان هم‌روندی (در سیستم‌های تک‌پردازنده‌ای) و امکان توازی (در سیستم‌های چندپردازنده‌ای) میان فرآیندهای مختلف کاربر و یا میان نخ‌های داخل یک فرآیندکاربر می‌شود. البته اگر نخی مسدود نگردد، درجه هم‌روندی و توازی بالاتر هم خواهد رفت، زیرا در اینصورت همه نخ‌ها به طور هم‌روند یا موازی در حال حرکت هستند.

مانند یک تیم فوتبال ۱۱ نفره که اگر بازیکنی مرتکب خطا گردد، هم‌روندی یا توازن همچنان برقرار است، چون داور تک تک بازیکنان را می‌شناسد و فقط بازیکن خاطی را مسدود، جریمه و اخراج می‌کند و بقیه بازیکنان تیم به بازی خود ادامه می‌دهند، اما اگر هیچ‌یک از بازیکنان تیم مرتکب خطا نگردد، واضح است هم‌روندی و توازی بالاتر هم خواهد رفت.

توجه: همانند نخ‌های سطح کاربر، همروندی و توازی برای نخ‌های سطح هسته سیستم عامل نیز، در حالت انسداد یک نخ یا عدم انسداد یک نخ، برقرار است.

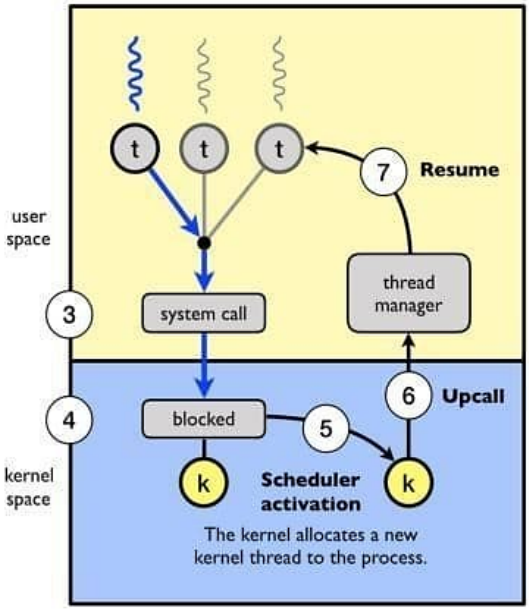
توجه: مثال کیک مطرح شده را مجدداً به یاد آورید، در روش ترکیبی، یک فرآیند چندنخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چندپردازنده‌ای بهره ببرد. در روش ترکیبی، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر است. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد.

توجه: نخ‌ها به دو صورت ممکن است مسدود شوند، یکی با فراخوان سیستمی و توسط هسته سیستم عامل و دیگری با فراخوان تابعی برای خوابیدن از کتابخانه سطح کاربر، بنابراین در روش ترکیبی علاوه بر اداره نخ‌ها در سطح هسته، کتابخانه سطح کاربر نیز می‌تواند بدون دخالت هسته اولویت اجرای نخ‌ها را در زمان اجرا جا به جا کند. برای مثال اگر نخ درحال اجرای فعلی بر روی یک LWP بخواهد بخوابد تا نخ دیگری عملیاتی را انجام داده و سپس او را بیدار کند، می‌تواند با فراخوانی تابع کتابخانه در سطح کاربر بخوابد. کتابخانه سطح کاربر ثبات‌ها و داده‌های نخی که کاندید خوابیدن است را در جدول نخ ذخیره می‌کند و سپس، نخ دیگری را برای اجرا انتخاب

khalilifar.ir

می‌کند و ثبات‌ها و داده‌هایش از جدول نخ به پردازنده بارگذاری می‌شود. بنابراین نخ جدید بر روی همان LWP نخ قبلی اجرا می‌شود. دقت کنید که زمان‌بند سطح هسته متوجه این تعویض نخ سطح کاربر نمی‌شود.

توجه: هسته به یک فرآیند کاربر تعدادی LWP اختصاص می‌دهد تا نخ‌های خود را بر روی آنها زمان‌بندی کند. همچنین سطح هسته موظف است وقوع رویدادها را به سطح کاربر اطلاع دهد. شروع این ارتباط بین کتابخانه نخ سطح کاربر و سطح هسته از فعال‌سازی زمان‌بند (Scheduler Activation) است و ادامه آن فراخوان رو به بالا (Upcall) است. در کتابخانه نخ سطح کاربر برای اداره Upcall ها، Upcall Handler وجود دارد که داخل LWP اجرا می‌شود. هرگاه نخ درحال اجرا بر روی یک LWP از طریق هسته مسدود شود، هسته یک Upcall صادر می‌کند و مشخص می‌کند که کدام نخ را مسدود کرده است. در این لحظه هسته یک LWP دیگر را در اختیار سطح کاربر می‌گذارد تا Upcall Handler بر روی آن اجرا شده و پس از ذخیره وضعیت نخ مسدود شده در جداول فضای کاربر، یک نخ جدید را برای اجرا بر روی LWP جدید زمان‌بندی کرده و وضعیت آنرا از جداول مربوطه به CPU بارگذاری کند. همچنین هرگاه رویدادی که نخ مسدود منتظر آن بوده است رخ دهد برای مثال تکمیل ورودی و خروجی، هسته یک Upcall دیگر به کتابخانه نخ کاربر می‌دهد تا آنرا از آماده اجرا بودن نخ مورد نظر آگاه سازد.



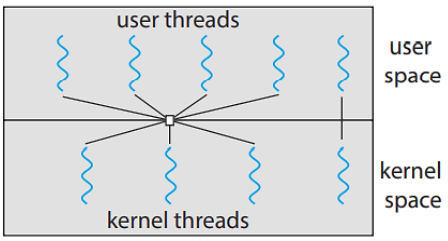
در **سیستم‌های بی‌درنگ** به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها داده شود. در روش چند به

khalilifar.ir

چند، سطح هسته وجود دارد و به تبع همروندی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود دارد که این امر منجر به این می‌شود که زمان پاسخ کوتاه باشد و پاسخ قبل از اتمام مهلت زمانی تولید شود. اما یک ریسک دارد اینکه در روش چند به چند برنامه‌نویس به تعداد دلخواه نخ ایجاد می‌کند، اما ممکن است تعداد LWP کمتری به آنها منتسب شود تا سربار هسته کمتر شود. زیرا در روش چند به چند نخ‌های سطح کاربر به تعداد کم‌تر یا مساوی از نخ‌های سطح هسته نگاشت می‌شود. برای مثال اگر یک فرآیند پنج درخواست (نخ) همزمان داشته باشد، اگر چهار LWP داشته باشیم، یکی از درخواست‌ها (نخ‌ها) تا بازگشت نتیجه یکی از چهار درخواست (نخ) دیگر از سطح هسته به جریان نخواهد افتاد و این یعنی تاخیر و ممکن است زمان پاسخ طولانی شود و به تبع این احتمال وجود دارد که پاسخ بعد از اتمام مهلت زمانی تولید شود. هرچند که در روش چند به چند، سطح کاربر وجود دارد، که این امر منجر به این می‌شود که اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده شود.

توجه: روش دو سطحی به نوعی روش **چند به چند پلاس** است، یعنی روش دوسطحی بهبود یافته روش چند به چند است، در واقع روش دوسطحی علاوه بر اینکه روش چند به چند را در خود دارد، جهت بهبود کارایی روش یک به یک را نیز در خود قرار داده است. در سیستم‌های بی‌درنگ به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها داده شود. در روش دوسطحی (Two-Level Model) یا همان چند به چند پلاس، سطح هسته وجود دارد و به تبع همروندی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود دارد که این امر منجر به این می‌شود که زمان پاسخ کوتاه باشد و پاسخ قبل از اتمام مهلت زمانی تولید شود. اما یک ریسک دارد اینکه در روش چند به چند برنامه‌نویس به تعداد دلخواه نخ ایجاد می‌کند، اما ممکن است تعداد LWP کمتری به آنها منتسب شود تا سربار هسته کمتر شود. زیرا در روش چند به چند نخ‌های سطح کاربر به تعداد کم‌تر یا مساوی از نخ‌های سطح هسته نگاشت می‌شود. برای مثال اگر یک فرآیند پنج درخواست (نخ) همزمان داشته باشد، اگر چهار LWP داشته باشیم، یکی از درخواست‌ها (نخ‌ها) تا بازگشت نتیجه یکی از چهار درخواست (نخ) دیگر از سطح هسته به جریان نخواهد افتاد و این یعنی تاخیر و ممکن است زمان پاسخ طولانی شود و به تبع این احتمال وجود دارد که پاسخ بعد از اتمام مهلت زمانی تولید شود. اما از آنجا که روش دوسطحی علاوه بر داشتن روش چند به چند، روش یک به یک را نیز دارد، می‌تواند در مواقع لزوم برای پاسخ سریع از روش یک به یک نیز استفاده نماید، زیرا در روش یک به یک هر رشته نخ کاربر به یک رشته نخ هسته نگاست می‌شود. شکل زیر گویای مطلب است:

khalilifar.ir



Two-level model.

همچنین در روش دوسطحی (Two-Level Model) یا همان چند به چند پلاس، سطح کاربر وجود دارد، که این امر منجر به این می‌شود که اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده شود. روش دوسطحی در سیستم عامل‌هایی نظیر IRIX، HP-UX و True64nIx پشتیبانی می‌شود. نسخه‌های قبل از Solaris9 نیز مدل دوسطحی را پشتیبانی می‌کنند، اما Solaris9 از روش یک به یک استفاده می‌کند.

صورت سوال به این شکل است:

کدام گزینه درباره ریسه‌های (Threads) سطح کاربر و سطح هسته درست است؟

۱) زمان‌بندی ریسه‌های سطح هسته سریعتر از ریسه‌های سطح کاربر است.

گزینه اول نادرست است، زیرا زمان‌بندی ریسه‌های سطح هسته کندتر از ریسه‌های سطح کاربر است. زمان‌بندی نخ‌های سطح هسته به دلیل دخالت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مد هسته به مد کاربر شامل تغییر MMU و TLB)، کندتر از ریسه‌های سطح کاربر است. در نتیجه نخ‌های هسته به طور کلی فرایند ایجاد، حذف و مدیریت کُندتری نسبت به نخ‌های کاربر دارند.

۲) ریسه‌های سطح کاربر و سطح هسته از طریق فراخوانی سیستمی (System Calls) به هم سرویس می‌دهند.

گزینه دوم نادرست است، زیرا زمان‌بندی نخ‌های سطح هسته نیاز به فراخوانی سیستمی و دخالت هسته و تله‌های سنگین دارد. اما سرویس ریسه‌های سطح کاربر و سطح هسته به هم ارتباطی به فراخوان سیستمی ندارد. فراخوان سیستمی برای صدا زدن یک تابع از سیستم عامل مورد استفاده قرار می‌گیرد و نه صدا زدن یک نخ از یک فرآیند. اصولا نخ توسط زمان‌بند سطح کاربر یا هسته زمان‌بندی و اجرا می‌شود و صدا زده نمی‌شود.

۳) ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس هم دسترسی داشته و می‌توانند در فضای آدرس هم بنویسند.

گزینه سوم درست است، زیرا اشتراکات نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه

khalilifar.ir

(PC)، رجیسترها و پشته می‌باشد. از آنجاکه نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند. بنابراین ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس مشترک هم دسترسی داشته و می‌توانند در فضای آدرس مشترک هم بنویسند.

۴) ریسه‌های سطح هسته به فضای آدرس ریسه‌های سطح کاربر دسترسی دارند، اما ریسه‌های سطح کاربر به فضای آدرس ریسه‌های سطح هسته دسترسی ندارند.

گزینه چهارم نادرست است، زیرا اشتراکات نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد. از آنجاکه نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند. بنابراین ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس مشترک هم دسترسی داشته و می‌توانند در فضای آدرس مشترک هم بنویسند.

۴- گزینه (۱) صحیح است.

در تکنیک صفحه‌بندی، برای هر فرآیند یک جدول صفحه معمولی تشکیل می‌شود که در آن به ازای همه صفحه‌های یک فرآیند، درایه وجود دارد و هر درایه مشخص می‌کند که کدام قاب فیزیکی به این صفحه اختصاص یافته است. در این روش وقتی فرآیندها بزرگ باشند، هزینه نگهداری جداول صفحه بسیار زیاد می‌شود، در ضمن به ازای هر فرآیند نیز باید یک جدول صفحه داشته باشیم. به عبارتی وقتی تعداد و اندازه فرآیندها بزرگ شود، این روش برای ترجمه آدرس مجازی به فیزیکی مقرون به صرفه نیست.

برای حل این مشکل **جدول صفحه چندسطحی** و **جدول صفحه معکوس** ابداع شده است. جدول صفحه معکوس از نظر سربار حافظه بهتر از روش جدول صفحه چندسطحی است و حالت حداقلی است. اغلب سیستم‌های کامپیوتری، فضای آدرس منطقی(مجازی) بزرگی را پشتیبانی می‌کنند، مانند کامپیوترهایی که آدرس‌های 32 یا 64 بیتی را پشتیبانی می‌کنند، در چنین محیط‌هایی جدول صفحه معمولی بسیار بزرگ خواهد شد. برای مثال یک سیستم را با 32 بیت فضای آدرس منطقی(مجازی) در نظر بگیرید. اگر اندازه صفحه در این سیستم برابر با 4KB (2^{12}) باشد، در اینصورت جدول صفحه شامل بیش از یک میلیون درایه خواهد بود. $\left(2^{32} / 2^{12} = 2^{20}\right)$. فرض کنید هر درایه جدول صفحه معمولی، شامل 4 بایت باشد، در اینصورت هر فرآیند به 4MB $\left(4 \times 2^{20}\right)$ فضای آدرس فیزیکی فقط برای نگهداری جدول صفحه معمولی نیاز دارد. همچنین مشخص است که با توجه به محدودیت اندازه قاب (4KB) نمی‌توان این جدول صفحه معمولی را بصورت پیوسته درون یک قاب جا داد. 4MB اندازه جدول صفحه معمولی در فضای 4KB مربوط به یک قاب جا نمی‌شود. یک راه حل تقسیم جدول صفحه معمولی به جداول صفحه جزئی و ایجاد جدول صفحه چندسطحی است، در این حالت جدول صفحه معمولی نیز همانند فرآیند

khalilifar.ir

صفحه‌بندی می‌شود. راه حل دیگر استفاده از جدول صفحه معکوس است. در راه حل جدول صفحه معکوس به جای اینکه در جداول صفحه مربوط به هر فرآیند، به ازای هر صفحه مجازی یک درایه داشته باشیم، به ازای هر قاب در حافظه فیزیکی یک درایه در جدول صفحه معکوس نگهداری می‌کنیم. در واقع به ازای هر قاب حافظه اصلی اینکه در حال حاضر کدام صفحه مربوط به کدام فرآیند در این قاب ذخیره شده است نگهداری می‌شود. بنابراین تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی(اصلی) است.

مثال: در یک سیستم صفحه‌بندی که دارای 34 بیت آدرس است 23 بیت اول برای شماره صفحه و 11 بیت بعدی برای آدرس‌دهی درون صفحه است. در یک سیستم با صفحه‌بندی معکوس (Inverted Page Table) با 128 مگابایت حافظه، جدول صفحه دارای چند خانه (درایه یا مدخل) است؟

پاسخ: تعداد درایه‌های جدول صفحه معکوس، مطابق رابطه زیر محاسبه می‌گردد:

تعداد قاب‌های حافظه فیزیکی = تعداد درایه‌های جدول صفحه معکوس

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

همانطور که گفتیم تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی است، بنابراین تعداد درایه‌های جدول صفحه معکوس به صورت زیر خواهد بود:

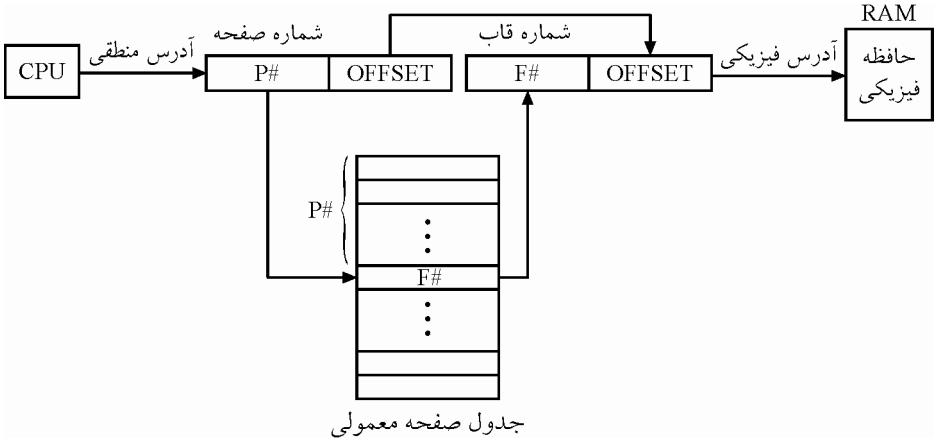
$$\text{تعداد درایه‌های جدول صفحه معکوس} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

در یک عبارت ساده، هدف از استفاده از جدول صفحه معکوس، **کاهش** میزان حافظه فیزیکی مورد نیاز برای ترجمه آدرس مجازی به فیزیکی است. با استفاده از تکنیک جدول صفحه معکوس، مقدار زیادی در حافظه فیزیکی صرفه‌جویی می‌شود اما تبدیل آدرس مجازی به فیزیکی کندتر و وقت‌گیرتر می‌شود. در واقع **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم. زیرا جدول صفحه معکوس بر اساس F# اندیس شده است و نه مانند جدول صفحه معمولی که بر اساس P# اندیس شده است. در واقع وقتی یک فرآیند با PID مختص به خود به صفحه مجازی P# مراجعه می‌کند، سخت‌افزار دیگر نمی‌تواند از P# به عنوان اندیس جدول صفحه استفاده کند و صفحه فیزیکی را بیابد و از این جهت باید سرتاسر جدول صفحه معکوس (وارونه) را برای یافتن درایه (PID,P#) از آدرس مجازی (PID,P#,OFFSET) جستجو کند. اگر یک تطبیق پیدا شود، در اینصورت آدرس فیزیکی (F#,OFFSET) تولید خواهد شد و اگر هیچ تطبیقی یافته نشود، در اینصورت یک دسترسی آدرس غیر مجاز می‌باشد. ضمن اینکه این جستجو باید به ازای هربار دسترسی به حافظه انجام شود. همانطور که گفتیم در جدول صفحه معکوس **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست

khalilifar.ir

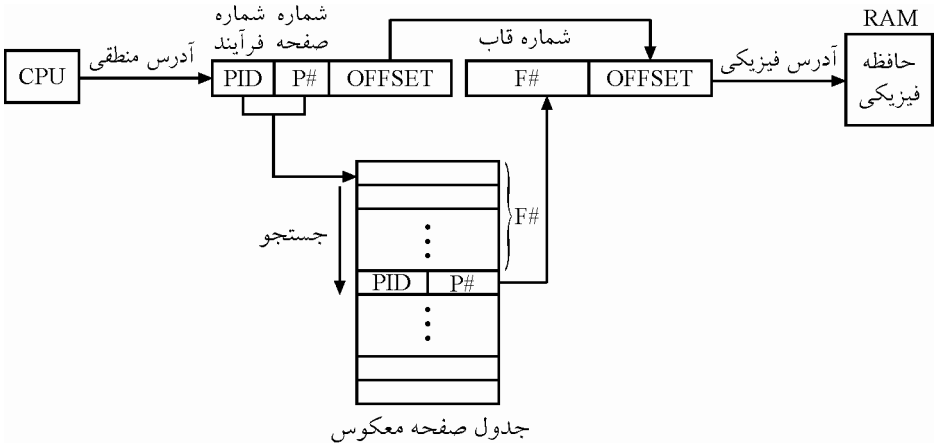
می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم، یعنی سرعت نگاشت آدرس منطقی به فیزیکی **کاهش** می‌یابد. درواقع جدول صفحه معکوس **زمان** نگاشت آدرس منطقی به آدرس فیزیکی را **افزایش** می‌دهد. هرچند موجب صرفه‌جویی در مصرف حافظه فیزیکی می‌شود. **بنابراین گزینه اول درست و گزینه‌ای دوم، سوم و چهارم نادرست است.**

نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معمولی** به صورت زیر است:



جدول صفحه معمولی

نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معکوس** به صورت زیر است:



جدول صفحه معکوس

توجه: برای افزایش سرعت تبدیل آدرس مجازی به فیزیکی در جدول صفحه معکوس می‌توان از تکنیک TLB استفاده نمود.

توجه: جدول صفحه معکوس، اطلاعات کاملی در مورد فضای آدرس منطقی یک فرآیند را ندارد. در حالی که سیستم صفحه‌بندی مبتنی بر تقاضا به این اطلاعات کامل جهت پردازش نقص‌های صفحه نیازمند است. برای دسترسی به این اطلاعات کامل، یک **جدول صفحه خارجی**، یکی به

khalilifar.ir

ازای هر فرآیند، باید نگهداری شود. جدول صفحه خارجی شامل فیلدهای Valid، Present و آدرس صفحه مجازی دچار نقص صفحه شده بر روی رسانه ذخیره‌سازی است.

توجه: هنگامی که یک نقص صفحه رخ می‌دهد، ممکن است یک **نقص صفحه دیگر** برای بارگذاری **جدول صفحه خارجی** از رسانه ذخیره‌سازی به حافظه فیزیکی رخ دهد. بنابراین در جدول صفحه معکوس، زمان سرویس نقص صفحه(page fault) به دلیل ایجاد یک نقص صفحه دیگر، **افزایش** می‌یابد.

توجه: در سیستم‌هایی که از جدول صفحه معکوس‌شده استفاده می‌کنند، استفاده اشتراکی از صفحات میان فرآیندها در حالت عادی امکان‌پذیر نیست، در حالت کلی اشتراک به شکل چند به یک است، یعنی صفحه مشترک میان فرآیندها در یک قاب مشخص قرار می‌گیرد. که پیاده‌سازی حالت چند به یک در جدول صفحه معکوس درحالت ابتدایی امکان‌پذیر نیست. زیرا در جدول صفحه معکوس برای هر درایه آن فقط می‌توان شماره یک صفحه از یک فرآیند را قرار داد. البته توسط مکانیزم‌هایی می‌توان اشتراک‌گذاری صفحات میان فرآیندها را در جدول صفحه معکوس نیز پیاده‌سازی نمود. برای مثال به جدول صفحه معکوس اجازه دهیم برای هر درایه، بیش از یک شماره صفحه را آدرس‌دهی کند.

۵- گزینه (۳) صحیح است.

تکه تکه شدن داخلی (Internal Fragmentation) زمانی رخ می‌دهد که حافظه از قبل مرزبندی شده باشد و در هر قسمت بتوان فقط یک قطعه را قرار داد. در این حالت چون احتمالاً قطعه‌ای که برای یک جایگاه انتخاب شده است قدری از آن کوچکتر است، در انتهای جایگاه مقداری فضای بلااستفاده به‌وجود خواهد آمد. به این مشکل تکه تکه شدن داخلی گویند.

اما وقتی از قبل مرزبندی مشخصی برای حافظه در نظر نگیریم احتمال وقوع تکه تکه شدن خارجی (External Fragmentation) وجود دارد. در این حالت احتمالاً فضاهای خالی و قابل استفاده حافظه به صورت جایگاه‌های کوچک، لابه لای فرآیندها پخش شده‌اند که مجموع این جایگاه‌های کوچک پراکنده احتمالاً نیاز ما را برآورده می‌کنند، اما چون همجوار نیستند بلااستفاده باقی می‌مانند.

در «**تکنیک صفحه‌بندی**» حافظه به بخش‌هایی با اندازه‌ی یکسان به نام قاب (Frame) تقسیم می‌شود. از طرفی برنامه‌ها نیز به قسمت‌های مساوی و هم‌اندازه با قاب‌ها تقسیم می‌شوند که به آن‌ها صفحه (Page) می‌گویند. حال هنگامی که برنامه‌ای به حافظه منتقل می‌شود باید تمام صفحاتش به داخل قاب‌های خالی آورده شوند. در این حالت اصلاً نیازی نیست صفحات مربوط به یک فرآیند در قاب‌های همجوار قرار گیرند.

مزیت عمده این روش «**از بین بردن و صفر شدن تکه تکه شدن خارجی**» و به حداقل رساندن تکه تکه شدن داخلی می‌باشد، اما در عوض عملیات محاسبه آدرس‌ها و مدیریت این صفحات

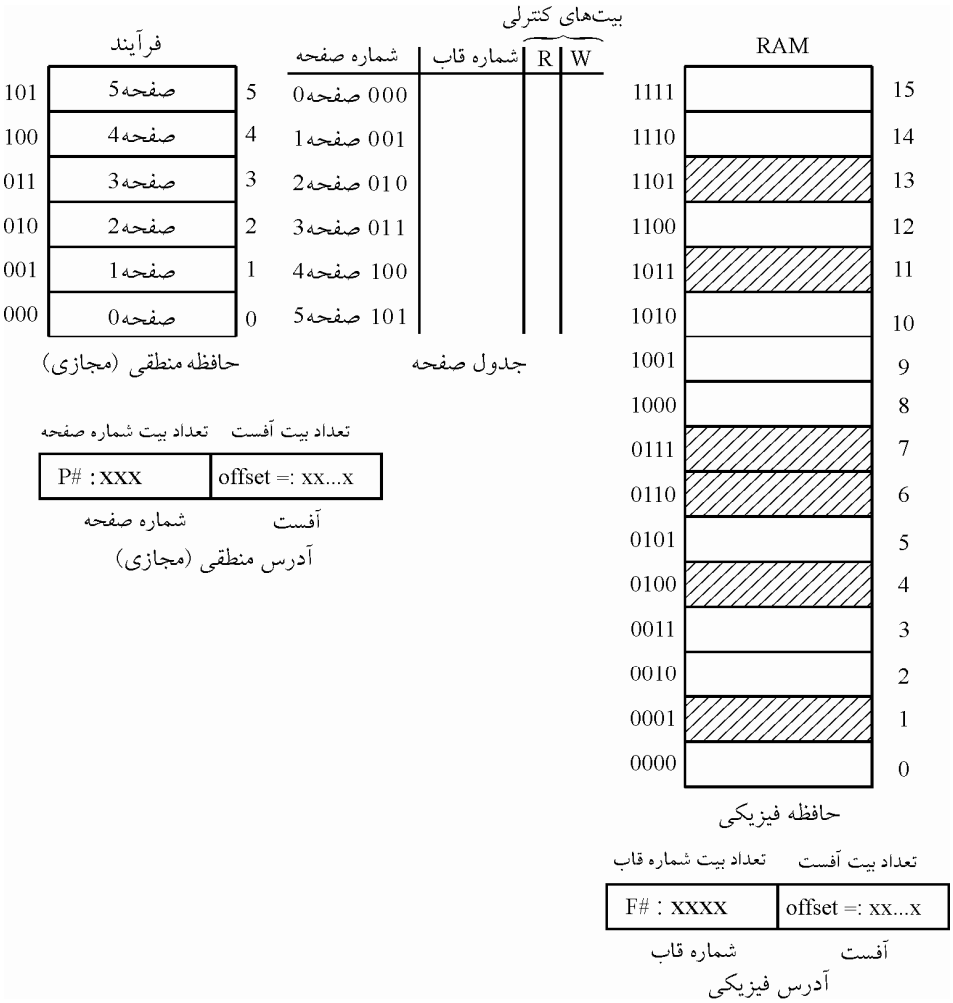
khalilifar.ir

قدری هزینه‌بر و زمان‌گیر است.

توجه: برای پیاده‌سازی این روش و مدیریت صفحه‌ها و از همه مهم‌تر تبدیل و نگاشت آدرس‌ها باید یک جدول صفحه به ازای هر فرآیند در نظر گرفت. در واقع جدول صفحه‌ی هر فرآیند دارای یک درایه به ازای هر صفحه می‌باشد که مشخص می‌کند هر صفحه از یک فرآیند در کدام قاب حافظه نگهداری می‌شود.

توجه: نقطه ضعف اصلی مکانیزم صفحه‌بندی این است که اگر فقط احتیاج به ناحیه بسیار کوچکی از حافظه باشد، در این صورت مقداری از فضای حافظه تلف می‌شود، زیرا کوچکترین واحدی از حافظه که می‌توان آن را به استفاده‌کننده اختصاص داد، یک صفحه است.

توجه: مطابق اطلاعات مساله شکل زیر گویای مطلب است:



khalilifar.ir

$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}}$$

$$\text{تعداد قاب‌های حافظه‌ی فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}}$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^6 = 3bit$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{16} = 4bit$$

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{2048} = 11bit$$

عرض جدول صفحه × تعدادصفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد. دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطح جدول صفحه می‌باشد.

توجه: در صورت سوال مطرح شده است که اگر Internal Fragmentation برابر 900 بایت باشد، اندازه پردازه و External Fragmentation چند بایت است؟

توجه: مطابق فرض سوال قسمت‌های خاکستری فریم‌های تخصیص داده شده به یک پردازه هستند، بنابراین تعداد صفحات فرآیند برابر 6 عدد است.

$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}}$$

بایت 12288= 2048× 6= اندازه صفحه (اندازه قاب) × تعداد صفحات فرآیند = اندازه فرآیند

توجه: در ایده‌ی صفحه‌بندی تکه تکه شدن خارجی نداریم اما ممکن است قدری تکه تکه شدن داخلی داشته باشیم و آن هم در هر فرآیند و به ازای «**آخرین صفحه**» رخ می‌دهد. در واقع چون فرآیند می‌تواند هر طولی داشته باشد، ممکن است مضرب صحیحی از اندازه صفحه‌ها نباشد و آخرین صفحه قدری خالی بماند. به همین دلیل به طور متوسط نیم صفحه به ازای هر فرآیند تکه تکه شدن داخلی خواهیم داشت. بنابراین اگر مقدار 900 بایت تکه تکه شدن داخلی که آن هم در هر فرآیند و به ازای «**آخرین صفحه**» رخ می‌دهد را از مقدار حداکثر اندازه فرآیند یعنی 12288 بایت کسر کنیم به مقدار 11388 بایت که اندازه واقعی پردازه است می‌رسیم. به صورت زیر:

$$12288 - 900 = 11388$$

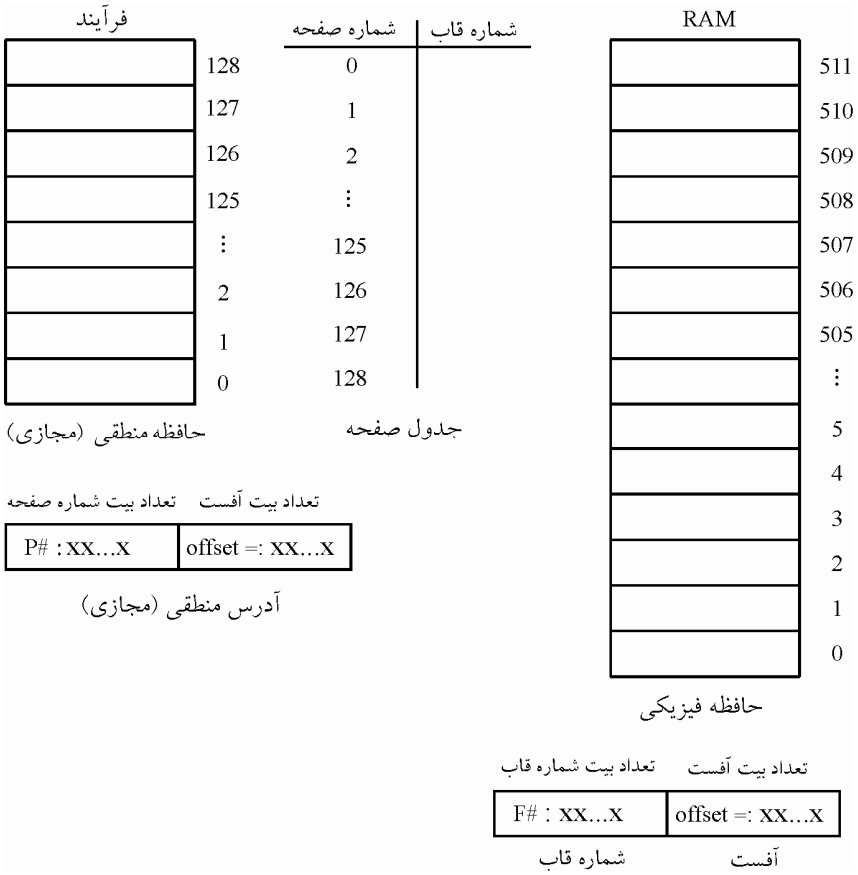
بنابراین پُرواضح است که گزینه سوم پاسخ سوال است.

khalilifar.ir

۶- گزینه (۲) صحیح است.

در صورت سوال مطرح شده است که طـول آدرس منطقـی 19 بیـت، تعـداد صـفحات فرآینـد در فضای آدرس منطقی 129 صفحه و اندازه فضـای آدرس فیزیکـی (حافظـه فیزیکـی) 1MB اسـت. همچنین مطابق فرض سوال طول بیـت‌هـای کنترلـی valid یـا invalid در page table در جـدول صفحه نادیده گرفته شده است و خواسته شده است طـول هـر مـدخل (entry) از جـدول صـفحه (Page Table) باید چند بیت باشد؟

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بنـدی بـه صـورت زیـر است:



توجه: مطابق فرض سوال تعداد صفحات فرآیند در فضـای آدرس منطقـی (فرآینـد) 129 صـفحه است.

$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}}$$

khalilifar.ir

تعداد بیت شماره صفحه = b = $\log_2^{\text{تعداد صفحات فرآیند}}$ = $\lceil \log_2^{129} \rceil$ = 8 bit

توجه: مطابق فرض سوال طول آدرس منطقی 19 بیت است.

همچنین، اندازه آدرس منطقی (مجازی) به صورت زیر است:

تعداد بیت شماره صفحه - طول آدرس منطقی = تعداد بیت آفست = 19bit − 8bit = 11bit

تعداد بیت آفست + تعداد بیت شماره صفحه = طول آدرس منطقی = 8bit + 11bit = 19bit

اندازه صفحه یا اندازه قاب = $2^{\text{تعداد بیت آفست}}$ = 2^{11} = 2048 B

همچنین داریم:

اندازه حافظه منطقی (فرآیند) = $2^{\text{تعداد بیت آدرس منطقی}}$ = $2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}}$

اندازه حافظه منطقی (فرآیند) = $2^{19} = 2^8 \times 2^{11} = 2^{19}$ = 512 KB

توجه: مطابق فرض سوال اندازه فضای آدرس فیزیکی (حافظه فیزیکی) $1MB = 2^{20}$ B است.

تعداد قاب‌های حافظه‌ی فیزیکی = $\frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}}$

تعداد قاب‌های حافظه‌ی فیزیکی = $\frac{2^{20}}{2^{11}} = 2^9 = 512$

تعداد بیت شماره قاب = b = $\log_2^{\text{تعداد قاب‌های حافظه فیزیکی}}$ = $\log_2^{512} = \log_2^{2^9}$ = 9bit

تعداد بیت آفست = b = $\log_2^{\text{اندازه صفحه یا اندازه قاب}}$ = $\log_2^{2^{11}}$ = 11bit

همچنین داریم:

اندازه حافظه فیزیکی (RAM) = $2^{\text{تعداد بیت آدرس فیزیکی}}$ = $2^{\text{تعداد بیت شماره قاب}} \times 2^{\text{تعداد بیت آفست}}$

اندازه حافظه فیزیکی (RAM) = $2^{20} = 2^9 \times 2^{11} = 2^{20}$ = 1 MB

همچنین، اندازه آدرس فیزیکی به صورت زیر است:

تعداد بیت شماره قاب = تعداد بیت آفست - طول آدرس فیزیکی = 20bit − 11bit = 9bit

طول آدرس فیزیکی = تعداد بیت شماره قاب + تعداد بیت آفست = 9bit + 11bit = 20bit

اندازه جدول صفحه = تعدادصفحات فرآیند (تعداد درایه‌های جدول صفحه) × عرض جدول صفحه

توجه: اندازه آدرس منطقی (مجازی) و فیزیکی (حقیقی) الزاماً برابر نیست.

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد، به صورت زیر:

khalilifar.ir

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب = عرض جدول صفحه

در سوال مطرح شده 0 بیت مربوط به بیت‌های کنترلی (مطابق فرض سوال طول بیت‌های کنترلی valid یا invalid در page table در جدول صفحه نادیده گرفته شده است) و 9 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس: عرض جدول صفحه فوق (طول هر مدخل (entry) از جدول صفحه (Page Table)) برابر $9bit + 0bit = 9bit$ می‌باشد.

همچنین اندازه جدول صفحه، از رابطه زیر محاسبه می‌گردد:

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

که مطابق رابطه فوق داریم:

اندازه جدول صفحه $= 129 \times 9bit = 1161bit$

بنابراین پُرواضح است که گزینه دوم پاسخ سوال است.

۷- گزینه (۲) صحیح است.

میانگین سربار تعویض متن در الگوریتم نوبت چرخشی (RR) به علت ثابت بودن برش زمانی زیاد است. پس مسئله این است که چگونه در کنار کاهش سربار سوئیچ، زمان بازگشت کارهای کوچک را نیز پایین نگه داریم؟ راه حل این است که با ایجاد صف‌های چندگانه (Multiple Queues)، کلاس‌های اولویت ایجاد شود. همه فرآیندهای تازه وارد در انتهای بالاترین صف (بالاترین کلاس اولویت) قرار می‌گیرند. فرآیندها در بالاترین کلاس، به مدت 1 کوانتوم اجرا می‌شوند. فرآیندهای کلاس بعدی 2 کوانتوم، فرآیندهای کلاس بعدی 4 کوانتوم و به همین ترتیب، فرآیند درون صف Q_n برابر 2^n کوانتوم می‌گیرد. اگر یک فرآیند از تمام کوانتوم مختص خودش در یک کلاس مورد نظر به صورت کامل استفاده کند و تمام نشود، به کلاس پایین‌تر منتقل می‌شود. پس از اولین اجرا (اولین کوانتوم) در صف اول برای فرآیند ممکن است سه حالت پیش بیاید:

حالت اول: ممکن است فرآیند مورد نظر با همان کوانتوم اول در صف اول تمام شود.

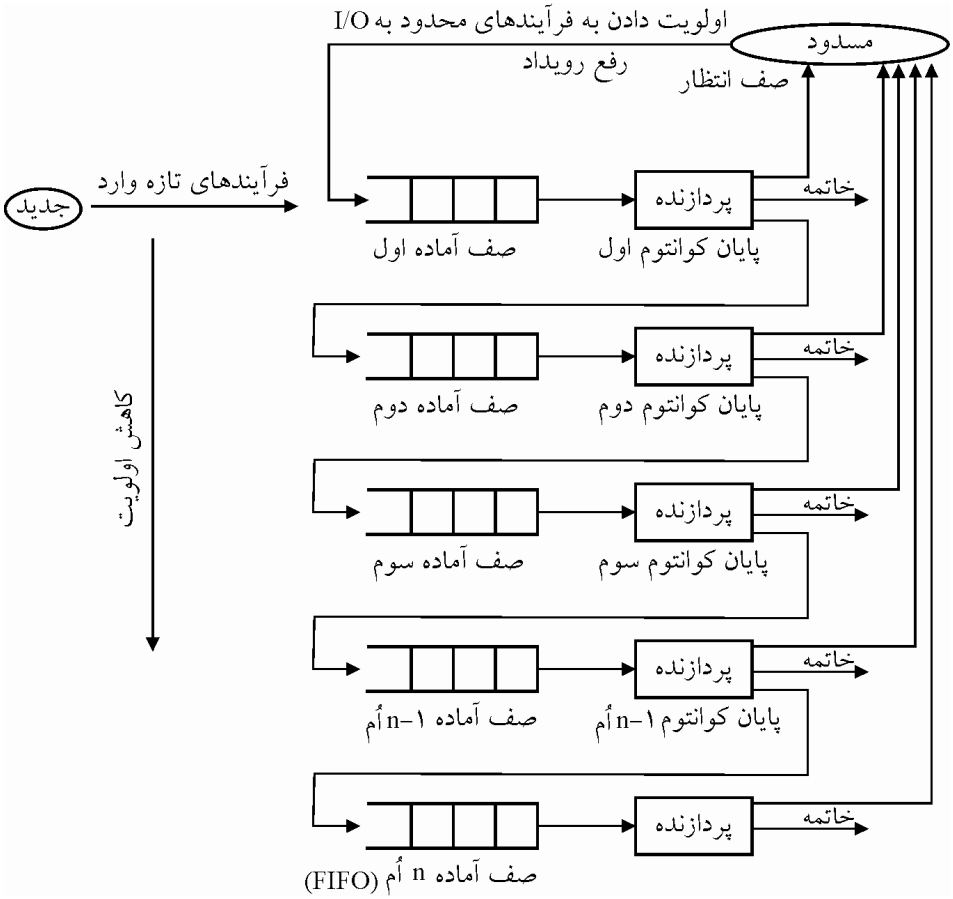
حالت دوم: ممکن است اجرای فرآیند مورد نظر ادامه داشته باشد که در این صورت، به صف دوم منتقل می‌شود. به همین ترتیب، فرآیندها پس از پایان کوانتوم زمانی در هر صف و تمام نشدن، به صف‌های پایین‌تر و کم اولویت‌تر بعدی منتقل می‌شوند که کوانتوم‌های متناظر با صف‌ها، با پایین آمدن سطح صف، افزایش می‌یابند. به این ترتیب هرچه فرآیندها در صف‌های اولویت، عمیق‌تر و عمیق‌تر فرو می‌روند، به همان اندازه فاصله زمانی مابین دوبار اجرای پشت سرهم آنها بیشتر و بیشتر می‌شود. در این الگوریتم فرآیندهای کوتاه به سرعت تکمیل می‌شوند بدون اینکه به صف‌های پایین‌تر منتقل شوند و از طرفی فرآیندهای طولانی کم کم به صف‌های پایین و کم

khalilifar.ir

اولویت‌تر هدایت می‌شوند. بنابراین فرآیندهای جدیدتر و کوتاه‌تر به فرآیندهای قدیمی و بلندتر ارجحیت می‌یابند.

حالت سوم: ممکن است فرآیند مورد نظر در یکی از صف‌ها برای وقوع رخدادی مثل ورودی و خروجی، منتظر بماند که در این صورت مسدود و به صف انتظار منتقل می‌شود. و پس از اتمام انتظار در انتهای صف اول به عنوان با اولویت‌ترین و بالاترین کلاس قرار می‌گیرد.

شکل زیر گویای مطلب است:



ایده موجود در روش MLFQ این است که فرآیندها با توجه به رفتارشان بین صف‌ها حرکت کنند. به عنوان مثال فرآیندی که در یک صف با اولویت بالاتر قرار دارد و زمان پردازنده را زیاد مصرف می‌کند، به صف پایین و پایین‌تر منتقل می‌شود و یا فرآیندی که مسدود می‌شود پس از رفع انسداد به صف اول با بالاترین اولویت منتقل می‌شود. زمانی سراغ پردازش فرآیندهای داخل صف‌های پایین‌تر می‌رویم که تمام صف‌های بالاتر از آن خالی باشد. اگر پردازش فرآیندی در یکی از

khalilifar.ir

صف‌های پایین‌تر در جریان باشد، و فرآیند جدیدی وارد صف اول شود، سیستم، پردازش ادامه فرآیندهای صف جاری را رها می‌کند و به صف اول باز می‌گردد، البته در مورد فرآیند جاری، این اجازه داده می‌شود که کوانتوم مربوطه را تمام کند و سپس بازگشت به صف اول انجام می‌شود. زیرا این الگوریتم «**غیرانحصاری مبتنی بر کوانتوم**» است.

توجه: مطابق سیاست‌های الگوریتم MLFQ منطقی است که هر فرآیند پس از عمل I/O به صف اول بازگردد، زیرا ممکن است فرآیندی در ابتدای روند پردازشش به مدت طولانی CPU مصرف کند که این امر منجر به این می‌شود که در گذر زمان فرآیند مورد نظر ته نشین شده و به صف‌های پایین‌تر و با اولویت کمتر منتقل شود، درحالیکه بعد از مدتی فرآیند مورد نظر به واسطه داشتن عملیات I/O به یک فرآیند تعاملی تبدیل شود، طبیعی است که جهت انجام عملیات تعاملی به CPU و به تبع اولویت بالاتر و سریع‌تر نیاز است، پس فرآیند مورد نظر پس از عمل I/O نباید در صف با اولویت پایین‌تر بماند و باید به صف اول جهت دریافت سریعتر CPU منتقل شود. در نتیجه این الگوریتم به نفع فرآیندهای با زمان اجرای کوتاه مدت یا محدود به I/O (I/O Bound) است. به طور مثال، یک فرآیند با وقفه I/O وارد صف انتظار شده و پس از انجام عمل I/O و به تبع رفع انتظار، برای اجرای مجدد به صف اول با بالاترین اولویت باز می‌گردد، بنابراین این الگوریتم به نفع فرآیندهای محدود به I/O هستند.

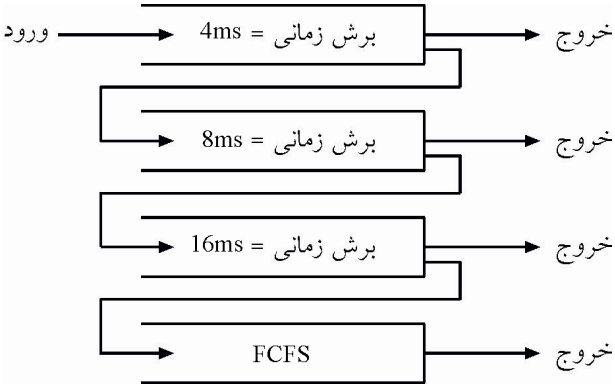
توجه: این الگوریتم به کارهای کوتاه اولویت می‌دهد، بدون آنکه لازم باشد قبل از اجرای یک کار، مدت اجرای آنرا بداند بر خلاف الگوریتم‌های SJF، SRT و HRRN که باید زمان اجرا را از قبل دانست. چون این الگوریتم بر اساس مکانیزم فیدبک (FB : FeedBack) رفتار می‌کند، ابتدا فرآیند وارد صف اول می‌شود و یک کوانتوم جهت اجرا به آن اختصاص می‌یابد، اگر در صف اول و کوانتوم اول فرآیند تمام شد که شد، اگر تمام نشد و بازهم CPU می‌خواست به صف دوم می‌رود و یک کوانتوم دیگر جهت اجرا به آن اختصاص می‌یابد. و این رویه ته نشینی به سمت صف‌های پایین‌تر آنقدر ادامه پیدا می‌کند تا فرآیند تمام شود. طبیعی است که اگر یک فرآیند خیلی کوتاه باشد به طور خودکار با بالاترین اولویت اجرا شده و از سیستم خارج می‌شود. هرچند که مشکل قحطی زدگی کارهای طولانی وجود دارد.

توجه: در این الگوریتم نرخ عملیات تعویض متن تا حد قابل توجهی کاهش می‌یابد، چون اگر فرآیندی زمان پردازش زیادی را لازم داشته باشد، به جای تخصیص مکرر کوانتوم‌های کوچک، به صف‌های پایین‌تر که کوانتوم زمانی‌شان بیش‌تر است هدایت می‌شود.

توجه: این الگوریتم با نام‌های صف‌های چندگانه یا صف‌های چندسطحی بازخوردی (**MLFQ**: Multi Level FeedBack Queus) یا MFQ یا FB شناخته می‌شود.

khalilifar.ir

مثال: فرض کنید سیستمی با 4 صف در اختیار داریم، در صف اول از روشRR با برش زمانی 4 میلی‌ثانیه، در صف دوم از روش RR با برش زمانی 8 میلی‌ثانیه، در صف سوم از روش RR با برش زمانی 16 میلی‌ثانیه و در صف آخر از روشFCFS استفاده می‌کنیم (شکل زیر).



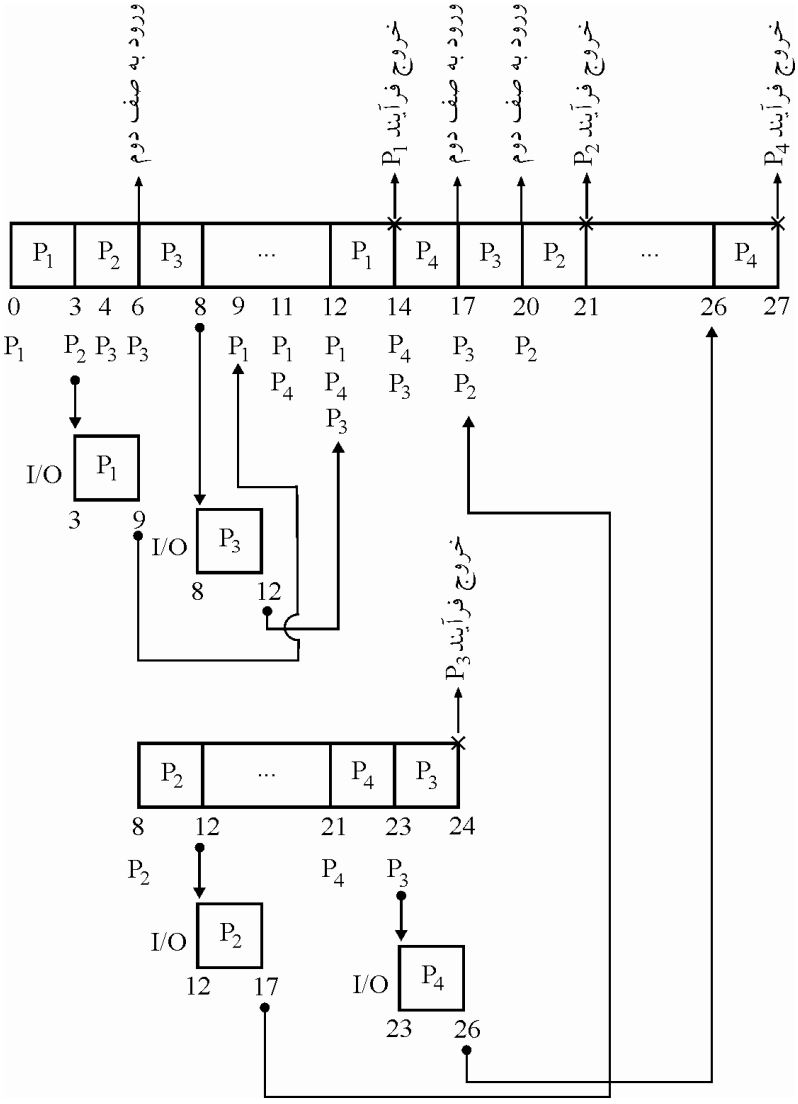
مثالی از الگوریتم MLFQ

در این حالت فرآیندها ابتدا به صف اول وارد می‌شوند، اگر در این صف به پایان نرسیدند به انتهای صف دوم، اگر در این صف نیز کامل نشدند به انتهای صف سوم و اگر باز هم به پایان نرسیدند به انتهای صف چهارم وارد می‌شوند که از روش FCFS استفاده می‌کنند.
با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	CPU Burst1	I/O Burst1	CPU Burst2	زمان انتظار+	زمان بازگشت =
P_1	0	3	6	2		
P_2	3	7	5	1		
P_3	4	2	4	4		
P_4	11	5	3	1		

ابتدا فرآیندها در صف اول با تکه زمانی معادل 3 واحد زمانی قرار می‌گیرند. و اگر در مدت 3 واحد زمانی تمام نشوند به صف دوم با تکه زمانی معادل ∞ واحد زمانی یعنی الگوریتم FCFS منتقل می‌گردند، تا بالاخره تمام شوند.
نمودار گانت کلی به صورت زیر است:

khalilifar.ir



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 14 - 0 = 14$

P_2 زمان بازگشت $= 21 - 3 = 18$

P_3 زمان بازگشت $= 24 - 4 = 20$

P_4 زمان بازگشت $= 27 - 11 = 16$

میانگین زمان بازگشت $= ATT = \frac{14+18+20+16}{4} = \frac{68}{4} = 17$

khalilifar.ir

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 14 - 5 = 9$

P_2 زمان انتظار $= 18 - 8 = 10$

P_3 زمان انتظار $= 20 - 6 = 14$

P_4 زمان انتظار $= 16 - 6 = 10$

$$\text{میانگین زمان انتظار} = \text{AWT} = \frac{9+10+14+10}{4} = \frac{43}{4} = 10.75$$

$$\text{میانگین زمان اجرا} = \text{AST} = \frac{5+8+6+6}{4} = \frac{25}{4} = 6.25$$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

17 = 6.25 + 10.75

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	0	5	9	14
P_2	3	8	10	18
P_3	4	6	14	20
P_4	11	6	10	16

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا
17 10.75 6.25

۸- گزینه () صحیح است.

الگوریتم LRU (Least Recently Used)

ایده اصلی این الگوریتم این است که اگر صفحه‌ای در چند دستور اخیر مراجعات زیادی داشته است، به احتمال قوی در دستورات بعدی هم ارجاعات زیادی خواهد داشت، همچنین اگر یک صفحه، اخیرا هیچ مراجعه‌ای نداشته، احتمالاً در آینده نزدیک هم ارجاعی نخواهد داشت. در واقع این الگوریتم بیان می‌کند هنگام وقوع خطای نقص صفحه، صفحه‌ای را حذف کنید که طولانی‌ترین زمان عدم استفاده را دارد.

می‌توان گفت LRU تقریبی از الگوریتم بهینه می‌باشد که در آن به جای توجه به آینده، به گذشته توجه می‌شود.

khalilifar.ir

توجه: الگوریتم LRU ناهنجاری بی‌لیدی ندارد، بنابراین با افزایش تعداد قاب، همواره، نقص صفحه کاهش می‌یابد.

اگر 4 قاب در نظر گرفته شود:

نگاه به گذشته ←

رشته مراجعات	1	2	3	4	3	2	2	1	6	7										
قاب 1	1	1	1	1	1	1	1	1	1	1										
قاب 2		2	2	2	2	2	2	2	2	2										
قاب 3			3	3	3	3	3	3	3	7										
قاب 4				4	4	4	4	4	6	6										
نقص صفحه	*	*	*	*					*	*										

در این حالت 6 نقص صفحه به وقوع پیوست. پاسخ سوال در گزینه‌ها نیست.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه سوم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

۹- گزینه (۱) صحیح است.

در ویراست دهم سال 2018 کتاب مرجع آبراهام سیلبرشاتز صفحه 458 الگوریتم‌های Look و C-Look از کتاب حذف شده‌اند. و از SCAN algorithm به صورت elevator algorithm به معنی «**الگوریتم آسانسور**» یاد شده است. به متن صریح کتاب در ادامه توجه نمایید:

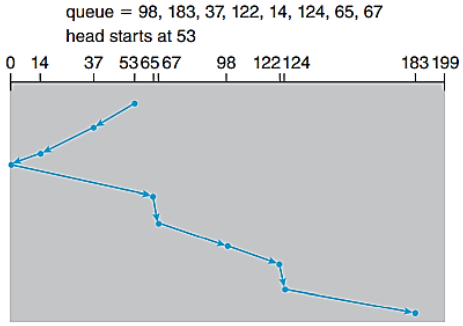
SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

در الگوریتم SCAN ، بازوی دیسک از یک انتهای دیسک شروع کرده و در جهت انتهای دیگر حرکت می کند و به درخواست هایی که در هر سیلندر به آنها می‌رسد، سرویس دهی می کند، تا زمانی که به انتهای دیگر برسد. در انتهای دیگر، جهت حرکت هد، برعکس می شود و سرویس دهی ادامه می یابد. هد به صورت پیوسته به عقب و جلو در عرض دیسک، حرکت می کند. الگوریتم SCAN گاهی اوقات **الگوریتم آسانسور (elevator algorithm)** نامیده می‌شود.

به مثال زیر از متن کتاب سیستم عامل آبراهام سیلبرشاتز دقت نمایید:

khalilifar.ir



SCAN disk scheduling

توجه: در این مورد کتاب اندرو تنن‌بام با کتاب آبراهام سیلبرشاتز اختلاف نظر دارد و جناب تنن‌بام به الگوریتم Look می‌گوید آسانسور، در چند مورد اختلافی حتی تست کنکور حذف شد.

توجه: در صورت سوال طراح محترم شماره ابتدا و انتهای دیسک را مطرح نکرده است، که به ناچار می‌بایست در حل این سوال الگوریتم آسانسور را همان الگوریتم Look در کتاب تنن‌بام در نظر گرفت.

توجه: در کتاب اندرو تنن‌بام الگوریتم آسانسور (Look) به صورت زیر تعریف شده است:

در الگوریتم آسانسور (Look)، هد تا جایی حرکت می‌کند که درخواستی وجود داشته باشد و سپس جهت آن بالعکس می‌شود. به عبارت دیگر در الگوریتم آسانسور (Look) حرکت هد لزوما تا انتهای دیسک و تا آخرین شیار ادامه نمی‌یابد. بلکه فقط تا آخرین درخواست جلو می‌رود و بعد از آخرین درخواست تغییر جهت داده و به سمت درخواست‌های دیگر حرکت می‌کند. در واقع در این الگوریتم کار از اولین درخواست شروع شده و به آخرین درخواست ختم می‌گردد.

توجه: مطابق فرض سوال بازو روی سیلندر 35 قرار دارد و جهت حرکت آن به سمت شیارهای بزرگتر با شماره بزرگتر است. و مطرح شده است که اگر زمان انتقال از یک سیلندر به بعدی 5ms باشد و از الگوریتم آسانسور برای دستیابی به سیلندرها استفاده شود، کل زمان جستجو برای دستیابی به سیلندرهای زیر (به ترتیب از چپ به راست) چند میلی ثانیه است؟

34, 45, 39, 80, 12, 25, 44

مطابق فرض سؤال دنباله درخواست‌ها به صورت زیر است:

34, 45, 39, 80, 12, 25, 44

و هد در ابتدای کار بر روی سیلندر 35 قرار دارد.

در این روش ابتدا درخواست 39 سرویس داده می‌شود. زیرا نزدیکترین درخواست به مکان فعلی هد در جهت صعودی است، سپس درخواست‌های 44، 45، 80 در ادامه پیمایش تغییر جهت هد صورت می‌گیرد و سپس درخواست‌های 34، 25 و در نهایت 12 سرویس می‌گیرند.

یعنی داریم:

$$35 \xrightarrow[4]{} 39 \xrightarrow[5]{} 44 \xrightarrow[1]{} 45 \xrightarrow[35]{} 80 \xrightarrow[46]{} 34 \xrightarrow[9]{} 25 \xrightarrow[13]{} 12$$

khalilifar.ir

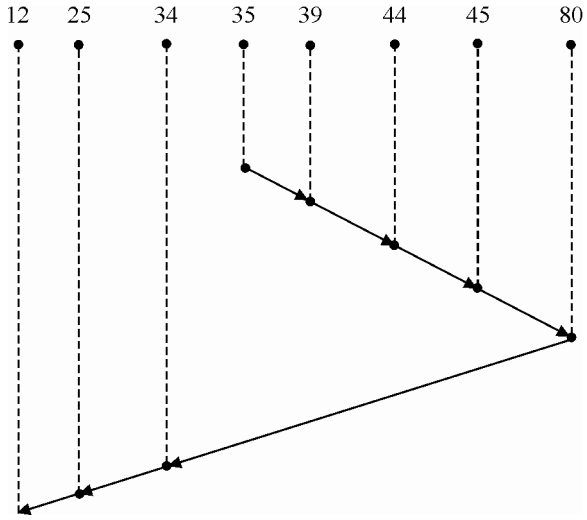
در نتیجه، تعداد کل حرکات هد به صورت زیر خواهد بود:

4+5+1+35+46+9+13 = 113

مطابق فرض سؤال، اگر حرکت از هر سیلندر به سیلندر دیگر (سیلندرهای متوالی) یعنی مجـاور 5 میلی‌ثانیه طول بکشد، مجموع زمان جستجو در این درخواست‌ها، به صورت زیر خواهد بود:

$T_{Total} = 113 \times 5 = 565 \text{ ms}$

شکل زیر گویای مطلب است:



$(80-35)+(80-12) = 45+68 = 113$

$T_{Total} = 113 \times 5 = 565 \text{ ms}$

بنابراین پُرواضح است که گزینه اول پاسخ سوال است.

۱۰- گزینه (۱) صحیح است.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کـردن مقـدار شـمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
Add  this process to s.queue;
block ();
  }
}
```

khalilifar.ir

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if (s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر اینصورت، فرآیند، وارد خط بعدی می‌شود.

تابع signal(s) : عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Signal (semaphore s)
{
s.count = s.count + 1
   if (s.count< = 0)
   {
    Remove a process from queue;
    Wake up ();
     }
}
```

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به خط بعدی است که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up() بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

khalilifar.ir

ترتیب خروجی گزینه اول امکان پذیر نیست:

خروجی	S1	دستور	فرآیند
A	0	Print "A"	P1
A	-1	wait (S1)	P1
فرآیند P1 در سطر دوم توسط دستور wait (S1) در صف سمافور S1 می‌خوابد.			
AC	-1	Print "C"	P2
AC	0	signal (S1)	P2
فرآیند P1 در سطر چهارم توسط دستور signal (S1) بیدار می‌شود، و دقیقا دستور بعد از wait (S1) یعنی Print "B" در صف آماده قرار می‌گیرد و می‌تواند اجرا شود.			
ACB	0	Print "B"	P1
ACBD	0	Print "D"	P2

نتیجه: در این سوال اجرای دستور Print "B" در فرآیند P1 بدون اجرای دستور signal (S1) در P2 و به تبع اجرای دستور Print "C" امکان پذیر نیست. به عبارت دیگر Print "C" همواره باید قبل از اجرای Print "B" باشد.

ترتیب خروجی گزینه دوم امکان پذیر است:

خروجی	S1	دستور	فرآیند
A	0	Print "A"	P1
AC	0	Print "C"	P2
AC	1	signal (S1)	P2
ACD	1	Print "D"	P2
ACD	0	wait (S1)	P1
ACDB	0	Print "B"	P1

ترتیب خروجی گزینه سوم امکان پذیر است:

خروجی	S1	دستور	فرآیند
C	0	Print "C"	P2
C	1	signal (S1)	P2
CA	1	Print "A"	P1
CA	0	wait (S1)	P1
CAB	0	Print "B"	P1
CABD	0	Print "D"	P2

khalilifar.ir

ترتیب خروجی گزینه چهارم امکان پذیر است:

فرآیند	دستور	S1	خروجی
P2	Print "C"	0	C
P2	signal (S1)	1	C
P1	Print "A"	1	CA
P2	Print "D"	0	CAD
P1	wait (S1)	0	CAD
P1	Print "B"	0	**CADB**

بنابراین پُرواضح است که گزینه اول پاسخ سوال است.

۱۱- گزینه (۴) صحیح است.

ساختارهای درونی طراحی سیستم عامل‌ها به صورت زیر است:

۱- ساختار یکپارچه (Monolithic)

ساختار سیستم عامل‌های یکپارچه به این صورت است که هیچ ساختاری ندارد! سیستم‌های یکپارچه به صورت مجموعه‌ای از رویه‌ها نوشته شده‌اند که هر یک می‌توانند دیگری را به هنگام نیاز فراخوانی کرده و برای انجام محاسبات خود از آن‌ها کمک بگیرند. هر یک از رویه‌ها دارای یک واسط شامل پارامترهای ورودی و خروجی است که به سادگی تعریف شده‌اند. همانطور که مشخص است این سیستم هیچ‌گونه نظم، ترتیب، دسته‌بندی و سلسله مراتب خاصی ندارد.

برای ساختن یک سیستم یکپارچه (ساختن object واقعی سیستم عامل) ابتدا باید تمام رویه‌ها در قالب تعدادی فایل و یا به صورت جداگانه کامپایل شوند و سپس با استفاده از یک پیونددهنده (Linker) به هم متصل شده و در فایل object نهایی قرار گیرند. در این ساختار تمام رویه‌ها همدیگر را می‌بینند. این به آن معنی است که بین رویه‌ها هیچ تعیین سطحی وجود ندارد و اصطلاحاً سیستم تخت است. در واقع برای مخفی کردن اطلاعات و محصورسازی (Encapsulation)، محدویت‌هایی وضع نشده است و از آنجاکه دسترسی به هر رویه‌ای امکان‌پذیر است، لذا حفاظت وجود ندارد. پنهان‌سازی اطلاعات (information hiding) نتیجه پیمانه‌ای کردن (Modularity) است. به بیان دیگر شرط لازم برای برقراری پنهان‌سازی اطلاعات، پیمانه‌ای کردن است و شرط کافی برای برقراری پنهان‌سازی اطلاعات تعریف متغیرهای محلی و دستورالعمل‌های مرتبط با تابع است. در یک بیان ساده پنهان‌سازی اطلاعات می‌گوید بخشی از نرم‌افزار در داخل یک پیمانه (تابع) محصور شود. هدف از پنهان‌سازی اطلاعات، پنهان کردن روال انجام دستورات تابع و متغیرهای محلی در پس واسط یا بلاک تابع است. استفاده‌کنندگان پیمانه‌ها (توابع) نیازی به دانستن جزئیات داخلی پیمانه‌ها (توابع) ندارند.

khalilifar.ir

توجه: تعریف متغیر سراسری در تابع ناقض اصل پنهان‌سازی اطلاعات است.

اما به هر حال حتی در سیستم‌های یکپارچه حداقل یک تابع اصلی برای فراخوانی توابع و رویه‌های دیگر وجود دارد. در این سیستم برای تقاضای یک سرویس از سرویس‌های سیستم عامل، ابتدا پارامترهای فراخوانی را در محل‌های از پیش تعیین شده مانند رجیسترها یا پشته قرار داده و سپس یک دستورالعمل تله مربوط به سرویس موردنظر اجرا می‌شود. این عمل به فراخوانی هسته یا فراخوانی راهبری معروف است. در واقع با این روش ماشین از مُد کاربر به مُد هسته می‌رود تا جهت انجام سرویس مورد نظر، کنترل در اختیار سیستم عامل قرار گیرد. اکنون با یک مثال چگونگی عملکرد فراخوان‌های سیستمی شرح داده می‌شود. فرض کنید باید فراخوانی زیر توسط برنامه اصلی (برنامه کاربر) انجام شود:

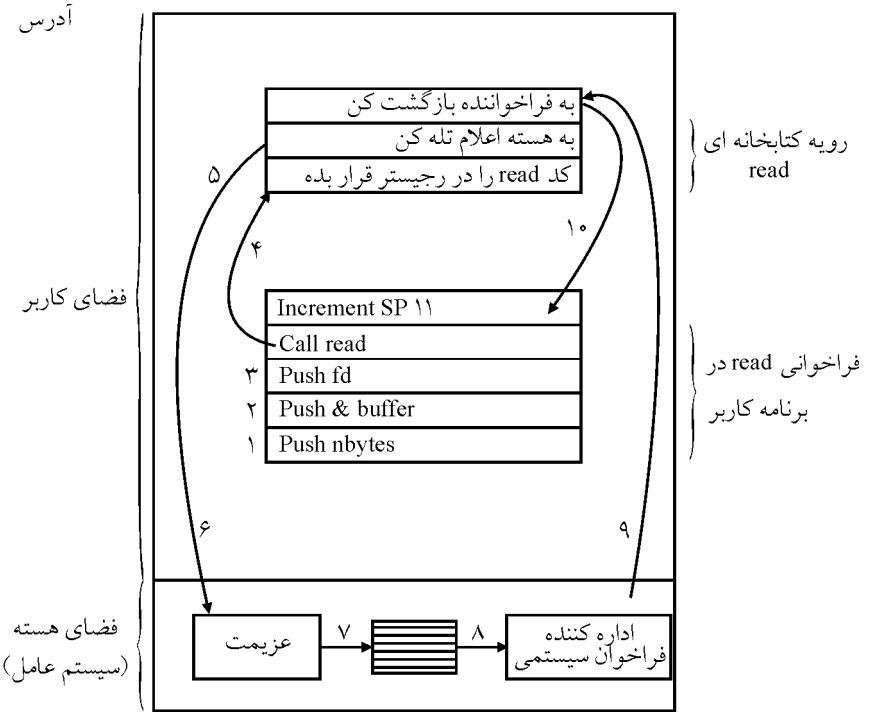
```
Count = Read (fd, buffer, nbyted);
```

تابع Read دارای سه پارامتر است، پارامتر اول مشخص‌کننده فایل است، پارامتر دوم یک اشاره‌گر به بافر است، و سومین پارامتر بیانگر تعدادی بایتی است که باید خوانده شود. این تابع تعداد بایت‌هایی که واقعاً از فایل خوانده و در بافر قرار داده است را برمی‌گرداند. در حالت عادی مقدار Count برابر با nbytes است ولی امکان دارد که کمتر از آن هم باشد (مثلاً به علت اتمام فایل).

در ابتدا برای صدا زدن رویه کتابخانه‌ای Read از کامپایلر، پارامترهای آن در پشته Push می‌شود (در شکل زیر این عمل با توجه به زبان برنامه نویسی C و C++ از پارامتر آخر به اول انجام می‌شود). با توجه به اینکه پارامتر دوم یعنی buffer یک اشاره‌گر است در هنگام push در ابتدای آن یک علامت & استفاده می‌شود، یعنی محتوای آن به عنوان یک آدرس در نظر گرفته شده است. مراحل push کردن پارامترها در شکل زیر با شماره‌های 1 تا 3 نمایش داده شده‌اند. اکنون باید رویه کتابخانه‌ای را فراخوانی کرد. برای این کار از یک دستورالعمل ساده فراخوانی رویه که برای صدا زدن همه رویه‌ها به کار می‌رود، استفاده می‌شود (مرحله 4). شماره فراخوانی سیستمی باید در محلی مانند یک رجیستر (که سیستم عامل انتظار دارد و در آنجا باشد) قرار گیرد (مرحله 5). اکنون باید دستور TRAP یا همان وقفه نرم‌افزاری (تله) رخ دهد تا کنترل اجرا از مُد کاربر به مُد هسته رفته و اجرای کُد مربوط به آن فراخوان از یک آدرس ثابت درون هسته آغاز شود (مرحله 6). اکنون هسته، شماره فراخوانی که در یک رجیستر گرفته بود را بررسی می‌کند و سپس به سراغ یک جدول که حاوی اشاره‌گرهایی به اداره‌کننده‌های فراخوانی‌های سیستمی (Systemcall Handler) هستند، می‌رود و با استفاده از شماره فراخوانی از بین درایه‌های جدول، محل قرار گرفتن اداره‌کننده فراخوانی مذکور (Read) را می‌یابد که عموماً محلی از حافظه است (مرحله 7) و آن را به اجرا درمی‌آورد (مرحله 8). در ادامه، اجرای اداره‌کننده فراخوانی سیستمی به طور کامل تمام می‌شود یا ممکن است تمام نشود و کنترل به فرآیند جدید دیگری منتقل شود و به تبع مراحل 1 تا 6 رویه کتابخانه‌ای در فضای کاربر برای فرآیند جدید هم اجرا شود. اینکه گفته می‌شود ممکن است به دلیل آن است که اگر فراخوان سیستمی در فرآیند جاری مجبور به انتظار

khalilifar.ir

باشد، مانند اینکه فراخوان سیستمی تلاش کند از صفحه کلید بخواند ولی هنوز چیزی از طرف کاربر تایپ نشده باشد، از آنجا فراخوان سیستمی منتظر و مسدود شده داخل فرآیند جاری است بنابراین فرآیند جاری از طرف سیستم عامل به صف مسدود و منتظر منتقل می‌شود. در این شرایط سیستم عامل به صف فرآیندهای آماده اجرا نگاه می‌کند تا ببیند آیا فرآیند دیگری می‌تواند پس از آن اجرا شود یا خیر. در صورت وجود فرآیند آماده جدید سیستم عامل تعویض متن انجام داده و پردازنده را در اختیار فرآیند جدید قرار می‌دهد. در ادامه هرگاه ورودی مورد نظر فراخوان سیستمی فرآیند قبل آماده شود، برای مثال کلیدی فشرده شود، فرآیند قبلی به صف آماده سیستم عامل منتقل می‌شود و ممکن است در دفعه بعد توسط زمان‌بند کوتاه مدت سیستم عامل جهت اجرا انتخاب شود و مراحل 9 تا 11 انجام می‌شود. (مرحله 9) سپس این رویه به روش بازگشت به برنامه کاربر بازمی‌گردد. (مرحله 10) برای خاتمه کار، برنامه کاربر پشته را پاک‌سازی می‌کند، همان‌گونه که پس از بازگشت از همه رویه‌ها این عمل صورت می‌گیرد. (مرحله 11)



ساختار کلی این سیستم عامل‌ها از سه بخش زیر تشکیل شده است:

۱- یک برنامه اصلی که می‌تواند رویه سرویس‌های خواسته شده را فراخوانی کند. این برنامه به ازای هر درخواست، یک روال سرویس را صدا می‌زند.

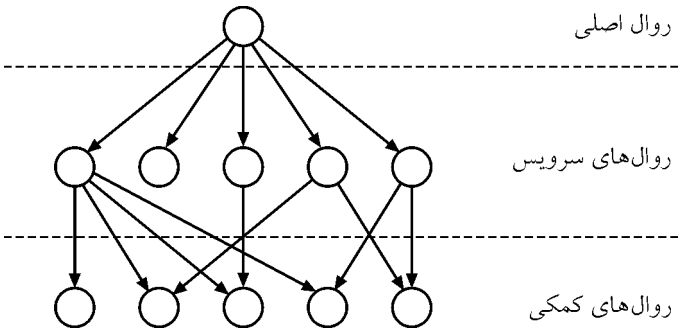
۲- مجموعه‌ای از رویه‌های سرویس که می‌توانند تعدادی فراخوانی سیستمی انجام دهند و یا

khalilifar.ir

تعدادی روال کمکی را فراخوانی کنند.

۳- تعدادی رویه کمکی و سودمند (Utility Procedures) که می‌توانند به رویه‌های سرویس کمک کنند و توسط رویه‌های سرویس فراخوانی شوند.

این مفاهیم در شکل زیر نشان داده شده است:



ساختار سیستم عامل‌های یکپارچه

پیمانه‌ای کردن یعنی تقسیم نرم‌افزار (در اینجا خود برنامه سیستمی سیستم عامل) به چند مولفه جداگانه و متمایز که این مولفه‌ها به عنوان پیمانه‌هایی هستند که از اجتماع آن‌ها، خواسته‌های مساله برآورده می‌شود. با این عمل مدیریت مفهومی یک برنامه حاصل می‌شود و قابلیت خوانایی نرم‌افزار چندین برابر می‌گردد. درک و فهم نرم‌افزار یکپارچه (monolithic) که تنها از یک پیمانه (ماژول) تشکیل شده است، بسیار مشکل است، زیرا تعداد متغیرها، حجم ارجاعات، تعداد مسیرهای کنترلی و پیچیدگی سراسری آن بسیار زیاد است. بنابراین اگر بتوان نرم‌افزار را به پیمانه‌هایی مناسب تقسیم نمود، پیچیدگی و هزینه کلی آن کاهش خواهد یافت. اما تعداد این پیمانه‌ها نباید بیش از حد باشد زیرا هزینه مجتمع‌سازی یا یکپارچه‌سازی و اتصال آن‌ها افزایش می‌یابد.

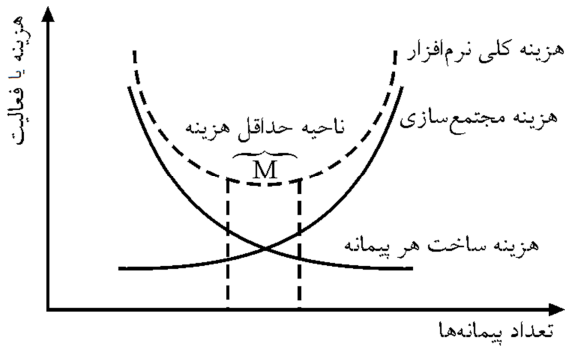
در یک نرم‌افزار با کاهش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها کاهش می‌یابد. اما هزینه ساخت هر پیمانه افزایش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با افزایش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها افزایش می‌یابد. اما هزینه ساخت هر پیمانه کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با داشتن تعداد مناسب پیمانه (ماژول)، همه هزینه‌ها کاهش می‌یابد. زیرا هم هزینه یکپارچه‌سازی و هم هزینه ساخت هر پیمانه، هر دو کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، کاهش می‌یابد.

khalilifar.ir

شکل زیر رابطه تعداد پیمانه‌های نرم‌افزار با هزینه توسعه نرم‌افزار را نشان می‌دهد. در این شکل، یکی از منحنی‌ها هزینه توسعه یک پیمانه را نشان می‌دهد و منحنی دیگر، هزینه یکپارچه‌سازی پیمانه‌ها را نشان می‌دهد. به تبع هزینه کلی نرم‌افزار برابر حاصل جمع این دو منحنی در هر نقطه است، که با منحنی خط‌چین نشان داده شده است.



نمودار فوق نشان می‌دهد که هزینه یا فعالیت لازم برای ساخت پیمانه‌های نرم‌افزاری با افزایش تعداد پیمانه‌ها الزاما کاهش می‌یابد ولی به موازات رشد بیش از حد پیمانه‌ها، هزینه مربوط به اجتماع و یکپارچه‌سازی پیمانه‌ها نیز رشد می‌کند. در واقع همواره با تعداد مناسبی از پیمانه‌ها که با M نشان داده شده است، می‌توان هزینه و کار را کمینه کرد.

استقلال عملیاتی، نتیجه پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است. به بیان دیگر شرط لازم برای برقراری استقلال عملیاتی، پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است و شرط کافی برای برقراری استقلال عملیاتی انسجام بالا و اتصال پایین است. در صورتی که پیمانه‌هایی را با عملکرد تک منظوره(انسجام بالا) و عدم ارتباط بیش از حد با پیمانه‌های دیگر (اتصال پایین) ایجاد کنیم، استقلال عملیاتی تحقق می‌یابد.

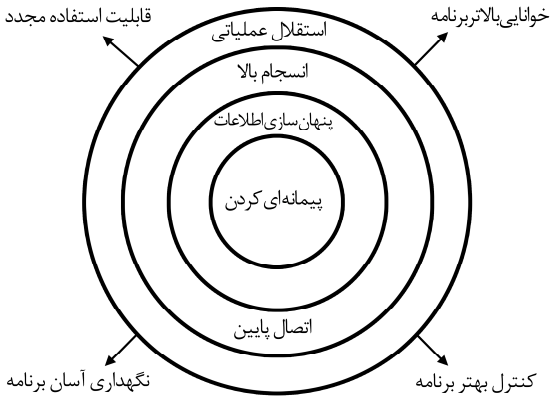
توجه: استقلال عملیاتی خوب، کلید طراحی خوب و طراحی خوب، کلید یک نرم‌افزار با کیفیت می‌باشد. استقلال عملیاتی خوب با دو مفهوم انسجام (Cohesion) و اتصال (Coupling) مورد ارزیابی قرار می‌گیرد. انسجام، معیاری است که توان نسبی کارکردی یک پیمانه را نشان می‌دهد و اتصال، معیاری است که میزان نسبی وابستگی پیمانه‌ها به یکدیگر را نشان می‌دهد.

توجه: در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

توجه: پیمانه‌ای‌کردن، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به **خوانایی بالاتر برنامه، کنترل بهتر برنامه، قابلیت استفاده مجدد** پیمانه‌های (توابع) برنامه جاری در برنامه‌های آتی و **نگهداری آسان برنامه** جاری می‌گردد.

khalilifar.ir

شکل زیر گویای مطلب است:



توجه: مهمترین مزیت سیستم‌های یکپارچه، سرعت نسبتاً بالای آنهاست، زیرا معمولاً هر تابع و روالی می‌تواند بدون محدودیت توابع دیگر را فراخوانی کند. اما در مقابل معایب اساسی زیادی نسبت به ساختارهای سیستم عامل‌های بعدی دارد که می‌توان از آن‌ها (نقیض آن‌ها) به عنوان معیار سیستم عامل خوب نیز یاد کرد. این معایب عبارتند از:

۱- عدم وجود (یا ناچیز بودن) طراحی پیمانه‌ای (Modularity).

۲- عدم انعطاف‌پذیری (Flexibility) یعنی برای اینکه بتوان آن را با عملکردهای جدید تطبیق داد باید تغییرات اساسی میان رویه‌های آن ایجاد کرد.

۳- پیچیدگی نگهداری (Maintainability)

۴- پیچیدگی در اشکال‌زدایی (Debug) و ترمیم (Recovery).

۵- پیچیدگی پیاده‌سازی.

۶- قابلیت اطمینان (Reliability) کم (به علت پیچیدگی و درهم ریختگی ساختار آن).

۲- ساختار لایه‌ای (Layered)

در این ساختار، سیستم عامل به صورت چند لایه طراحی می‌شود که هر لایه بر روی دیگری قرار گرفته است و می‌توان هر لایه را مستقل از لایه دیگر طراحی کرد و گسترش داد. هر لایه در این ساختار به لایه بالاتر از خود سرویس داده و جزئیات کار را از دید آن مخفی می‌سازد. به این ترتیب هر لایه می‌تواند از توابع و سرویس‌های لایه پایین‌تر استفاده کند بدون اینکه بداند این سرویس چگونه پیاده‌سازی شده است. تنها باید بداند هر سرویس چه می‌کند و چگونه و از طریق چه واسط‌هایی می‌توان به آن سرویس دسترسی پیدا کرد.

اولین سیستمی که به این روش طراحی شد، سیستم THE بود که توسط دِکسترا و دانشجویانش در سال 1968 ساخته شد. این سیستم دارای شش لایه بود که در شکل زیر نمایش داده شده است:

khalilifar.ir

لایه	وظیفه
5	اپراتور
4	برنامه‌های کاربردی
3	مدیریت ورودی و خروجی
2	ارتباط فرآیند – اپراتور
1	مدیریت حافظه اصلی و جانبی (drum)
0	تخصیص پردازنده و چندبرنامگی

ساختار سیستم عامل THE

در ادامه شرح مختصری برای این لایه‌ها بیان می‌شود:

لایه صفر (تخصیص پردازنده و چندبرنامگی): وظیفه این لایه تعویض متن در زمان وقوع وقفه یا پایان برش زمانی می‌باشد.

لایه یک (مدیریت حافظه اصلی و جانبی): این لایه فضایی از حافظه اصلی و همچنین بخشی به اندازه 512 هزار کلمه بر روی drum را به فرآیندها تخصیص می‌دهد. از drum برای نگهداری بخش‌هایی (صفحه‌هایی) از اطلاعات مورد نیاز فرآیندها که در حافظه اصلی به علت نبودن فضای کافی جا نگرفته‌اند استفاده می‌شود تا در صورت لزوم با داده‌های داخل حافظه اصلی در فضای آدرس خود فرآیند، جابه‌جا شوند.

لایه دو (ارتباط فرآیند و اپراتور): وظیفه این لایه برقراری ارتباط بین فرآیند و کنسول اپراتور است، یعنی در بالای این لایه، هر فرآیند به کنسول اپراتور مخصوص به خودش متصل می‌شود.

لایه سه (مدیریت ورودی و خروجی): وظیفه این لایه مدیریت دستگاه‌های I/O و بافرکردن جریان اطلاعات مربوطه می‌باشد.

لایه چهار (برنامه‌های کاربردی): در این لایه برنامه‌های کاربردی قرار می‌گیرند که از نظر نیازهای تخصیص پردازنده، مدیریت حافظه، ارتباط با کنسول و I/O توسط لایه‌های زیرین تأمین شده‌اند.

لایه پنج (اپراتور): فرآیند اپراتور سیستم در این لایه قرار دارد.

مزیت‌های سیستم‌های لایه‌ای:

۱- طراحی پیمانه‌ای (Modularity) مناسب.

توجه: مزیت اصلی روش لایه ای، پیمانه‌ای بودن (Modularity) است، یعنی لایه‌ها به نحوی تقسیم‌بندی می‌شوند که هر لایه فقط توابع و سرویس‌های لایه‌های پایین‌تر را استفاده می‌کند. بدین ترتیب هر لایه را را می‌توان مستقل از لایه‌های دیگر و به راحتی طراحی کرد، خطایابی و نگهداری کرد و توسعه داد.

۲- انعطاف‌پذیری (Flexibility) مناسب در برابر تغییرات.

khalilifar.ir

۳- قابلیت نگهداری (Maintainability) مناسب.
۴- سادگی در اشکال‌زدایی (Debug) و ترمیم (Recovery).
۵- سادگی در پیاده‌سازی.
۶- قابلیت اطمینان (Reliability) مناسب. (توانایی در محافظت لایه‌ها از یکدیگر)

معایب سیستم‌های لایه‌ای:
۱- کاهش کارایی
توجه: یک نقض سیستم‌های لایه‌ای این است که ممکن است قدری نسبت به سیستم‌های دیگر کند به نظر برسند و به تبع منجر به «**کاهش کارایی**» گردد، زیرا دستورات و فراخوانی‌ها از لایه بالا به سمت لایه پایین حرکت می‌کنند و زمان زیادی برای این روال صرف می‌شود. در واقع هر لایه قدری سربار (Overhead) به دستورات و فراخوانی‌ها می‌افزاید، در نتیجه یک فراخوانی سیستمی در این ساختار، زمان بیشتری نسبت به ساختار غیر لایه ای صرف می‌کند. برای رفع این نقص سعی می‌شود تعداد لایه‌های کمتری با قابلیت عمل بیشتری طراحی شود. به عنوان مثال محصول اولیه Windows NT با لایه‌های زیاد، کارایی کمتری نسبت به ویندوز 95 داشت. در NT 4.0 سعی شد لایه‌ها به همدیگر نزدیکتر و مجتمع‌تر شوند تا کارایی بیشتر گردد.
۲- نیاز به دقت بالا در پیمانه‌ای کردن و تقسیم‌بندی لایه‌ها دارد، مسأله اصلی در سیستم‌های لایه‌ای، تعریف مناسب هر لایه و سرویس‌های درون آن می‌باشد. از آنجا که هر لایه فقط می‌تواند از سرویس‌های لایه پایین‌تر استفاده کند، در طراحی لایه‌ها باید دقت بسیار به خرج داد.

۳- ساختار ماشین مجازی (Virtual Machine)

در این ساختار، یک سیستم عامل بر روی سخت‌افزار اجرا شده و برای لایه بالاتر چندین ماشین مجازی با تمام جزئیات سخت‌افزاری فراهم می‌کند که بر روی هر یک از این ماشین‌های مجازی می‌توان یک سیستم عامل جداگانه نصب و اجرا کرد.
این ماشین مجازی (که پایین‌ترین لایه را برای لایه‌های بالاتر فراهم می‌کند) فقط یک ماشین توسعه یافته (مثلاً با سیستم فایل و دیگر امکانات پیشرفته) نیست، بلکه کپی دقیقی از سخت‌افزار (شامل همه قسمت‌هایی که یک سخت‌افزار واقعی دارد) است.
توجه: جهت درک چگونگی پیاده‌سازی سیستم عامل‌های ماشین مجازی به شکل زیر دقت کنید:

فرآیندها	فرآیندها	فرآیندها
هسته سیستم عامل 1	هسته سیستم عامل 2	هسته سیستم عامل 3
ماشین مجازی		
سخت‌افزار		

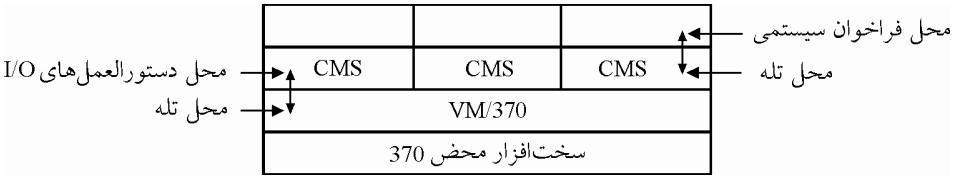
مدل ماشین مجازی

فرآیندها
هسته سیستم عامل
سخت افزار

مدل ماشین غیر مجازی

khalilifar.ir

یک سیستم اشتراک زمانی دو وظیفه عمده و مجزا دارد:
۱- مدیریت منابع (چندبرنامگی و مالتی‌پلکس کردن پردازنده و سایر منابع)
۲- مدیریت واسط کاربر (یک ماشین توسعه‌یافته با استفاده از یک واسط که بر روی سخت‌افزار خام قرار گیرد.)
اما ایده ساختار ماشین مجازی این است که دو وظیفه فوق از یکدیگر جدا شوند، سیستم عامل VM/370 بر روی سیستم‌های IBM بهترین مثال از مفهوم ماشین مجازی است. قلب سیستم که به نام مانیتور ماشین مجازی (VMM یا Virtual machin manitor) یا Hypervisor معروف است بر روی سخت‌افزار عریان اجرا شده و فقط وظیفه اول یعنی مدیریت منابع (چندبرنامگی و مالتی‌پلکس کردن پردازنده و سایر منابع) را انجام می‌دهد و اکنون می‌توان یک یا چند ماشین مجازی را بر روی لایه بالاتر از آن قرار داد، انگار که یک ماشین فیزیکی کامل، به چندین نسخه ماشین مجازی کامل تکثیر شده است. البته باید توجه داشته باشید که ماشین‌های مجازی برخلاف سیستم‌های عامل یک ماشین توسعه‌یافته نیستند بلکه هر کدام یک نسخه دقیق از سخت‌افزار خام هستند که شامل تمام خصوصیات سخت‌افزار مانند مد کاربر، مد هسته، I/O و وقفه می‌باشند. این ماشین مجازی انگار که همان وظیفه دوم یعنی مدیریت واسط کاربر است، اما فقط در حد یک سخت افزار خشک و خالی است. اکنون می‌توان بر روی هر کدام از این ماشین‌های مجازی یک سیستم عامل معمولی نصب نمود، این عمل همان عمل نصب سیستم عامل بر روی یک سخت‌افزار خام حقیقی است. شکل زیر ساختار VM/370 را نشان می‌دهد.



توجه: هر کاربر یک برنامه (CMS: Conversational sational monitor system) مخصوص به خود را دارد که یک سیستم عامل تک کاربره محاوره‌ای است.
توجه: سیستم عامل‌هایی که بر روی ماشین‌های مجازی اجرا می‌شوند، هیچ‌گونه ارتباطی با یکدیگر ندارند، در واقع هر کدام خود را در یک سیستم مجزا تصور می‌کنند بنابراین می‌توان بر روی هر ماشین مجازی یک نوع سیستم عامل مجزا نصب نمود.
همانطور که مشخص است هر ماشین مجازی باید از دیگران کاملاً مستقل باشد یا حداقل اینطور تصور کند. حال سوال این است که این استقلال چگونه بر روی یک سخت‌افزار مشترک پیاده‌سازی می‌شود؟ جواب تسهیم (Multiplexing) و Slooping می‌باشد.
در واقع کافی است منابع سیستم با توجه به خصوصیاتش به دو صورت زمانی یا فضایی بین ماشین‌های مجازی تسهیم و یا Spool شوند. به عنوان مثال:

khalilifar.ir

- ایجاد پردازنده مجازی با تسهیم زمانی پردازنده حقیقی که این عمل به کمک PCBهای مجازی انجام می‌شود.
- ایجاد حافظه‌های مجازی با تسهیم فضای حافظه اصلی بین ماشین‌های مجازی.
- Spool کردن دستگاه‌های ورودی و خروجی.
- تشکیل دیسک مجازی با تسهیم فضای دیسک حقیقی.

همانطور که مشخص است دستیابی به سخت‌افزار و فراخوان‌های سیستمی نمی‌تواند بر روی CMS اجرا شود و در واقع CMS قدرت این کار را ندارد، پس برنامه اجرا شده در CMS چگونه فراخوانی سیستمی انجام می‌دهد؟ فرض کنید برنامه‌ای که روی CMS اجرا می‌شود قصد انجام عمل I/O دارد و یک فراخوانی سیستمی صادر می‌کند. این فراخوانی سیستمی باعث تغییر مد کاربر به مد هسته و می‌شود و سیستم عامل روی CMS (از آنجا که نمی‌داند CMS سخت‌افزار حقیقی نیست) سعی می‌کند که این عمل را روی CMS اجرا کند و در واقع یک تله مجازی در CMS رخ می‌دهد، اکنون CMS که در حقیقت یک نرم‌افزار است، این درخواست را به VM/370 ارجاع می‌دهد و اکنون یک تله واقعی در VM/370 رخ می‌دهد و درخواست (با توجه به تسهیم یا Spool) به اجرا رسیده و پاسخ به لایه‌های بالاتر ارجاع می‌شود. از آنجا که وظیفه چندبرنامگی و ارائه ماشین توسعه یافته کاملاً مجزا از یکدیگر هستند، هر یک از این تکه برنامه‌ها بسیار ساده‌تر شده و از انعطاف‌پذیری (Flexibility) بیشتر و قابلیت نگهداری (Maintainability) آسان‌تری برخوردار هستند.

توجه: در ساختار ماشین مجازی دو وظیفه اصلی چندبرنامگی و ایجاد واسط راحت (مستقل از سخت افزار) از یکدیگر مجزا شده‌اند. مانیتور ماشین مجازی وظیفه چندبرنامگی را بر عهده دارد و لایه بالای آن وظیفه ایجاد واسط کاربر با سخت افزار را بر عهده دارد. لذا هر یک از این بخش‌ها ساده‌تر شده و از قابلیت انعطاف بیشتری برخوردارند.

توجه: از آنجا که هر ماشین مجازی کاملا مشابه سخت افزار واقعی است، هر یک از آنها می‌توانند هر سیستم عاملی را مستقلا اجرا کنند لذا می‌توان همزمان سیستم‌عامل‌های مختلفی را روی این ماشین اجرا کرد.

یکی دیگر از کاربردهای ماشین‌های مجازی به مجازی‌سازی محیط برنامه‌نویسی (Programming-environment Virtualization) موسوم است، در این روش VMMها سخت افزار واقعی را جهت اجرای سیستم‌عامل‌های متعدد نسخه‌برداری و مجازی‌سازی نمی‌کنند، بلکه هدف ایجاد قابلیت حمل (Portability) برنامه‌ها بر روی سخت‌افزار و سیستم‌عامل‌های مختلف است. کامپایلر زبان java که توسط شرکت Sun Microsystem طراحی شده است، یک خروجی بایت کد (byte code) تولید می‌کند. این بایت کدها دستوراتی هستند که بر روی ماشین مجازی جاوا (JVM) اجرا می‌شوند. جهت اجرای برنامه‌های java در یک ماشین، آن کامپیوتر می‌بایست دارای یک JVM باشد. بدین ترتیب برنامه‌هایی که به زبان java نوشته شده‌اند به راحتی بر روی

khalilifar.ir

انواع کامپیوترها اجرا می‌شوند. فقط کافی است بایت کدها را روی آن ماشین کامپایل کرد. بدیهی است به علت نیاز به کامپایل شدن بایت کدها، برنامه‌های جاوا سرعت کمتری نسبت به برنامه‌هایی نظیر C دارد. برنامه‌های C توسط کامپایلر بومی یک کامپیوتر، برای یکبار تبدیل به زبان ماشین آن کامپیوتر می‌گردد. پس خروجی زبان ماشین کامپایلر C از یک نوع کامپیوتر به کامپیوتر دیگر متفاوت است ولی بایت کدهای خروجی java برای همه ماشین‌ها یکسان است.

مزیت‌های سیستم‌های ماشین‌های مجازی:

۱- امکان اجرای چند سیستم عامل بر روی یک ماشین حقیقی (این سیستم عامل ها از یک زیرساخت مشترک پیروی می‌کنند).

۲- استقلال کامل ماشین‌های مجازی قرار گرفته روی یک ماشین حقیقی و امنیت بالای آن‌ها.

۳- با در اختیار داشتن چند ماشین مجازی بدون استفاده از سخت‌افزار اضافی و با هزینه پایین می‌توان نصب و تست سیستم عامل‌های مختلف و حتی شبکه‌های کامپیوتری انجام داد.

۴- در ماشین مجازی java قابلیت حمل (Protability) بالایی وجود دارد.

معایب سیستم‌های ماشین‌های مجازی:

۱- به دلیل ساخت نسخه‌های متعدد از سخت افزار پیچیدگی پیاده‌سازی آن بسیار زیاد است.

۲- به دلیل سربار حاصل از دوبار وقفه (وقفه مجازی و وقفه واقعی) کارایی آن پایین است.

۴- ساختار ریز هسته (Microkernel)

ایده‌ی اصلی در این ساختار، هر چه کوچک‌تر و ساده‌تر نمودن قسمت هسته (Kernel) سیستم عامل است. بنابراین به دلیل اینکه یک هسته بسیار کوچک خواهیم داشت به ساختار **ریز هسته** (Microkernel) **ساختار مشتری – سرویس‌دهنده** (Client - Server) نیز گفته می‌شود.

در سیستم عامل‌های مشتری – سرویس‌دهنده، قسمت اعظم کد سیستم عامل در حالت کاربر اجرا می‌شود و وظیفه هسته برقراری ارتباط میان فرایندهای سرویس‌دهنده و مشتری است. در این ساختار درخواست فرایند کاربر (که به آن فرایند مشتری گویند) به یک فرایند سرویس‌دهنده فرستاده شده و پاسخ آن به مشتری برگشت داده می‌شود.

توجه: امروزه سیستم عامل‌ها به جای داشتن یک هسته بزرگ به سمت داشتن هسته هرچه کوچکتر حرکت می‌کنند که به آن اصطلاحاً **ریز هسته** می‌گویند. در واقع فقط چند عمل اصلی مانند ارتباط بین فرآیندها و زمانبندی پایه‌ای را به هسته واگذار می‌کنند و دیگر خدمات سیستم عامل توسط تعدادی فرآیند سرویس‌دهنده صورت می‌گیرد. این همان ایده‌ی مدل مشتری – سرویس‌دهنده است که به آن **معماری ریز هسته** نیز می‌گویند.

توجه: روند طراحی سیستم عامل‌های جدید همواره با این این ایده همراه بوده است که تا جایی که ممکن است کدها به لایه‌های بالاتر منتقل شوند تا نهایتا یک هسته کمینه پدید آید. برای این

khalilifar.ir

منظور اکثر وظایف سیستم عامل را در سطح کاربر و مشابه فرآیندهای کاربر پیاده‌سازی می‌کنند. برای درخواست یک سرویس، مانند خواندن یک بلوک از فایل، فرآیند کاربر (که اکنون به عنوان فرآیند مشتری شناخته می‌شود) یک درخواست به فرآیند سرویس‌دهنده ارسال می‌نماید و از آن می‌خواهد که کارش را انجام دهد و پاسخ را برگرداند. کار هسته در این مدل برقراری ارتباط بین مشتری‌ها و سرویس‌دهنده از طریق تبادل پیام است. البته هسته، وظایف دیگری نیز دارد. به طور کلی وظایف هسته در این مدل به صورت زیر است:

۱- تبادل پیام مابین مشتری‌ها و سرویس‌دهنده‌ها

۲- زمان‌بندی پردازنده، فرآیندها و ایجاد چندبرنامگی

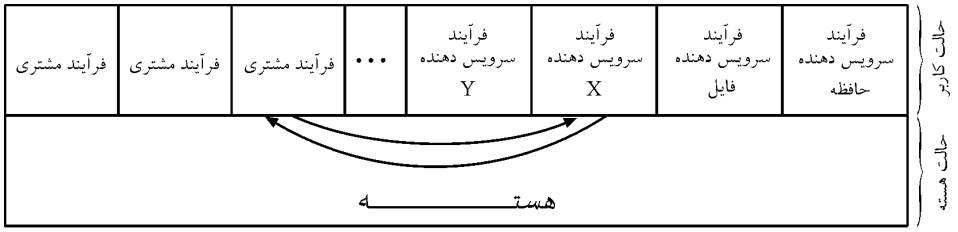
۳- بخش سطح پایین مدیریت حافظه مانند برنامه‌ریزی ثبات‌های سخت‌افزار مدیریت حافظه

۴- بخش سطح پایین مدیریت I/O که برنامه‌ریزی ثبات‌های ویژه کنترل کننده‌ها و کارهایی را بر عهده دارند که اگر در سطح کاربر انجام شود، آنگاه امنیت کل سیستم به مخاطره خواهد افتاد.

توجه: در این روش، سیستم عامل به چند بخش (خدمات فایل، خدمات فرآیند، خدمات ترمینال و خدمات حافظه) تقسیم شده است، که هریک به عنوان یک سرویس‌دهنده فقط یکی از وظایف سیستم عامل را انجام می‌دهد، بنابراین سیستم عامل را می‌توان ساده‌تر تحلیل، طراحی، پیاده‌سازی و نگهداری کرد.

توجه: سرویس‌دهنده‌ها در مُد کاربر اجرا شده و ریز هسته با آنها همانند فرآیندهای کاربر رفتار می‌کند. بنابراین هیچ یک از سرویس‌دهنده‌ها دسترسی مستقیم به سخت‌افزار را ندارد. همچنین اگر کارکرد یکی از سرویس‌دهنده‌ها مختل شود، فقط همان بخش مختل می‌شود و این اختلال به دلیل استقلال عملیاتی هر سرویس‌دهنده به سایر سرویس‌دهنده‌ها و به تبع کل سیستم منتقل نمی‌شود.

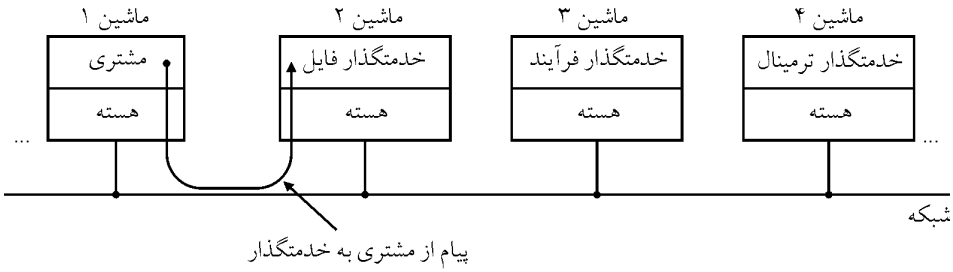
توجه: نحوه ی پیاده‌سازی و عملکرد مدل مشتری – سرویس‌دهنده در شکل زیر نشان داده شده است:



ساختار مشتری – سرویس‌دهنده

در حالت دیگر می‌توان از مدل مشتری و سرویس‌دهنده در سیستم‌های توزیع شده استفاده نمود. به شکل زیر توجه کنید.

khalilifar.ir



با توجه به شکل فوق فرض کنید، مشتری بر روی یک ماشین مجزا از سرویس‌دهنده اجرا شده است، در این حالت همانطور که در مدل مشتری و سرویس دهنده ماشین محلی، یک پیام درخواست از مشتری به سرویس‌دهنده از طریق هسته ارسال می‌شد و سرویس‌دهنده پاسخ می‌داد، در سیستم توزیع شده هم این پیام از مشتری به هسته ماشین مشتری و از آن به یک شبکه ارتباطی و از آن به هسته ماشین سرویس‌دهنده و سپس به سرویس‌دهنده انتقال می‌یابد و پاسخ هم همین مسیر را باز می‌گردد. نکته‌ای که در این روش وجود دارد این است که برای مشتری و یا سرویس‌دهنده بین پیامی که از ماشین محلی آن‌ها می‌آید با پیامی که از ماشین راه دور دیگر می‌آید، تفاوتی وجود ندارد (و در واقع برای آن‌ها قابل تشخیص نیست). به عبارت دیگر از آنجا که یک مشتری به وسیله ارسال پیام‌هایش با یک سرویس‌دهنده ارتباط برقرار می‌کند، مشتری نیاز ندارد که بداند آیا به پیغام وی به صورت محلی در ماشین خودش رسیدگی می‌شود و یا اینکه پیغام از طریق شبکه به یک ماشین دور ارسال می‌شود.

مزیت‌های سیستم‌های مشتری و سرویس‌دهنده:

مهمترین مزیت سیستم‌های مشتری و سرویس‌دهنده طراحی پیمانه‌ای عالی آن‌ها می‌باشد. از این رو که وظایف سیستم عامل به طور مفصل و منسجم از هم جدا شده و هر کدام در یک پیمانه قرار گرفته‌اند، مزیت پیمانه‌ای بودن این روش باعث بروز خصوصیات زیر می‌شود:

۱- انعطاف‌پذیری (Flexibility) بالا

۲- قابلیت نگهداری (Maintainability)، اشکال‌زدایی (Debug) و ترمیم (Recovery) بالا

۳- قابلیت اطمینان (Reliability) بالا

۴- مقیاس‌پذیری (قابلیت توسعه) بالا

۵- سهولت پیاده‌سازی

۶- مناسب بودن برای سیستم‌های شبکه‌ای و توزیعی

معایب سیستم‌های مشتری و سرویس‌دهنده:

در مقابل همین خاصیت پیمانه‌ای بودن و جدا کردن سرویس‌دهنده‌ها از مشتری‌ها باعث پایین آمدن کارایی (سرعت) این سیستم نسبت به سیستم‌های دیگر شده است. فرض کنید که مشتری

khalilifar.ir

اقدام به ارسال یک پیام درخواست می‌کند این پیام در حین ارسال توسط مشتری و دریافت توسط سرویس‌دهنده باعث وقوع 2 تله می‌شود و پاسخ آن نیز به همین صورت، بنابراین برای هر درخواست و پاسخ 4 تله رخ می‌دهد، 2 فراخوان سیستمی برای ارسال و دریافت «**پیام درخواست**» و همچنین 2 فراخوان سیستمی برای ارسال و دریافت «**پیام پاسخ**». همچنین در موازات آن فراخوانی‌های سیستمی برای مدیریت حافظه و I/O نیز رخ می‌دهد که کارایی را تا حد زیادی کاهش می‌دهد.

۱۲- گزینه (۱) صحیح است.

ساختار ریز هسته (Microkernel)

ایده‌ی اصلی در این ساختار، هر چه کوچک‌تر و ساده‌تر نمودن قسمت هسته (Kernel) سیستم عامل است. بنابراین به دلیل اینکه یک هسته بسیار کوچک خواهیم داشت به ساختار **ریز هسته** (Microkernel) **ساختار مشتری - سرویس‌دهنده** (Client - Server) نیز گفته می‌شود.

در سیستم عامل‌های مشتری – سرویس‌دهنده، قسمت اعظم کد سیستم عامل در حالت کاربر اجرا می‌شود و وظیفه هسته برقراری ارتباط میان فرایندهای سرویس‌دهنده و مشتری است. در این ساختار درخواست فرایند کاربر (که به آن فرایند مشتری گویند) به یک فرایند سرویس‌دهنده فرستاده شده و پاسخ آن به مشتری برگشت داده می‌شود.

توجه: امروزه سیستم عامل‌ها به جای داشتن یک هسته بزرگ به سمت داشتن هسته هرچه کوچکتر حرکت می‌کنند که به آن اصطلاحاً **ریز هسته** می‌گویند. در واقع فقط چند عمل اصلی مانند ارتباط بین فرآیندها و زمانبندی پایه‌ای را به هسته واگذار می‌کنند و دیگر خدمات سیستم عامل توسط تعدادی فرآیند سرویس‌دهنده صورت می‌گیرد. این همان ایده‌ی مدل مشتری – سرویس‌دهنده است که به آن **معماری ریز هسته** نیز می‌گویند.

توجه: روند طراحی سیستم عامل‌های جدید همواره با این این ایده همراه بوده است که تا جایی که ممکن است کدها به لایه‌های بالاتر منتقل شوند تا نهایتا یک هسته کمینه پدید آید. برای این منظور اکثر وظایف سیستم عامل را در سطح کاربر و مشابه فرآیندهای کاربر پیاده‌سازی می‌کنند. برای درخواست یک سرویس، مانند خواندن یک بلوک از فایل، فرآیند کاربر (که اکنون به عنوان فرآیند مشتری شناخته می‌شود) یک درخواست به فرآیند سرویس‌دهنده ارسال می‌نماید و از آن می‌خواهد که کارش را انجام دهد و پاسخ را برگرداند. کار هسته در این مدل برقراری ارتباط بین مشتری‌ها و سرویس‌دهنده از طریق تبادل پیام است. البته هسته، وظایف دیگری نیز دارد. به طور کلی وظایف هسته در این مدل به صورت زیر است:

۱- تبادل پیام مابین مشتری‌ها و سرویس‌دهنده‌ها

۲- زمان‌بندی پردازنده، فرآیندها و ایجاد چندبرنامگی

۳- بخش سطح پایین مدیریت حافظه مانند برنامه‌ریزی ثبات‌های سخت‌افزار مدیریت حافظه

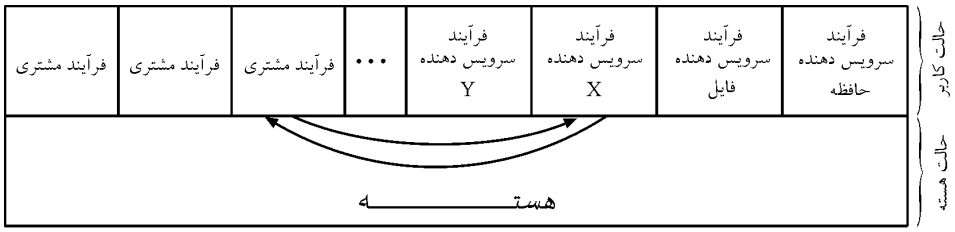
khalilifar.ir

۴- بخش سطح پایین مدیریت I/O که برنامه‌ریزی ثبات‌های ویژه کنترل کننده‌ها و کارهایی را بر عهده دارند که اگر در سطح کاربر انجام شود، آنگاه امنیت کل سیستم به مخاطره خواهد افتاد.

توجه: در این روش، سیستم عامل به چند بخش (خدمات فایل، خدمات فرآیند، خدمات ترمینال و خدمات حافظه) تقسیم شده است، که هریک به عنوان یک سرویس‌دهنده فقط یکی از وظایف سیستم عامل را انجام می‌دهد، بنابراین سیستم عامل را می‌توان ساده‌تر تحلیل، طراحی، پیاده‌سازی و نگهداری کرد.

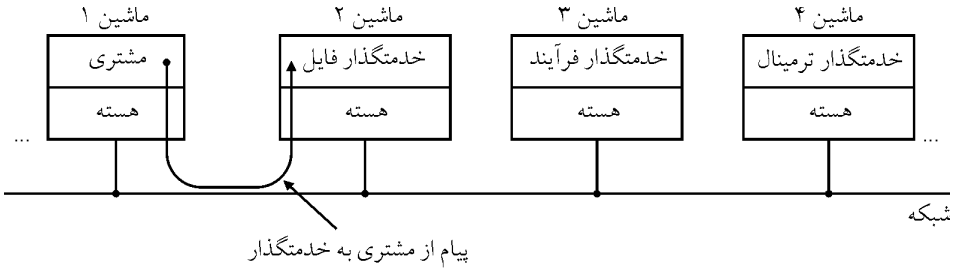
توجه: سرویس‌دهنده‌ها در مُد کاربر اجرا شده و ریز هسته با آنها همانند فرآیندهای کاربر رفتار می‌کند. بنابراین هیچ یک از سرویس‌دهنده‌ها دسترسی مستقیم به سخت‌افزار را ندارد. همچنین اگر کارکرد یکی از سرویس‌دهنده‌ها مختل شود، فقط همان بخش مختل می‌شود و این اختلال به دلیل استقلال عملیاتی هر سرویس‌دهنده به سایر سرویس‌دهنده‌ها و به تبع کل سیستم منتقل نمی‌شود.

توجه: نحوه ی پیاده‌سازی و عملکرد مدل مشتری – سرویس‌دهنده در شکل زیر نشان داده شده است:



ساختار مشتری – سرویس‌دهنده

در حالت دیگر می‌توان از مدل مشتری و سرویس‌دهنده در سیستم‌های توزیع شده استفاده نمود. به شکل زیر توجه کنید.



با توجه به شکل فوق فرض کنید، مشتری بر روی یک ماشین مجزا از سرویس‌دهنده اجرا شده است، در این حالت همانطور که در مدل مشتری و سرویس دهنده ماشین محلی، یک پیام درخواست از مشتری به سرویس‌دهنده از طریق هسته ارسال می‌شد و سرویس‌دهنده پاسخ می‌داد، در سیستم توزیع شده هم این پیام از مشتری به هسته ماشین مشتری و از آن به یک شبکه ارتباطی

khalilifar.ir

و از آن به هسته ماشین سرویس‌دهنده و سپس به سرویس‌دهنده انتقال می‌یابد و پاسخ هم همین مسیر را باز می‌گردد. نکته‌ای که در این روش وجود دارد این است که برای مشتری و یا سرویس‌دهنده بین پیامی که از ماشین محلی آن‌ها می‌آید با پیامی که از ماشین راه دور دیگر می‌آید، تفاوتی وجود ندارد (و در واقع برای آن‌ها قابل تشخیص نیست). به عبارت دیگر از آنجا که یک مشتری به وسیله ارسال پیام‌هایش با یک سرویس‌دهنده ارتباط برقرار می‌کند، مشتری نیاز ندارد که بداند آیا به پیغام وی به صورت محلی در ماشین خودش رسیدگی می‌شود و یا اینکه پیغام از طریق شبکه به یک ماشین دور ارسال می‌شود.

مزیت‌های سیستم‌های مشتری و سرویس‌دهنده:

مهمترین مزیت سیستم‌های مشتری و سرویس‌دهنده طراحی پیمانه‌ای عالی آن‌ها می‌باشد. از این رو که وظایف سیستم عامل به طور مفصل و منسجم از هم جدا شده و هر کدام در یک پیمانه قرار گرفته‌اند، مزیت پیمانه‌ای بودن این روش باعث بروز خصوصیات زیر می‌شود:

۱- انعطاف‌پذیری (Flexibility) بالا

۲- قابلیت نگهداری (Maintainability) ، اشکال‌زدایی (Debug) و ترمیم (Recovery) بالا

۳- قابلیت اطمینان (Reliability) بالا

۴- مقیاس‌پذیری (قابلیت توسعه) بالا

۵- سهولت پیاده‌سازی

۶- مناسب بودن برای سیستم‌های شبکه‌ای و توزیعی

معایب سیستم‌های مشتری و سرویس‌دهنده:

در مقابل همین خاصیت پیمانه‌ای بودن و جدا کردن سرویس‌دهنده‌ها از مشتری‌ها باعث پایین آمدن کارایی (سرعت) این سیستم نسبت به سیستم‌های دیگر شده است. فرض کنید که مشتری اقدام به ارسال یک پیام درخواست می‌کند این پیام در حین ارسال توسط مشتری و دریافت توسط سرویس‌دهنده باعث وقوع 2 تله می‌شود و پاسخ آن نیز به همین صورت، بنابراین برای هر درخواست و پاسخ 4 تله رخ می‌دهد، 2 فراخوان سیستمی برای ارسال و دریافت «**پیام درخواست**» و همچنین 2 فراخوان سیستمی برای ارسال و دریافت «**پیام پاسخ**». همچنین در موازات آن فراخوانی‌های سیستمی برای مدیریت حافظه و I/O نیز رخ می‌دهد که کارایی را تا حد زیادی کاهش می‌دهد.

۱۳- گزینه () صحیح است.

الگوریتم SJF (Shortest Job First)

در این روش ابتدا کاری برای اجرا انتخاب می‌شود که از همه کوتاهتر باشد (زمان اجرای کمتری داشته باشد).

khalilifar.ir

توجه: این الگوریتم، **SPN** (Shortest Process Next) و
SPT (Shortest Process Time) نیز نامیده می‌شود.

توجه: **SJF** یک الگوریتم **انحصاری** (Non Preemptive) است. در سایر متون فارسی به الگوریتم انحصاری، الگوریتم «**غیرقبضه‌ای**» یا «**غیرقابل پس گرفتن**» یا «**غیرقابل تخلیه پیش هنگام**» نیز گفته می‌شود.

توجه: یک نقص عمده الگوریتمSJF این است که ممکن است باعث قحطی‌زدگی فرآیندهای طولانی شود. به این ترتیب که اگر همواره تعدادی فرآیند کوچک وارد سیستم شوند، اجرای فرآیندهای بزرگ به طور متناوب به تعویق می‌افتد. این روال حتی می‌تواند تا بینهایت ادامه یابد و هیچگاه نوبت به فرآیندهای بزرگ نرسد!!!!!

توجه: در این روش اگر دو فرآیند مدت زمان اجرای برابر داشته باشند، بر اساس FCFS زمان‌بندی می‌شوند.

توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسخ و میانگین زمان گردش کار (بازگشت) فرآیندهاست.

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآیندها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجرای فرآیندها را فقط حدس زده و به طور تقریبی بدست آورد.

الگوریتم SRT (Shortest Remaining Time)

این الگوریتم نسخه **غیرانحصاری** (Preemptive) الگوریتم SJF است. در سایر متون فارسی به الگوریتم غیرانحصاری، الگوریتم «**قبضه‌ای**» یا «**قابل پس گرفتن**» یا «**قابل تخلیه پیش هنگام**» نیز گفته می‌شود.

در این الگوریتم اگر حین اجرای یک فرآیند، فرآیندی وارد شود که زمان اجرای کوتاه‌تری داشته باشد، پردازنده را در اختیار می‌گیرد.

توجه: این الگوریتم، **SRPT** (Shortest Remaining Processing Time)،
SRTF (Shortest Remaining Time First) و
SRTN (Shortest Remaining Time Next) نیز نامیده می‌شود.

توجه: اگر لحظه ورود همه فرآیندها یکی باشد، الگوریتمSRT مشابه SJF عمل می‌کند.

توجه: در الگوریتم **SRT** نیز همانند الگوریتم SJF، احتمال وقوع **قحطی‌زدگی برای کارهای بزرگ** وجود دارد.

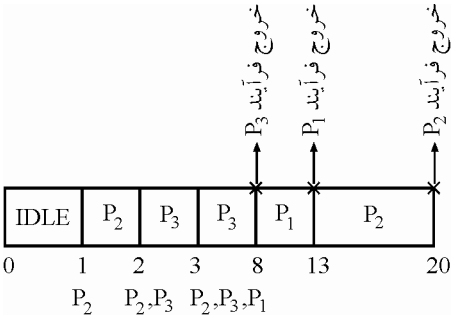
توجه: مطابق فرض سوال الگوریتم «**Preemptive Shortest Remaining Job First**» در نظر گرفته شده است که همان SRT **غیرانحصاری** است.

khalilifar.ir

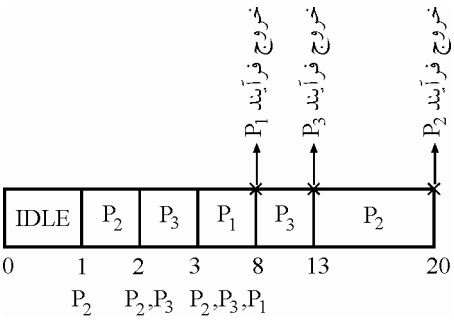
با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	3	5		
P_2	1	8		
P_3	2	6		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



توجه: در لحظه 3 و لحظه ورود فرآیند P_1 ، زمان باقی‌مانده فرآیند P_1 و P_3 هر دو برابر مقدار 5 واحد زمانی است که به دلیل جلوگیری از تعویض متن و به تبع رعایت اصل بهینگی و بالابردن بهره‌وری CPU در لحظه 3 پردازنده به همان فرآیند P_3 اختصاص می‌یابد. که البته اگر اول فرآیند P_1 و سپس فرآیند P_3 هم اجرا شود بازهم پاسخ در گزینه‌ها وجود ندارد.



توجه: در ادامه، برای حل سوال نمودار گانت اول مورد استفاده قرار گرفته است.

زمان ورود فرآیند – زمان خروج اول فرآیند = زمان پاسخ فرآیند

زمان پاسخ $P_1 = 13 - 3 = 10$

زمان پاسخ $P_2 = 2 - 1 = 1$

زمان پاسخ $P_3 = 8 - 2 = 6$

khalilifar.ir

میانگین زمان پاسخ = ART = $\frac{10+1+6}{3} = \frac{17}{3} = 5.66$

زمان ورود فرآیند – زمان خروج کامل فرآیند = زمان بازگشت فرآیند

زمان بازگشت P_1 = $13-3=10$

زمان بازگشت P_2 = $20-1=19$

زمان بازگشت P_3 = $8-2=6$

میانگین زمان بازگشت = ATT = $\frac{10+19+6}{3} = \frac{35}{3} = 11.67$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

زمان انتظار P_1 = $10-5=5$

زمان انتظار P_2 = $19-8=11$

زمان انتظار P_3 = $6-6=0$

میانگین زمان انتظار = AWT = $\frac{5+11+0}{3} = \frac{16}{3} = 5.34$

میانگین زمان اجرا = AST = $\frac{5+8+6}{3} = \frac{19}{3} = 6.33$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

11.67= 6.33 + 5.34

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	3	5	5	10
P_2	1	8	11	19
P_3	2	6	0	6

میانگین زمان بازگشت = 11.67 میانگین زمان انتظار + 5.34 میانگین زمان اجرا 6.33

khalilifar.ir

توجه: همانطور که پُر واضح است، پاسخ سوال در گزینه‌ها موجود نیست.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه دوم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

۱۴- گزینه (۱) صحیح است.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
Add  this process to s.queue;
block ();
  }
}
```

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if (s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر اینصورت، فرآیند، وارد خط بعدی می‌شود.

تابع signal(s): عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Signal (semaphore s)
{
s.count = s.count + 1
   if (s.count< = 0)
   {
    Remove a process from queue;
    Wake up ();
     }
}
```

khalilifar.ir

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به خط بعدی است که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up() بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

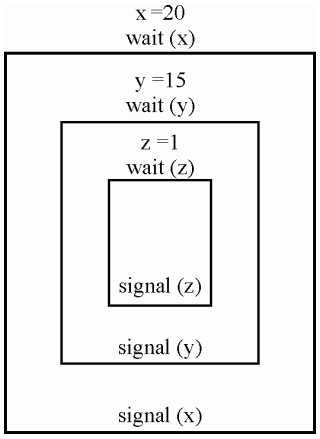
به بیان دیگر هر فرآیند با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

توجه: در صورت سوال مطرح شده است که سه سمافور با مقدار اولیه x=20، y=15 و z=1 در نظر گرفته شود. همچنین گفته شده است که قطعه کد زیر توسط 50 پردازه (process) اجرا می‌شود.

```
wait (x);
 wait (y);
   wait (z);
      a= a+1;
   signal (z);
 signal (y);
signal (x);
```

در ادامه خواسته سوال این است که، حداکثر طول صفی که برای سمافور z تشکیل می‌شود، چقدر است؟

شکل زیر را در نظر بگیرید:



khalilifar.ir

دقت کنید که در صورت سوال مطرح شده است که قطعه کد فوق توسط 50 پردازه (process) اجرا می‌شود. پس می‌بایست قطعه کد فوق توسط 50 فرآیند اجرا گردد و از آن عبور کنند.
50 فرآیند P_1، P_2، P_3، ...، P_{47}، P_{48}، P_{49}، P_{50} را در نظر بگیرید. از این تعداد با توجه به شمارنده سمافور x که برابر مقدار 20 است، ابتدا فرآیندهای P_1، P_2، P_3، ...، P_{19}، P_{20} به ترتیب اجرا می‌شوند و از سمافور x عبور می‌کنند. و فرآیندهای P_{21}، P_{22}، ... ، P_{47}، P_{48}، P_{49}، P_{50} پشت سمافور x در صف سمافور x آرام می‌گیرند و می‌خوابند. در ادامه از بین فرآیندهای عبور کرده از سمافور x یعنی فرآیندهای P_1، P_2، P_3، ...، P_{19}، P_{20} با توجه به شمارنده سمافور y که برابر مقدار 15 است، فقط فرآیندهای P_1، P_2، P_3، ...، P_{14}، P_{15} این شانس را پیدا می‌کنند که از سمافورy عبور کنند و عبور می‌کنند، و فرآیندهای P_{16}، P_{17}، P_{18}، P_{19}، P_{20} پشت سمافور y در صف سمافور y آرام می‌گیرند و می‌خوابند. پس درحال حاضر 5 فرآیند در صف سمافور y به خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابر مقدار5 است. در ادامه از بین فرآیندهای عبور کرده از سمافور y یعنی فرآیندهای P_1، P_2، ...، P_{14}، P_{15} با توجه به شمارنده سمافور z که برابر مقدار 1 است، فقط فرآیند P_1 این شانس را پیدا می‌کند که از سمافورz عبور کند و عبور می‌کند، و فرآیندهای P_2، P_3، ...، P_{14}، P_{15} پشت سمافور z در صف سمافور z آرام می‌گیرند و می‌خوابند. پس درحال حاضر 14 فرآیند در صف سمافور z به خواب رفته‌اند، بنابراین طول صفی که برای سمافور z تشکیل شده است، هم اکنون برابر مقدار14 است.

در ادامه فرآیند P_1 پس از عبور از ناحیه سمافور z تابع signal(z) را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_2 می‌گردد، در این لحظه فرآیند P_2 در خط بعد از تابع wait(z) و قبل از تابع signal(z) قرار می‌گیرد، همچنین در ادامه فرآیند P_1 پس از عبور از ناحیه سمافور y تابع signal(y) را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{16} می‌گردد، در این لحظه فرآیند P_{16} در خط بعد از تابع wait(y) و قبل از تابع wait(z) قرار می‌گیرد، همچنین در ادامه فرآیند P_1 پس از عبور از ناحیه سمافور x تابع signal(x) را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{21} می‌گردد، در این لحظه فرآیند P_{21} در خط بعد از تابع wait(x) و قبل از تابع wait(y) قرار می‌گیرد، این روند تا اتمام ملاقات هر سه ناحیه x، y و z برای همه فرآیندها از P_1 تا P_{50} ادامه پیدا می‌کند.

۱۵- گزینه () صحیح است.

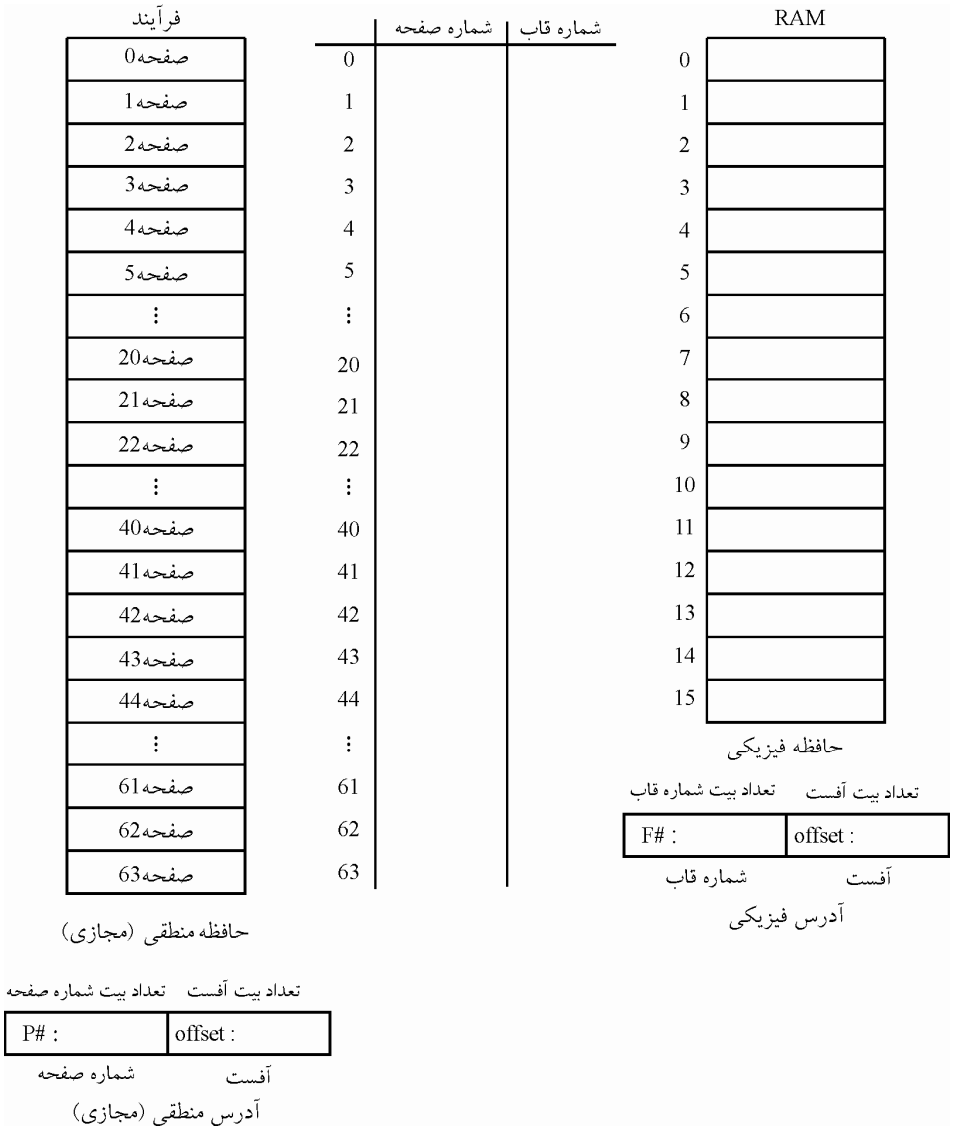
در صورت سوال مطرح شده است که سامانه داری 64 صفحه مجازی (Virtual Pages) است که به 16 قاب فیزیکی (Physical Frames) بر اساس رابطه زیر نگاشت داده می‌شود. طول هر صفحه یک کیلو کلمه (1 K Words) است. اگر آدرس مجازی برابر 1010101000111101 باشد، کدام گزینه آدرس فیزیکی را نشان می‌دهد؟

توجه: همانطور که پُر واضح است، به دلیل بیان نشدن رابطه نگاشت، سوال غیرقابل حل است.

khalilifar.ir

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه سوم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بندی به صورت زیر است:



$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}} = 64$$

khalilifar.ir

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = 16$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^{64} = 6\text{bit}$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{16} = 4\text{bit}$$

توجه: در صورت سوال ذکر نشده است که یک کلمه یک بایت در نظر گرفته شود یا چهار بایت، اما بر اساس طول 16 بیتی آدرس مجازی و طول 6 بیتی تعداد بیت شماره قاب، یک کلمه معادل یک بایت در نظر گرفته می‌شود تا طول تعداد بیت آفست همان 10 بیت باشد.

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{1\text{KB}} = \log_2^{2^{10}} = \log_2^{2^{10}} = 10\text{bit}$$

همچنین، اندازه آدرس‌های منطقی (مجازی) و فیزیکی به صورت زیر است:

طول آدرس منطقی = تعداد بیت شماره صفحه + تعداد بیت آفست = $6\text{bit} + 10\text{bit} = 16\text{bit}$

طول آدرس فیزیکی = تعداد بیت شماره قاب + تعداد بیت آفست = $4\text{bit} + 10\text{bit} = 14\text{bit}$

همچنین داریم:

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{\text{تعداد بیت آدرس منطقی}} = 2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{16} = 2^6 \times 2^{10} = 2^{16} = 64\,\text{KB}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{\text{تعداد بیت آدرس فیزیکی}} = 2^{\text{تعداد بیت شماره قاب}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{14} = 2^4 \times 2^{10} = 2^{14} = 16\,\text{KB}$$

توجه: اندازه آدرس منطقی (مجازی) و فیزیکی (حقیقی) الزاماً برابر نیست.

اندازه جدول صفحه = تعداد درایه‌های جدول صفحه (تعداد صفحات فرآیند) × عرض جدول صفحه

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد، به صورت زیر:

عرض جدول صفحه = تعداد بیت‌های شماره قاب + تعداد بیت‌های کنترلی

در سوال تعداد بیت‌های کنترلی بیان نشده است پس 0 بیت مربوط به بیت‌های کنترلی و 4 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس: عرض جدول صفحه فوق برابر $4\text{bit} + 0\text{bit} = 4\,\text{bit}$ می‌باشد.

همانطور که گفتیم، اندازه جدول صفحه، از رابطه زیر محاسبه می‌گردد:

khalilifar.ir

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه
که مطابق رابطه فوق داریم:

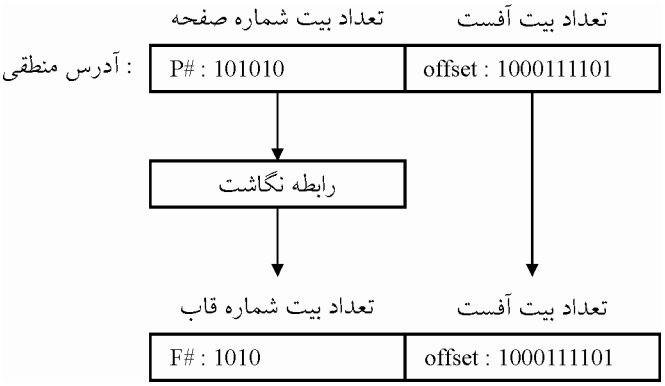
$$\text{اندازه جدول صفحه} = 64 \times 4\,\text{bit} = 256\,\text{bit} = 32\,\text{Byte}$$

توجه: در صورت سوال مطرح شده، برای تبدیل شماره صفحه به شماره قاب یک الگوی ثابت و از قبل تعیین شده بر اساس یک رابطه نگاشت، مشخص گردیده است که خود این فرض نادرست می‌باشد. تعیین شماره قاب در حافظه فیزیکی برای یک صفحه از یک فرآیند براساس قاب‌های آزاد در حافظه فیزیکی و توسط سیستم عامل انجام می‌گردد و نه براساس یک الگوی از قبل مشخص شده و نگاشت ثابت، زیرا یک صفحه از فرآیند همواره به یک قاب خاص از حافظه فیزیکی نگاشت نمی‌شود.

از آنجا که فرآیند موجود در این سیستم شامل 64 صفحه است، بنابراین 6 بیت سمت چپ $(b = \log_2^{64} = 6\,\text{bit})$ از آدرس منطقی 16 بیتی مربوط به شماره صفحه و 10 بیت سمت راست باقیمانده مربوط به آفست است. همچنین از آنجا که حافظه فیزیکی موجود در این سیستم نیز شامل 16 قاب است، بنابراین 4 بیت سمت چپ $(b = \log_2^{16} = 4\,\text{bit})$ از آدرس فیزیکی نیز مربوط به تعداد بیت‌های شماره قاب می‌باشد. با توجه به اینکه در ترجمه آدرس منطقی به فیزیکی در سیستم‌های صفحه‌بندی شده، تعداد بیت‌های آفست ثابت مانده و فقط شماره صفحه با شماره قاب جای‌گزین می‌گردد، بنابراین آدرس فیزیکی نیز دارای طولی برابر 14 بیت خواهد بود.

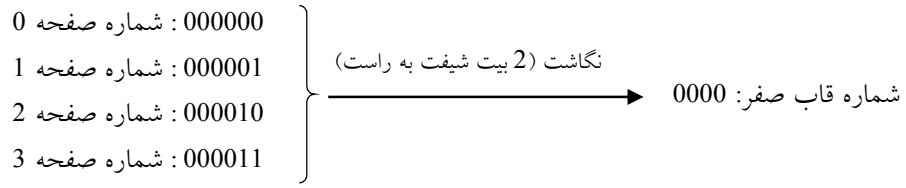
توجه: باتوجه به فرض مطرح شده در صورت سؤال، باید رابطه نگاشت مطرح می‌شد که نشده‌است، بنابراین همانطور که گفتیم به دلیل بیان نشدن رابطه نگاشت، سوال غیرقابل حل است.

توجه: اما با توجه به کلید اولیه سازمان سنجش یعنی گزینه سوم می‌توان حدس زد که رابطه نگاشت دوبار شیفت به راست بوده است که در صورت سوال فراموش شده است مطرح شود، پس برای این کار، کافی است مقدار دودویی شماره صفحه را دوبار به سمت راست شیفت دهیم تا مقدار شماره قاب بدست بیاید و سپس 10 بیت سمت راست آدرس منطقی را به عنوان آفست به آن بیافزاییم:



khalilifar.ir

که مقدار حاصل مطابق کلید اولیه سازمان سنجش آموزش کشور یعنی گزینه سوم است. هرچند که گفتیم این نوع نگاشت ثابت نادرست است. چرا که همیشه مجموع صفحاتی که در 2 بیت سمت راست شماره صفحه و به شکل 00، 01، 10، 11 متفاوت ولی در 4 بیت باقی‌مانده یکسان هستند به دلیل انتقال 2 بیت به سمت راست، همگی یک شماره قاب واحد خواهند داشت. یعنی به هر قاب موجود در حافظه فیزیکی، 4 صفحه از آدرس منطقی نسبت داده می‌شود. مثلاً صفحات شماره 0، 1، 2 و3 همگی به شماره قاب 0 نگاشت می‌شوند و نمی‌توانند همزمان با هم درون حافظه فیزیکی قرار داده شوند.



۱۶- گزینه (۴) صحیح است.

وظیفه زمانبند فرآیند، انتخاب یک فرآیند جهت در اختیار گرفتن پردازنده است (در واقع مشخص می‌کند که در لحظه حاضر، پردازنده در اختیار کدام فرآیند می‌باشد).

الگوریتم‌های زمانبندی اهداف زیر را دنبال می‌کند (در واقع موارد زیر معیارهایی جهت مقایسه عملکرد الگوریتم‌های مختلف می‌باشند):

عدالت (Fairness)

منظور از عدالت این است که فرآیندهای هم‌ارز از دید زمانبند، یکسان تلقی شوند. به عبارت دیگر یک فرآیند نباید از فرآیندهای هم‌سطح خود سهم بیشتری ازCPU دریافت کند.

البته این امر درباره فرآیندهایی که اولویت یکسان ندارند صادق نیست و منطقی است که فرآیندی با اولویت بالاتر، سهم بیشتری از CPU دریافت کند.

زمان پاسخ (Response)

یک الگوریتم زمانبندی باید زمان پاسخ فرآیندها را به حداقل برساند. منظور از زمان پاسخ، مدت زمان بین لحظه ورود کار و لحظه دریافت نتیجه و خروجی اول می‌باشد. این معیار در سیستم‌های محاوره‌ای و تعاملی اهمیت ویژه‌ای دارد.

زمان گردش (برگشت یا بازگشت) (Turnaround Time)

یک الگوریتم زمانبندی باید زمان گردش کار فرآیندها را به حداقل برساند. منظور از زمان گردش کار، مدت زمان بین لحظه ورود کار و لحظه خروج کامل از سیستم می‌باشد.

توجه: به تفاوت بین زمان پاسخ و زمان بازگشت دقت کنید:

khalilifar.ir

زمان پاسخ، مدت زمان بین صدور فرمان و تولید اولین نتیجه و خروجی آن می‌باشد اما زمان بازگشت، مدت زمان بین ورود کار و تکمیل نهایی و نتیجه و خروج کامل آن است.

زمان انتظار (Waiting Time)

یک الگوریتم زمانبندی باید زمان انتظار فرآیندها را به حداقل برساند. منظور از زمان انتظار، مجموع زمان‌هایی است که یک فرآیند در صف فرآیندهای آماده و انتظار، منتظر دریافت CPU می‌باشد.

واضح است که یک الگوریتم زمانبندی، بر روی مدت زمان اجرای یک فرآیند و مدت زمان ورودی و خروجی تأثیری ندارد، بلکه فقط بر روی مدت زمان انتظار یک فرآیند در صف آماده مؤثر است.

بهره‌وری پردازنده (CPU Utilization or CPU Efficiency)

یک الگوریتم زمانبندی باید بهره‌وری پردازنده را به حداکثر برساند، به این معنی که حتی الامکان در تمام زمان‌ها پردازنده مشغول باشد تا زمان‌های هدر رفته پردازنده به حداقل برسد. که به آن درصد استفاده مفید از CPU نیز گفته می‌شود. زمان‌های بیکاری و زمان‌های تعویض متن، زمان‌های غیرمفید محسوب می‌شوند. برای محاسبه آن، زمان مفید را بر زمان کل یعی حاصل جمع زمان مفید و زمان غیرمفید، تقسیم می‌کنیم و در 100 ضرب می‌کنیم.

توان گذردهی (توان عملیاتی) (Throughput)

یک الگوریتم زمانبندی باید توان گذردهی را به حداکثر برساند. منظور از گذردهی، تعداد فرآیندهایی است که در واحد زمان تکمیل می‌شوند. به عبارت دیگر توان گذردهی تعداد کارهای انجام شده، تکمیل شده و خارج شده در واحد زمان است. که برای محاسبه آن تعداد کل کارها بر زمان اجرای کل کارها تقسیم می‌شود.

مهلت (Deadline)

مدت زمانی که از لحظه ورود یک فرآیند بی‌درنگ فرصت وجود دارد تا پردازش آن تمام شود. ملاک حفظ و گارانتی کردن مهلت زمانی است.

مطلوبیت (Proportionality)

مطلوبیت به معنی برآورده‌سازی نیازهای غیروظیفه‌مندی کاربران نهایی است، یعنی اگر به کمک سیستم عامل نیازهای فرآیندها مثل پردازنده و حافظه برآورده می‌شود و به تبع آن نیازهای کاربران نهایی برآورده می‌شود، کاربران نهایی دوست دارند که کار فرآیندها به کمک سیستم عامل در زمان کوتاه هم انجام شوند و این نیاز غیروظیفه مندی کاربران نهایی است. کاربران نهایی از سیستم عامل توقع دارند هم وظیفه فرآیندها به درستی انجام شود و هم غیروظیفه‌مندی (سرعت انجام)

khalilifar.ir

فرآیندها مطلوب باشد.

توجه: علاوه بر معیارهای ذکر شده، معیارهای زیر نیز کم و بیش برای مقایسه الگوریتم‌های زمانبندی به‌کار می‌روند:

- **تعادل یا توازن (Balance) در استفاده از منابع:** در واقع الگوریتم‌های زمانبندی باید باعث شود از تمام منابع سیستم به خوبی استفاده و بهره‌برداری شود. به عبارت دیگر منابع را مشغول نگه دارد. به بیان دیگر مشغول نگه داشتن همه بخش‌های سیستم و چندبرنامگی کارهای محدود به CPU و محدود به I/O در کنار یکدیگر و ایجاد توازن بار در استفاده از همه منابع.
- **پیش بینی پذیری (Predicatability):** الگوریتم زمانبندی باید به گونه‌ای عمل کند که در اجراهای مختلف و چندباره یک کار، به‌خصوص زمان انتظار و زمان پاسخ تغییرات شدیدی نداشته باشند. این معیار برای کاربران اهمیت دارد. مانند پرهیز از تنزل کیفیت در سیستم‌های چند رسانه‌ای.
- رعایت اصول و اولویت‌ها
- رعایت ضرب العجل‌ها

نکته: برخی از معیارها و اهداف فوق با هم محقق نمی‌شوند و ممکن است با یکدیگر در تضاد باشند. به عنوان مثال ممکن است جهت برقراری عدالت، سیاستی اعمال شود که بهره‌وری CPU را کاهش دهد.

توجه: الگوریتم‌های زمانبندی به دو دسته تقسیم می‌شوند:

۱- Non Preemptive (غیر قابل پس‌گیری، انحصاری)

۲- Preemptive (قابل پس‌گیری، غیر انحصاری)

در زمانبندی انحصاری، هنگامی که پردازنده در اختیار فرآیندی قرار گیرد نمی‌توان پردازنده را از وی پس گرفت، مگر این‌که خود داوطلبانه آن را آزاد کند و یا فرآیند خاتمه یابد اما در زمانبندی غیر انحصاری می‌توان پردازنده را از فرآیندی پس گرفت و به دیگری تحویل داد.

توجه: در سیستم‌های تعاملی، الگوریتم‌های غیر انحصاری (قابل پس‌گیری) تنها گزینه قابل قبول هستند، زیرا در غیر این‌صورت یک فرآیند CPU را به انحصار خود درآورده و به دیگران اجازه کار نمی‌دهد. شاید به‌نظر برسد برای سیستم‌های بی‌درنگ تنها باید از الگوریتم‌های غیر انحصاری (قابل پس‌گیری) استفاده کرد، اما در این سیستم‌ها از هر دو نوع الگوریتم می‌توان بهره برد. از آن جا که فرآیندها در سیستم‌های بی‌درنگ اغلب بسیار کوچک هستند و به سرعت اجرا شده و خاتمه می‌یابند (یا مسدود می‌شوند)، الگوریتم‌های انحصاری (غیر قابل پس‌گیری) نیز در این سیستم‌ها کاربرد دارند.

توجه: در متون فارسی به الگوریتم **انحصاری** (Non Preemptive)، الگوریتم «**غیرقبضه‌ای**» یا «**غیرقابل پس گرفتن**» یا «**غیرقابل تخلیه پیش هنگام**» نیز گفته می‌شود.

khalilifar.ir

توجه: در متون فارسی به الگوریتم **غیرانحصاری** (Preemptive)، الگوریتم «**قبضه‌ای**» یا «**قابل پس گرفتن**» یا «**قابل تخلیه پیش هنگام**» نیز گفته می‌شود.

توجه: زمان Burst (Burst time) مربوط به زمان اجرای یک فرآیند است و به معیارهای الگوریتم‌های زمان‌بندی فرآیند مرتبط نیست، بنابراین پُرواضح است که گزینه چهارم پاسخ سوال است.

۱۷- گزینه (۲) صحیح است.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه و نهایی خود، گزینه دوم را به عنوان پاسخ اعلام کرده بود. اما بهتر بود در کلید نهایی گزینه اول به عنوان پاسخ نهایی اعلام می‌شد.

توجه: اصولا کار لایه‌های نرم افزار ورودی و خروجی سطح کاربر(درخواست ورودی و خروجی)، نرم افزار ورودی و خروجی مستقل از دستگاه (جستجوی ورودی و خروجی)، گرداننده دستگاه (Device Driver) (جستجوی ورودی و خروجی) برای جستجوی ورودی و خروجی است و کار لایه روتین سرویس دهنده وقفه (اداره‌کننده وقفه) برای تحویل ورودی و خروجی مورد نظر به فرآیند مرتبط است. بنابراین برای خواندن از دیسک، محاسبات مربوط به شیار(Track)، قطاع (Sector) و هد دیسک در لایه‌های قبل از لایه روتین سرویس دهنده وقفه (اداره‌کننده وقفه) یعنی لایه گرداننده دستگاه (Device Driver) و گزینه اول انجام می‌شود. مگر اینکه نظر طراح این باشد که لایه روتین سرویس دهنده وقفه (اداره‌کننده وقفه) دو بخشی است و بخشی از آن که مربوط به محاسبات است قبل از وقفه تحویل ورودی و خروجی و در لایه گرداننده دستگاه (Device Driver) انجام می‌شود که البته فرض نادرستی است.

توجه: یکی از مهم‌ترین وظایف سیستم عامل کنترل و مدیریت دستگاه‌های I/O در کامپیوتر است. سیستم عامل باید فرامینی را برای دستگاه‌ها صادر نماید، وقفه‌های ایجاد شده به وسیله دستگاه‌ها را دریافت نماید، خطاها را پردازش نماید و برای سهولت استفاده از سیستم رابط‌هایی بین دستگاه‌ها و بقیه سیستم ایجاد کند.

توجه: هر دستگاه I/O از دو بخش مولفه **الکترونیکی** و مولفه **مکانیکی** تشکیل شده است. مولفه الکترونیکی همان **کنترل‌کننده دستگاه** (Device Controller) و مولفه مکانیکی همان **دستگاه** است. و علت ذکر تفاوت بین کنترل‌کننده و دستگاه این است که سیستم عامل با کنترل‌کننده کار می‌کند و نه خود دستگاه. هر کنترل‌کننده دارای تعداد محدودی رجیستر (جهت دریافت فرمان و یه به منظور تعیین وضعیت دستگاه) است که برای ارتباط برقرار کردن با پردازنده به کار می‌روند. سیستم عامل با نوشتن در این رجیسترها قادر است، دستوراتی مبتنی بر تحویل و یا دریافت داده، خاموش و روشن کردن دستگاه را صادر نماید و همچنین با خواندن از آن می‌تواند، از وضعیت دستگاه باخبر گردد. علاوه بر رجیسترهای کنترلی، بسیاری از دستگاه‌ها دارای بافر داده‌ای نیز هستند که سیستم عامل می‌تواند از آنها بخواند یا در آنها بنویسد. به عنوان مثال می‌توان به بافر داده

khalilifar.ir

Video RAM اشاره نمود که سیستم عامل و یا برنامه‌های کاربردی جهت نمایش پیکسل روی صفحه مانیتور، در آن می‌نویسد. برای مثالی دیگر هد دیسک تحت کنترل نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک قرار دارد، و به خودی خود هد دیسک فهم و شعور ندارد، یعنی بدون نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک، هد دیسک نمی‌داند کجا برود و کجا نرود، چه کند و چه نکند. بلکه نرم‌افزار زمان‌بندی دیسک به واسطه استفاده از یک الگوریتم خاص، توسط کنترل‌کننده هد دیسک، هد دیسک را کنترل و جابه‌جا می‌کند، در واقع این نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک است که به هد دیسک فهم و شعور می‌دهد و به هد می‌گوید که کجا برود و کجا نرود، چه کند و چه نکند. بنابراین در زمان حرکت مکانیکی هد به سمت یک سیلندر، هد شخصاً تصمیم جدیدی نمی‌گیرد، بلکه تابع تصمیمات نرم‌افزار زمان‌بندی و کنترل‌کننده خود است

توجه: سیستم عامل با نوشتن فرامین در ثبات‌های خاص کنترل‌کننده، عملیات ورودی و خروجی مورد نظر را انجام می‌دهد. هنگامی که فرمانی پذیرفته شد، پردازنده، کنترل‌کننده دستگاه را رها می‌کند و به سایر کارها می‌پردازد. بعد از کامل شدن فرمان، کنترل‌کننده، وقفه‌ای ایجاد می‌کند که به سیستم عامل امکان می‌دهد، کنترل پردازنده را به دست آورد و نتایج عمل را بررسی نماید، پردازنده نتایج دلخواه و وضعیت دستگاه را، با خواندن چند بایت از ثبات‌های کنترل‌کننده به دست می‌آورد.

توجه: نرم‌افزار I/O ، به صورت مجموعه‌ای از لایه‌هاست که رابط مناسبی را برای کاربر فراهم نماید. نرم‌افزار I/O باید به گونه‌ای طراحی شود که «**استقلال از دستگاه**» را به همراه داشته باشد. این امکان باید وجود داشته باشد که برنامه‌ها به گونه‌ای نوشته شوند که بتوانند با انواع دستگاه‌ها کار کنند، بدون آنکه با تعویض نوع دستگاه نیازی به اصلاح آنها باشد.

توجه: نرم‌افزار I/O کنترل و پردازش خطا یعنی کشف و تصحیح خطا را نیز انجام می‌دهد. البته خطاها باید تا حد امکان نزدیک به سخت افزار پردازش شود. اگر کنترلر یک خطای خواندن را کشف کند، باید در صورتی که می‌تواند، خود به تصحیح آن اقدام کند، در غیر این صورت نرم‌افزار گرداننده آن خطا را پردازش می‌نماید و احتمالا این کار را با خواندن مجدد آن بلوک انجام می‌دهد. بسیاری از خطاها گذرا هستند و در صورت تکرار عمل برطرف خواهند شد. تنها در صورتی که لایه‌های زیرین نرم‌افزاری قادر به رفع مشکل نباشند، لایه‌های بالایی از وجود مشکل باخبر خواهند شد و خطا را تصحیح خواهند کرد.

توجه: نرم‌افزار I/O یک ساختار چهار لایه‌ای دارد به طوری که هر لایه وظیفه خاصی را بر عهده دارد، به صورت زیر:

khalilifar.ir

فرآیندهای کاربران
نرم افزار ورودی و خروجی سطح کاربر (درخواست ورودی و خروجی)
نرم افزار ورودی و خروجی مستقل از دستگاه (جستجوی ورودی و خروجی)
گرداننده دستگاه (Device Driver) (جستجوی ورودی و خروجی)
روتین سرویس دهنده وقفه (اداره‌کننده وقفه) (تحویل ورودی و خروجی)
سخت افزار

توجه: هنگامی که یک برنامه کاربردی سعی می‌کند که یک فایل را بخواند، به ترتیب، عملیات ذیل طبق لایه‌های سیستم I/O در شکل فوق انجام می‌شود:

۱- فراخوان سیستمی (System Call) در سطح کاربر به واسطه دستور ورودی و خروجی انجام می‌شود.

۲- بخشی از سیستم عامل جهت اجرای فراخوان سیستمی صدا زده می‌شود. (اجرای ISR ورود به I/O)

۳- نرم‌افزار مستقل از دستگاه، بلوک مورد درخواست را در حافظه پنهان جستجو می‌کند. در صورتی که بلوک مورد نظر در آن موجود نباشد، مرحله بعدی اجرا می‌شود. در غیر اینصورت رکورد یافت شده به برنامه کاربر تحویل داده می‌شود.

۴- گرداننده دستگاه فراخوانی می‌شود تا تقاضای کاربر به سخت‌افزار و دیسک صادر گردد.

۵- گرداننده دستگاه و برنامه کاربر تا اجرای کامل I/O مسدود می‌گردد.

۶- زمانی که عملیات I/O توسط کنترل‌کننده دستگاه به پایان می‌رسد، سخت‌افزار مربوطه وقفه‌ای را تولید می‌کند.

۷- روتین سرویس‌دهنده وقفه (اداره کننده وقفه یا اجرای ISR خروج از I/O) اجرا می‌شود تا دریابد چه اتفاقی رخ داده است؟

۸- وضعیت دستگاه وقفه دهنده را بررسی کرده، گرداننده دستگاه، نرم افزار مستقل از دستگاه و فرآیند مسدود شده را بیدار می‌کند. سپس رکورد خوانده شده، به برنامه کاربردی تحویل داده می‌شود. واضح است که اجرای روتین سرویس دهی به وقفه بعد از اتمام عملیات ورودی خروجی است. بدین ترتیب فرآیند ورودی و خروجی به پایان می‌رسد و برنامه کاربردی کار خود را ادامه می‌دهد.

توجه: در ادامه جزئیات بیشتری در مورد لایه‌های نرم افزار I/O **با رویکرد بالا به پایین** بیان

khalilifar.ir

می‌کنیم:

نرم افزار ورودی و خروجی سطح کاربر

در این لایه جهت «**درخواست ورودی و خروجی**» مورد نظر فرآیند، توسط فرآیند جاری، لایه نرم افزار ورودی و خروجی سطح کاربر فراخوانی می‌شود.

یک برنامه کامپیوتری (فرآیند کاربر) که در سطح کاربر اجرا می‌شود، علاوه بر دستورات پردازشی و محاسباتی، شامل دستورات ورودی و خروجی نیز می‌باشد، مانند دستورات خواندن (Read) و نوشتن (Write) در پاسکال و یا دستورات خواندن (scanf) و نوشتن (printf) در زبان C و یا دستورات خواندن (cin) و نوشتن (cout) در زبان C++ که بدنه این توابع داخل کتابخانه زبان برنامه‌نوسی تعریف شده است. که هنگام کامپایل، بدنه تعریف این توابع ورودی و خروجی به فرآیند کاربر متصل می‌گردد و در نهایت بدنه دستورات پردازشی و بدنه تعریف توابع ورودی و خروجی در یک برنامه کاربردی یکپارچه می‌گردد تا آماده اجرا گردد. هنگام اجرای بدنه توابع ورودی و خروجی برای مثال دستور خواندن (Read) جهت انجام کار از دل این تابع عمل فراخوان سیستمی (System Call) صورت می‌گیرد. هنگامی‌که رویه فراخوان سیستمی در سطح کاربر فراخوانی می‌شود، به واسطه این فراخوانی و وقفه ورود به I/O بخشی از سیستم عامل جهت اجرای فراخوان سیستمی صدا زده می‌شود. (اجرای ISR ورود به I/O)

فراخوان سیستمی، یک روش برنامه‌نویسی است که در آن یک برنامه کاربردی از هسته سیستم‌عاملی که روی آن اجرا می‌گردد، یک خدمت (سرویس) درخواست می‌کند. این موضوع می‌تواند شامل سرویس‌های مرتبط با سخت‌افزار (برای مثال، دسترسی به دیسک سخت) باشد. فراخوان‌های سیستمی یک واسط اساسی بین یک فرآیند کاربر و سیستم‌عامل فراهم می‌کند.

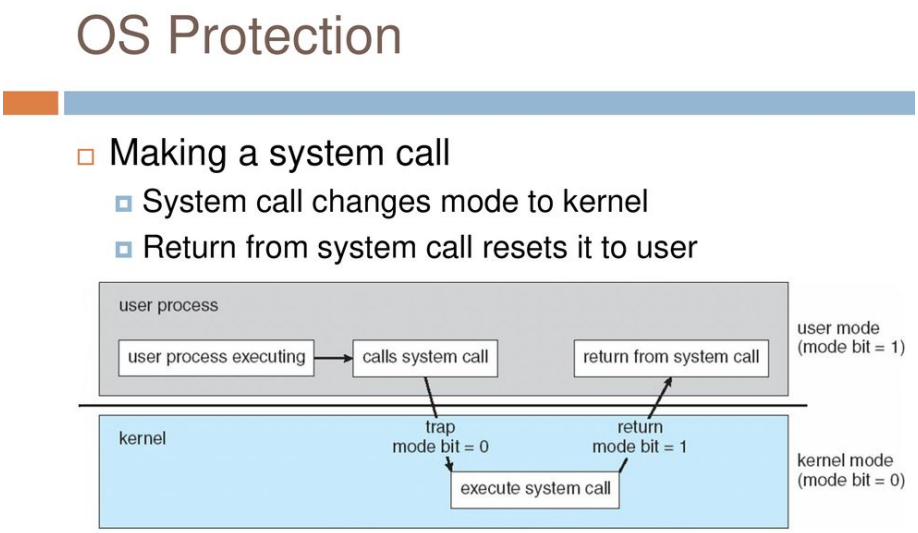
هرگاه یک برنامه سطح کاربر نیاز به دسترسی به منابع سیستم و سخت افزار را داشته باشد، یکی از توابع درون سیستم عامل را فراخوانی می‌کند. که به این عمل فراخوان سیستمی (System Call یا Syscall) می‌گویند فراخوان سیستمی را گاه فراخوان هسته‌ای (kernel call) نیز می‌نامند چراکه برای انجام فراخوان سیستمی پردازنده باید در حالت مد هسته (kernel mode) باشد. به زبانی دیگر هیچ برنامه‌ای حق دسترسی مستقیم به سخت‌افزار را ندارد و باید توسط واسطه‌ای که سیستم عامل در اختیارش قرار می‌دهد و نامش فراخوان سیستمی است به سخت افزار سیستم دسترسی پیدا کند. در حقیقت فراخوان سیستمی پس از مدیریت منابع (Resource Management)، دومین هدف اصلی یک سیستم عامل می‌باشد.

سیستم عامل استفاده از کامپیوتر را ساده می‌سازد. این بدان معناست که برای مثال کاربر یا برنامه نویس بدون درگیر شدن با مسائل سخت افزاری دیسک ها، به راحتی فایلی را بر روی دیسک ذخیره و حذف کند. این کار در واقع با به کار بردن دستورهای ساده‌ای که فراخوان‌های سیستمی را صدا می‌زنند انجام می‌پذیرد. در صورت عدم وجود سیستم عامل، کاربر یا برنامه نویس

khalilifar.ir

می‌بایست آشنایی کاملی با سخت افزارهای مختلف کامپیوتر داشته باشد و روتین هایی برای خواندن یا نوشتن آن‌ها به زبان‌های سطح پائین بنویسد. از این جنبه، گاه از سیستم عامل با عنوان ماشین توسعه یافته (Extended machine) یا ماشین مجازی (Virtual machine) یاد می‌شود که واقعیت سخت افزار را از دید کاربران مخفی می‌سازد.

توجه: اگرچه بخش عظیمی از لایه‌های نرم افزار ورودی و خروجی در داخل سیستم عامل است، اما همانطور که گفتیم بخش اندکی از آن شامل کد برنامه و کتابخانه‌های ورودی و خروجی که توسط کامپایلر به برنامه‌های کاربران پیوند می‌خورد در ناحیه خارج از هسته اجرا می‌گردد. و همانطور که گفتیم فراخوان‌های سیستمی ورودی و خروجی از بدنه تعریف توابع کتابخانه‌ای فراخوانی می‌شوند. که مجموعه تمام بدنه تعریف توابع کتابخانه‌ای متناظر با عملیات ورودی و خروجی در لایه نرم افزار ورودی و خروجی سطح کاربر قرار دارد. شکل زیر گویای مطلب است:



بخش مهم دیگری که در لایه نرم افزار ورودی و خروجی سطح کاربر موجود می‌باشد «سیستم Spooling» را تشکیل می‌دهد. Spooling روشی است که به وسیله آن دستگاه‌های ورودی و خروجی انحصاری در یک سیستم چندبرنامگی کاربرد پیدا می‌کند. به عنوان مثال یک دستگاه چاپگر را در نظر بگیرید. در صورتی که فرآیندی یک فایل متنی را برای چاپ باز کند و سپس ساعت‌ها کاری انجام ندهد، هیچ فرآیند دیگری قادر نخواهد بود چیزی را چاپ نماید. در عوض می‌توان از یک فرآیند خاصی به نام سرویس‌دهنده (Deamon) چاپگر و یک فهرست خاص به نام فهرست Spooling استفاده نمود. برای چاپ کردن یک فایل، فرآیند ابتدا تمام فایلی را که باید چاپ شود، ایجاد می‌نماید و آنرا در فهرست Sppling قرار می‌دهد. این برعهده برنامه Deamon است که فایل‌های این فهرست را چاپ نماید و این فرآیند تنها فرآیندی است که اجازه استفاده از

khalilifar.ir

فایل خاص چاپگر را دارد.
مفهوم Spooling تنها در چاپگر کاربرد ندارد، بلکه در شرایط دیگری نیز به کار گرفته می‌شود. به عنوان مثال جهت انتقال فایل بر روی شبکه، اغلب از یک برنامه سرویس‌دهنده (Deamon) شبکه‌ای استفاده می‌شود. کاربر برای ارسال یک فایل به مکانی، آن فایل را در فهرست Spooling شبکه قرار می‌دهد و مدتی بعد برنامه Deamon شبکه آن فایل را دریافت می‌کند و مبادرت به ارسال آن می‌نماید.
توجه: در ادامه جهت «**جستجوی ورودی و خروجی**» مورد نظر فرآیند، توسط system call ، لایه نرم افزار ورودی و خروجی مستقل از دستگاه فراخوانی می‌شود.

نرم افزار ورودی و خروجی مستقل از دستگاه

اگرچه قسمتی از نرم افزار ورودی و خروجی وابسته به دستگاه است، ولی بخش قابل توجهی از آن به صورت مستقل از دستگاه می‌باشد. مرز بین نرم افزار گرداننده دستگاه و نرم افزار مستقل از دستگاه به سیستم و دستگاه وابستگی دارد. زیرا کارهای چندی که می‌توانند به صورت مستقل از دستگاه انجام شود، برای کارایی بیشتر و یا به دلایل دیگر در گرداننده دستگاه انجام می‌شود.
به طور کلی اعمال زیر در لایه نرم افزار مستقل از دستگاه قابل اجرا می‌باشند:

۱- رابط یک شکل و یکنواخت برای گردانندههای دستگاه

وظیفه اصلی نرم افزار ورودی و خروجی مستقل از دستگاه مهیا کردن توابع ورودی و خروجی مشترک برای تمام دستگاه‌ها و ایجاد یک رابط یک شکل برای استفاده در نرم افزار سطح کاربردی است. در واقع نویسندگان گردانندهها خود را با پارامترهای واسط یکسان و استاندارد تطابق می‌دهند. برای مثال یک برنامه که یک فایل را به عنوان ورودی می‌خواند، باید قادر به خواندن فایل از روی یک دیسک نرم یا سخت باشد بدون اینکه نیازی به تغییر برنامه برای هر نوع دستگاه متفاوت وجود داشته باشد. پلی‌مورفیسم (که در زبان یونانی به معنی «many forms» است) کیفیتی است که این امکان را به وجود می‌آورد تا یک نام، برای یک، دو، یا چندین تکنیک با منظورهای متفاوت ولی در یک راستا و در ارتباط با یک مفهوم، مورد استفاده قرار گیرد. پلی‌مورفیسم (سر بارگذاری تابع) آن گونه که در برنامه‌نویسی شیءگرا به کار گرفته شده است، این امکان را فراهم می‌کند تا بتوان برای مشخص کردن یک تابع عمومی حاوی عملیات مختلف، تنها از یک نام استفاده کرد. در این تابع عمومی نحوه انتخاب هر یک از عملیات مورد نظر صرفاً به وسیله نوع دادهای که مورد استفاده قرار می‌گیرد، تعیین می‌شود.
برای مثال در زبان C که در آن پلی‌مورفیسم مورد تاکید و اهمیت قرار نگرفته است، برای محاسبه قدرمطلق یک مقدار، سه تابع ()abs، ()labs و ()fabs وجود دارد. این توابع به ترتیب مقدار قدر مطلق مقادیر عدد صحیح، عدد صحیح از نوع long و ممیز شناور را بر می‌گردانند. در حالی که در C++ به لحاظ تاکید روی پلی‌مورفیسم هر یک از توابع فوق می‌توانند با نام ()abs

khalilifar.ir

مورد استفاده قرارگیرند، نوع داده‌ای که به عنوان آرگومان تابع ()abs مورد استفاده قرار می‌گیرد، گونه خاصی از تابع را انتخاب می‌کند. به بیان کلی‌تر مفهوم پلی مورفیسم همان ایده «**one interface, multiple methods**» است. بدین ترتیب امکان طراحی یک واسط کاربر کلی (generic interface) برای گروهی از فعالیت‌های مرتبط فراهم می‌شود، در حالی که انتخاب نحوه اجرای هریک از فعالیت‌ها صرفاً بستگی به نوع داده مورد استفاده خواهد داشت.
یکی از مزیت‌های پلی‌مورفیسم، کمک در کاهش پیچیدگی برنامه است، این عمل با استفاده از بکارگیری یک واسط کاربر یکسان برای مجموعه‌ای از عملیات تحت یک نام عمومی فراهم شده است.

۲- نام‌گذاری یکسان

چگونگی نام‌گذاری دستگاه‌های ورودی و خروجی، جنبه دیگری از داشتن رابط تقریبا یکسان را در دستگاه‌ها نشان می‌دهد. نگاشت اسامی سمبولیک دستگاه به گرداننده مناسب آن، در نرم افزار مستقل از دستگاه صورت می‌گیرد. در واقع نام‌های سیستمی دستگاه‌ها مورد استفاده قرار می‌گیرد و نه نامی که توسط کاربران مشخص شده است. برای مثال در سیستم عامل unix نام دستگاهی مانند «/dev/disk0» به طور منحصر به فرد شامل شماره دستگاه اصلی (Major device number) است که جهت یافتن گرداننده مناسب آن دستگاه بکار می‌رود و همچنین حاوی شماره دستگاه فرعی (Minor device number) می‌باشد که به عنوان پارامتری به گرداننده داده می‌شود تا دستگاه فرعی مورد نظر جهت اجرای عملیات خواندن و نوشتن بر روی آن تعیین گردد. تمام دستگاه‌ها دارای چنین شماره اصلی و فرعی می‌باشند و تمام گردانندهای دستگاه‌ها از طریق شماره اصلی مورد دستیابی قرار می‌گیرند.

۳- حفاظت

آنچه در ارتباط با نام‌گذاری مطرح است، «**مسئله حفاظت**» می‌باشد. چگونه سیستم، کاربرانی را که حق دستیابی به دستگاه را ندارند منع نماید؟ در سیستم عامل‌های windows و unix شیوه‌های امنیتی متفاوتی حاکم است. فایل‌های مخصوص (Special files) متناظر با دستگاه‌های ورودی و خروجی در unix با استفاده از بیت‌های حفاظتی (برای مثال بیت‌های rwx در unix) مورد حفاظت قرار می‌گیرند و تمام قوانین حفاظتی فایل‌ها برای دستگاه‌های ورودی و خروجی نیز به کار گرفته شده و قابل اعمال می‌باشد.

۴- بافر کردن

بافربندی موضوع دیگری است که به دلایل متفاوت هم در دستگاه‌های بلوکی و هم در دستگاه‌های کاراکتری مورد بحث قرار می‌گیرد. فرض کنید فرآیندی داده‌ای را از مودم می‌خواهد بخواند، فرآیند، فراخوان سیستمی Read را درخواست نماید و در انتظار یک کاراکتر ورودی مسدود شود

khalilifar.ir

و سپس با ورود هر کاراکتر یک وقفه رخ دهد و روتین سرویس‌دهنده وقفه کاراکتر را تحویل فرآیند کاربر داده و فرآیند را از حالت مسدود خارج نماید. بعد از اینکه کاراکتر در جایی ذخیره شد، فرآیند کاراکتر بعدی را می‌خواند و مجددا تا دریافت کاراکتر بعدی مسدود می‌شود. اشکالی که در این روش دیده می‌شود این است که فرآیند برای هر کاراکتر ورودی راه اندازی می‌گردد. در نتیجه تعداد دفعات اجرا بسیار زیاد اما مدت اجرا کوتاه مدت و ناکارآمد می‌باشد. اصلاح این روش می‌تواند به این صورت باشد که فرآیند، بافری n کاراکتری در فضای کاربر تهیه نماید و سپس n کاراکتر را بخواند. روتین سرویس‌دهنده وقفه کاراکترهای ورودی را یک به یک در بافر قرار می‌دهد تا کاملا پُر شود و سپس فرآیند کاربر را بیدار می‌کند.

۵- گزارش خطا

بسیاری از خطاها وابسته به دستگاه هستند و باید توسط گرداننده‌های دستگاه کنترل، کشف، تصحیح و پردازش گردند، اما گزارش خطا در بخش مستقل از دستگاه قرار دارد. یک خطای متعارف زمانی رخ می‌دهد که بلوک دیسک خراب شده باشد و نتواند خوانده شود. پس از اینکه گرداننده دستگاه به تعداد مشخصی، سعی در خواندن آن بلوک کند و در این کار موفق نباشد، نرم افزار مستقل از دستگاه را از وجود این خطا آگاه می‌سازد. چگونگی اداره خطا از این مرحله به بعد مستقل از دستگاه خواهد بود. به عبارت دیگر، تنها اگر لایه‌های پایینی قادر به حل مشکل به صورت شفاف نباشند، لایه‌های بالایی باید درگیر اداره خطا شوند.

۶- تخصیص و آزادسازی دستگاه‌های اختصاصی

دستگاه‌های انحصاری (اختصاصی یا غیراشتراکی) مانند نوارمغناطیسی در هر لحظه، فقط می‌توانند مورد استفاده یک فرآیند قرار گیرند و تا انتهای کار هر فرآیند، فقط به همان فرآیند اختصاص می‌یابند. و اگر منبع انحصاری به یک فرآیند تعلق گیرد دیگر نمی‌توان به زور منبع را از فرآیند پس گرفت و باید خودش داوطلبانه منبع را رها سازد. اگر فرآیندی منبعی انحصاری را درخواست کند که از قبل در اختیار فرآیند دیگری قرار گرفته است، آنگاه سیستم عامل فرآیند درخواست‌کننده منبع را در انتهای صف انتظار منبع قرار می‌دهد. پس از رها شدن منبع مورد درخواست توسط فرآیند جاری، فرآیندهای خوابیده در صف انتظار منبع، به ترتیب و یک به یک از ابتدای صف، منبع مورد درخواست را در اختیار می‌گیرند و کار خود را ادامه می‌دهند. تخصیص دستگاه‌های اختصاصی بر عهده نرم افزار مستقل از دستگاه در سیستم عامل است.

۷- اداره صف‌ها و زمان‌بندی

اگر در هنگام درخواست یک منبع (دستگاه) اختصاصی، آن منبع در اختیار فرآیند دیگری باشد، درخواست جدید باید در صف انتظار منبع مورد درخواست قرار داده شود. مدیریت صف‌های منابع اختصاصی بر عهده نرم افزار مستقل از دستگاه در سیستم عامل است.

khalilifar.ir

۸- تعیین اندازه بلوک‌های داده مستقل از دستگاه

اندازه بلوک‌ها در دیسک‌های متفاوت ممکن است مختلف باشد و این امر بر عهده نرم افزار مستقل از دستگاه است که با ایجاد «**یک اندازه بلوک یکسان**» برای لایه‌های بالاتر این مشکل را حل نماید. به عنوان مثال ممکن است، چندین بلوک فیزیکی به عنوان یک بلوک منطقی تعیین گردد. بدین ترتیب لایه‌های بالاتر فقط از دستگاه‌های انتزاعی که همگی دارای اندازه بلوک منطقی یکسانی هستند، استفاده می‌نمایند. اندازه بلوک منطقی مستقل از اندازه بلوک فیزیکی می‌باشد. به عبارت دیگر اندازه بلوک‌ها در دیسک‌های مختلف یکسان نیست. پنهان کردن این موضوع و ایجاد بلوک‌های هم اندازه برای لایه‌های بالاتر، بر عهده نرم افزار مستقل از دستگاه است.

توجه: در ادامه جهت «**جستجوی بیشتر ورودی و خروجی**» مورد نظر فرآیند، توسط لایه نرم‌افزار ورودی و خروجی مستقل از دستگاه ، لایه گرداننده دستگاه (Device Driver) فراخوانی می‌شود.

گرداننده دستگاه (Device Driver)

توجه: در حالت کلی، کار گرداننده دستگاه، پذیرش درخواست‌های لایه نرم افزار ورودی و خروجی مستقل از دستگاه و مراقبت از اجرا شدن درخواست‌ها است. برای مثال یکی از درخواست‌های متداول برای یک گرداننده دستگاه دیسک، خواندن بلوک‌های دیسک است.

توجه: همانطور که گفتیم هر کنترل‌کننده دستگاه از چندین رجیستر (جهت دریافت فرمان و یا به منظور تعیین وضعیت دستگاه) برخوردار است. تعداد این رجیسترها و طبیعتا فرمان‌ها از دستگاهی به دستگاه دیگر متفاوت است. برای مثال **گرداننده** ماوس اطلاعاتی را که با حرکت دادن ماوس از چپ به راست و یا بالعکس و یا همچنین با فشار دادن دکمه‌ها در ماوس صورت می‌گیرد، دریافت می‌کند. در حالی که گرداننده دیسک باید از قطاع‌ها، شیارها، سیلندرها و حرکت بازوی دیسک اطلاع داشته باشد.

توجه: همه دستگاه‌های I/O متصل به یک کامپیوتر نیازمند یک کد خاص یعنی **گرداننده دستگاه** (Device Driver) برای کنترل آن دستگاه هستند. این کد توسط سازندگان دستگاه و برای انواع سیستم عامل‌ها نوشته می‌شود. به تفاوت گرداننده دستگاه و کنترل کننده دستگاه دقت نمایید. گرداننده دستگاه به کنترل کننده دستگاه فرمان می‌دهد.

توجه: گردانندهای دستگاه‌ها به دلیل دسترسی مستقیم به سخت افزار دستگاه (رجیسترهای کنترل کننده دستگاه) مجبورند که بخشی از هسته سیستم عامل را تشکیل دهند. البته امکان این وجود دارد که گردانندهها در سطح کاربر به کمک یک سری فراخوان‌های سیستمی جهت نوشتن و خواندن از رجیسترها اجرا شوند. در حقیقت این طراحی (طراحی میکروکرنل) می‌تواند ایده خوبی باشد، زیرا هسته از گردانندهها جدا می‌گردد و چنانچه خطایی در گرداننده ایجاد گردد هسته را متاثر نمی‌نماید و باعث فروپاشی (Crash) سیستم نخواهد شد.

توجه: به طور کلی سیستم عامل‌ها گردانندهای دستگاه‌های I/O را به دو گروه دستگاه‌های **بلوکی**

khalilifar.ir

و دستگاه‌های **کاراکتری** تقسیم می‌کنند. دستگاه‌های بلوکی آدرس‌پذیر هستند مانند انواع دیسک‌های ذخیره‌سازی و دستگاه‌های کاراکتری آدرس‌پذیر نیستند و قابلیت هیچگونه عمل جستجویی را فراهم نمی‌کنند مانند کیبورد، ماوس، چاپگرهای خطی و سایر دستگاه‌هایی که عملکردشان شبیه دیسک‌ها نمی‌باشد.

توجه: گرداننده دستگاه دارای وظایف متعددی است اما مهمترین وظیفه آن، پذیرش تقاضای خواندن و نوشتن انتزاعی از کد نرم افزار مستقل از دستگاه در لایه بالایی خود و نیز نظارت و کنترل اجرای آنها می‌باشد. علاوه بر این، وظایف دیگری را مثل راه اندازی دستگاه‌ها و مدیریت و ثبت حوادث (Log event) را نیز به عهده می‌گیرد. در صورتی که گرداننده دستگاه در لحظه رسیدن تقاضا بیکار باشد، بلافاصله به تقاضا پاسخ می‌دهد در غیر اینصورت تقاضای جدید را به صف تقاضاهای منتظر اضافه می‌نماید تا هرچه زودتر به آنها رسیدگی شود. روش کار جهت پاسخ‌دهی به یک تقاضا بدین صورت است که ابتدا صحت پارامترهای ورودی توسط گرداننده بررسی می‌شود. اگر پارامترها صحیح نباشند، خطا برگردانده می‌شود. چنانچه پارامترها صحیح باشند، اولین گام جهت اجرای تقاضای ورودی و خروجی «**تبدیل تقاضا از فرم انتزاعی به فرم واقعی**» آن می‌باشد، به عبارت دیگر محل بلوک مورد تقاضا (شماره سیلندر و قطاع) بر روی دیسک تعیین می‌شود.

سپس گرداننده، وضعیت دستگاه را بررسی می‌کند. اگر مشغول باشد، تقاضا در صف قرار می‌گیرد، در غیر اینصورت، چک می‌شود که آیا وضعیت سخت افزاری به گونه‌ای است که بتوان تقاضا را هم اکنون پردازش کرد؟ ممکن است قبل از شروع انتقال داده، لازم باشد دستگاه را روشن کند و موتور را به راه اندازد. زمانی که دستگاه روشن می‌شود و آماده کار می‌گردد، کنترل واقعی می‌تواند شروع شود. منظور از کنترل، صدور دنباله‌ای از فرمان‌ها به دستگاه است. برنامه گرداننده دستگاه، باید معین کند چه عملیاتی توسط کنترل کننده و به چه ترتیبی باید انجام پذیرد. بعد از مشخص شدن این که چه فرامینی باید به کنترل کننده صادر گردد، برنامه گرداننده دستگاه با نوشتن فرامین در رجیسترهای دستگاه کنترل کننده، شروع به ارسال فرامین خواهد کرد.

توجه: بعضی از کنترل کننده‌ها، در هر لحظه یک فرمان را از گرداننده دستگاه دریافت کرده و اجرا می‌نمایند و زمانی که آماده اجرای فرمان بعدی باشند، گرداننده فرمان بعدی را به آنها تحویل می‌دهد و این کار تا آخرین فرمان یک به یک صورت می‌گیرد. برخی دیگر از کنترل کننده‌ها قادرند فهرستی از فرامین را بپذیرند و پس از آن بدون نیاز به گرداننده به ترتیب دستورات را اجرا نمایند. بعد از اینکه فرمان‌ها صادر شدند یکی از دو حالت ذیل پیش می‌آید:

۱- در بسیاری از مواقع، گرداننده دستگاه باید در انتظار بماند تا کنترل کننده، کارهای مورد درخواست را انجام دهد، بنابراین خود را مسدود می‌نماید و زمانی که وقفه‌ای فرا رسید، از حالت مسدود خارج می‌شود.

khalilifar.ir

۲- در سایر حالات بدون تاخیر سریعا کار انجام می‌شود و نیازی به مسدود شدن گرداننده نیست. مانند پیمایش صفحه در برخی از پایانه‌ها، که فقط با نوشتن تعدادی بایت در رجیسترهای کنترل‌کننده انجام پذیر است و حرکت مکانیکی در این راستا صورت نمی‌پذیرد. بنابراین، کل کار در چند میکروثانیه انجام می‌شود.

توجه: در هر دو حالت پس از پایان عملیات، باید خطاهای احتمالی بررسی شوند. اگر همه چیز صحیح باشد، گرداننده می‌تواند داده را به نرم افزار مستقل از دستگاه ارسال نماید. اگر تقاضاهای دیگر در صف انتظار دستگاه وجود داشته باشد در آن لحظه، یکی از آنها انتخاب می‌شود و به تقاضای آن رسیدگی می‌گردد. در صورتی که صف انتظار خالی باشد، گرداننده دستگاه به حالت مسدود در می‌آید و منتظر تقاضای بعدی می‌ماند.

توجه: در ادامه جهت «**تحویل ورودی و خروجی**» مورد نظر فرآیند، توسط لایه گرداننده دستگاه (Device Driver)، لایه روتین سرویس‌دهنده وقفه فراخوانی می‌شود.

روتین سرویس‌دهنده وقفه (اداره‌کننده وقفه)

پس از تکمیل شدن عملیات ورودی و خروجی، از سوی کنترل‌کننده دستگاه مورد نظر وقفه خروج از I/O رخ می‌دهد، که منجر به فعال شدن روتین سرویس‌دهنده وقفه (اجرای ISR خروج از I/O) می‌شود، این روتین کارش بیدارکردن لایه‌های I/O قبل به صورت یک به یک و رو به بالا و در نهایت «**تحویل**» ورودی و خروجی مورد نظر به فرآیند کاربر است. روند و جزئیات خوابیدن و بیدارشدن یک به یک کل لایه‌های I/O از بالا به پایین و از پایین به بالا و انجام وقفه از سوی برنامه‌های کاربردی پنهان است و فقط نتایج به برنامه کاربردی «**تحویل**» داده می‌شود.

khalilifar.ir

khalilifar.ir

تست‌های کنکور کارشناسی ارشد سال ۱۴۰۰

۱- کدام سطح از RAID را Disk mirroring می‌گویند؟ (مهندسی کامپیوتر – دولتی ۱۴۰۰)

۱) 0 ۲) 1 ۳) 2 ۴) 3

۲- کدام مورد، سیستم عامل را مجبور می‌کند دستورات S_1، S_2، S_3 و S_4 که به ترتیب در پردازه‌های همروند P_1 ، P_2 ، P_3 و P_4 قرار دارند به همان ترتیب S_1، S_2، S_3 و S_4 اجراکند؟

(مقدار اولیه سمافورها $a = b = c = 0$) (مهندسی کامپیوتر – دولتی ۱۴۰۰)

P_1	P_2	P_3	P_4
S_1 signal (a)	wait (a) S_2 signal (b)	wait (b) S_3 signal (c)	۱) wait (c) S_4
S_1 signal (a) signal (b)	wait (b) S_2 signal (a)	wait (a) S_3 signal (b)	۲) wait (a) wait (b) S_4
S_1 signal (a)	wait (a) S_2 signal (a) signal (a)	wait (a) wait (a) S_3 signal (a) signal (a) signal (a)	۳) wait (a) wait (a) wait (a) S_4 signal (a) signal (a) signal (a) signal (a)
S_1 wait (a) signal (b) signal (c)	wait (a) S_2 signal (b) signal (c)	wait (a) signal (b) S_3 signal (c)	۴) wait (a) signal (b) signal (c) S_4

۳- فرض کنید که طول آدرس مجازی 47 بیت و اندازه صفحه 16 KB و هر مدخل از جدول صفحه 8 بایت باشد. اگر بخواهیم هر جدول صفحه تنها در یک صفحه ذخیره شود، از جدول صفحه چند سطحی استفاده شود؟ (مهندسی کامپیوتر – دولتی ۱۴۰۰)

۱) 2 ۲) 3 ۳) 4 ۴) 5

khalilifar.ir

۴- الگوریتم زیر برای حل مسئله ناحیه بحرانی (Critical - Problem) را در نظر بگیرید. در این الگوریتم، در حالتی که تنها دو پردازه P_0 و P_1 وجود داشته باشد، متغیرهای flag و turn بین این دو پردازه مشترک هستند:

```
boolean flag [2]; /*initially false */
int turn;
```

با فرض اینکه ساختار پردازه P_i ($i = 0$ OR 1) به صورت زیر باشد، کدام گزینه صحیح است؟

(مهندسی کامپیوتر – دولتی ۱۴۰۰)

```
do {
        flag [i] = true;
        while (flag [j]) {
                if (turn = = j) {
                        flag [i] = false;
                        while (turn = = j);
                                /* do nothing */
                          flag [i] = true;
        }
}
/ * critical section */
turn = j;
flag [i] = false;
/ * reminder section */
} while (true);
```

۱) شرط پیشرفت ممکن است نقض شود.

۲) شرط انتظار محدود ممکن است نقض شود.

۳) شرط انحصار متقابل ممکن است نقض شود.

۴) هر سه شرط انحصار متقابل، انتظار محدود و پیشرفت همواره تضمین می‌شود.

۵- یک کامپیوتر دارای m چاپگر از یک نوع است. این چاپگرها به وسیله 3 پردازه A و B و C استفاده می‌شوند که در زمان بیشترین نیاز (حداکثر تقاضا) به ترتیب به 3 و 4 و 6 چاپگر نیاز دارند. کمترین مقدار m که برای آن هیچ وقت در این کامپیوتر بن‌بست پیش نیاید چند است؟

(مهندسی کامپیوتر – دولتی ۱۴۰۰)

۱) 10 ۲) 11 ۳) 12 ۴) 13

۶- دو پردازه متناوب با مشخصات زیر مفروض است. کدام گزینه بزرگترین مقدار x را برای پردازه 2 نشان می‌دهد به نحوی که زمان‌بندی قبضه‌ای (نرخ یکنواخت) Rate Monotonic امکان‌پذیر باشد؟

(مهندسی کامپیوتر – دولتی ۱۴۰۰)

	Period	CPU Time
P_1	50	25
P_2	80	x

۱) 20 ۲) 25 ۳) 30 ۴) 35

khalilifar.ir

۷- در یک الگوریتم برنامه‌ریزی اولویت‌دار که پنج پردازه و اولویت‌های آن‌ها به صورت زیر است، وجود دارد. میانگین زمان انتظار چند میلی ثانیه است؟ (مهندسی کامپیوتر – دولتی ۱۴۰۰)

فرض کنید که هرچه مقدار اولویت کمتر باشد، اولویت پردازه بیشتر است.
یعنی پردازه P_4 دارای کمترین اولویت و پردازه P_2 دارای بیشترین اولویت است.

پردازه	زمان	اولویت
P_1	10 ms	3
P_2	1 ms	1
P_3	2 ms	4
P_4	1 ms	5
P_5	5 ms	2

۱) 7 ms ۲) 8 ms ۳) 8.2 ms ۴) 7.75 ms

۸- کدام الگوریتم جایگزینی صفحه از ناهنجاری Belady رنج می‌برد؟ (مهندسی IT – دولتی ۱۴۰۰)

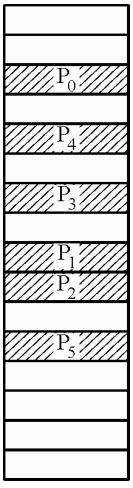
۱) LRU ۲) FIFO ۳) MRU ۴) LIFO

۹- یک کامپیوتر دارای 6 چاپگر است و n پردازه در کامپیوتر برای به دست آوردن این چاپگرها رقابت می‌کنند. هر کدام از پردازه‌ها به 3 چاپگر نیاز دارند. بیشترین مقدار n که تضمین نماید سیستم بدون بن‌بست است، چند است؟ (مهندسی IT – دولتی ۱۴۰۰)

۱) 1 ۲) 2 ۳) 3 ۴) 4

۱۰- کدام گزینه از مزایای ساختار سیستم عامل لایه‌ای (Layered) نسبت به ساختار سیستم عامل یکپارچه (Monolithic) نیست؟ (مهندسی IT – دولتی ۱۴۰۰)

۱) خطایابی ساده‌تر
۲) مدیریت ساده‌تر
۳) سرعت بیشتر
۴) قابلیت گسترش بیشتر



۱۱- در یک سیستم که تخصیص حافظه در آن بر اساس صفحه‌بندی (Paging) انجام می‌شود، اندازه‌ی هر فریم (Frame) برابر 2 kbyte (2048 byte) است. شکل مقابل، حافظه‌ی اصلی سیستم را نشان می‌دهد. قسمت‌های خاکستری فریم‌های تخصیص داده شده به یک پردازه هستند. اگر در حین اجرای پردازه، پردازنده آدرس 7000 را تولید کند، چه آدرسی از حافظه‌ی اصلی دسترسی خواهد شد؟ (مهندسی IT – دولتی ۱۴۰۰)

۱) 7000 ۲) 9048 ۳) 13144 ۴) 17240

khalilifar.ir

۱۲- برنامه زیر از 3 پردازه همروند تشکیل شده است. و این 3 پردازه از 3 سمافور باینری که به‌صورت زیر مقداردهی شده‌اند، استفاده می‌کنند. دستور print ('HELLO') چند بار اجرا می‌شود؟ (مهندسی IT – دولتی ۱۴۰۰)

S_0=1 ،S_1=0 ،S_2=0

Proccess P_0:
```
while (true) {
    wait(S0);
     print ('HELLO');
     signal (S1);
     signal (S2);
}
```

Process P_1:
```
wait(S1);
signal (S0);
```

Proccess P_2:
```
wait(S2);
signal (S0);
```

۱) دقیقاً یک‌بار
۲) دقیقاً 2 بار
۳) دقیقاً 3 بار
۴) حداقل 2 بار

۱۳- فرض کنید زمان رسیدن و مدت زمان اجرای سه پردازه به صورت زیر باشد:
بیشینه زمان چرخش (Turnaround Time) برای الگوریتم SJF (Shortest Job Firs) بدون قبضه (non preemptive) کدام است؟ (مهندسی IT – دولتی ۱۴۰۰)

Process	Arrival Time	Burst Time
P1	1	14
P2	3	7
P3	2	7

۱) 14 ms ۲) 21 ms ۳) 28 ms ۴) 31 ms

۱۴- الگوریتم زیر که تغییر یافته الگوریتم پترسون برای حل ناحیه بحرانی است برای دو فرآیند i و j ارائه شده‌است. (مشابه آن برای j وجود دارد که اندیس‌های i و j تعویض می‌گردد.) کدام گزینه درست است؟ (فرض کنید که الگوریتم پترسون درست عمل می‌کند) (مهندسی IT – دولتی ۱۴۰۰)

Process i
```
do {
      .
      .
       turn = i;
       while (turn = = j);
      / * critical section */
       turn = j;
     / * reminder section */
} while (true);
```

۱) دقیقا یک شرط لازم حل ناحیه بحرانی را نقض می‌کند.
۲) دقیقا دو شرط لازم حل ناحیه بحرانی را نقض می‌کند.
۳) هر سه شرط حل ناحیه بحرانی را نقض می‌کند.
۴) شروط لازم ناحیه بحرانی را دارد و راه حل مناسب است.

khalilifar.ir

پاسخ تست‌های کنکور کارشناسی ارشد سال ۱۴۰۰

۱- گزینه (۲) صحیح است.

هدف اصلی تکنیک RIAD (Redundant Array of Independent Disks) به معنی «**آرایه افزونه از دیسک‌های مجزا**»، پیوند دادن چند دیسک سخت جداگانه در چهارچوب یک آرایه (array) برای دستیابی به **افزایش کارایی خواندن و نوشتن (read & write performance)** یعنی افزایش سرعت خواندن و نوشتن یا همان افزایش نرخ انتقال خواندن و نوشتن، **افزایش قابلیت اطمینان (reliability) یا همان افزایش تحمل‌پذیری در برابر خطا (fault tolerance)** و **افزایش فضای ذخیره‌سازی** به معنی گنجایشی بیش از یک دیسک بزرگ و گران می‌باشد. همچنین در نهایت کل این آرایه برای سیستم عامل میزبان، به گونه‌ای یکپارچه رفتار می‌کند.

می‌دانیم که نرخ بهبود کارایی سرعت دیسک، نسبت به نرخ افزایش کارآمدی پردازنده و حافظه اصلی به طور قابل ملاحظه‌ای کمتر بوده‌است. و این عدم تطابق باعث شده که برای بهبود کارآمدی عمومی سیستم کامپیوتری، دیسک سخت در کانون توجه قرار گیرد. همانند سایر حوزه‌های کارایی کامپیوتر، طراحان دیسک سخت دریافته‌اند که اگر مولفه‌ای فقط تا حدی می‌تواند به جلو رانده شود، یعنی در حرکت رو به جلو مقاومت دارد، تقویت بیشتر کارایی با استفاده از مولفه‌های موازی بیشتر خواهد بود. در مورد دیسک سخت نیز این موضوع منجر به ایجاد آرایه‌ای از دیسک‌ها شده‌است که به طور مستقل و موازی عمل می‌کنند. با دیسک‌های چندگانه، درخواست‌های ورودی و خروجی جداگانه می‌توانند تا جایی که داده‌های مورد نیاز بر روی دیسک‌های جداگانه قرار داشته باشند، به طور موازی به اجرا در آیند. از آن گذشته، یک درخواست ورودی و خروجی هم می‌تواند به صورت موازی انجام گیرد، به شرطی که بلوک داده‌های مورد دسترسی، بر روی دیسک‌های مختلف توزیع شده باشد. با استفاده از دیسک‌های چندگانه، راه‌های مختلفی برای سازماندهی داده‌ها روی دیسک‌های مختلف و اضافه‌کردن افزونگی برای بهبود قابلیت اطمینان (پشتیبان اطلاعات) ایجاد می‌شود. این موضوع می‌تواند ایجاد طرح‌های بانک اطلاعاتی قابل استفاده روی کامپیوترها و سیستم‌های عامل متعدد را سخت کند. خوشبختانه صنعت کامپیوتر، روی طرح استانداردی برای طراحی بانک اطلاعاتی دیسک‌های چندگانه توافق کرده‌است و این طرح به نام RAID خوانده می‌شود. طرح RAID شامل هفت سطح (0 تا 6) است. البته سطوح بیشتری توسط برخی طراحان یا شرکت‌ها تعریف شده است. ولی هفت سطحی که در اینجا مطرح شده مورد توافق همگان است. این سطوح بیان کننده رابطه سلسله مراتبی نیستند، بلکه مشخص‌کننده معماری‌های مختلفی هستند که در سه خصوصیت مشترکند:

خصوصیت اول: RAID مجموعه‌ای از گردانندهای دیسک فیزیکی است که از نظر سیستم عامل به صورت یک واحد گرداننده دیسک منطقی دیده می‌شود.

خصوصیت دوم: داده‌ها بر روی گردانندهای فیزیکی یک آرایه توزیع می‌شوند.

khalilifar.ir

خصوصیت سوم: ظرفیت افزونگی دیسک (پشتیبان اطلاعات) برای ذخیره داده‌های افزونه (داده‌های توازن یا کنترلی) به کار گرفته می‌شود. این اطلاعات قابلیت بازیابی داده‌ها در شرایط وقوع حادثه یا خرابی دیسک را تضمین و گارانتی می‌کنند.

توجه: جزئیات خصوصیات دوم و سوم در سطوح مختلف RAID متفاوت است. RAID 0 خصوصیت سوم را پشتیبانی نمی‌کند.

توجه: در تکنیک RIAD کارت کنترل‌کننده دیسک با یک کنترل‌کننده RAID جایگزین شده و داده در RAID کپی می‌شود و سپس عملیات عادی ادامه پیدا می‌کند.

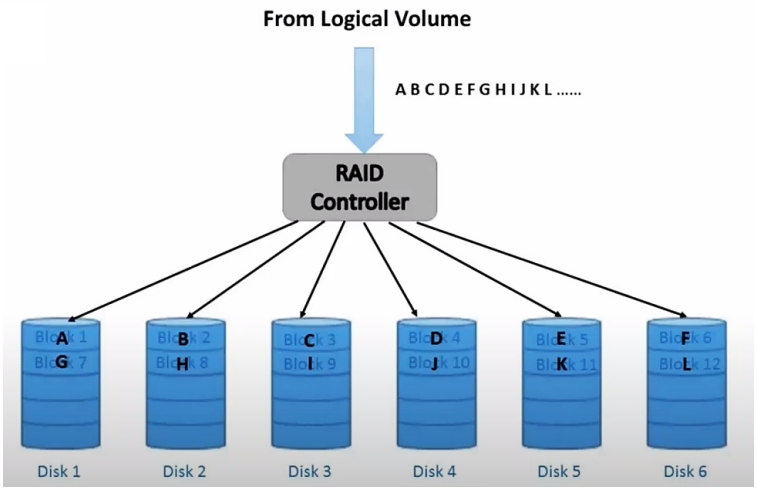
واژه RAID ابتدا در مقاله‌ای توسط گروهی از **پژوهشگران دانشگاه کالیفرنیا در برکلی** به کار برده شد. در این مقاله سرواژه RAID برای عبارت (Redundant Array of Inexpensive Disks) به معنی «**آرایه افزونه از دیسک‌های ارزان**»، به کار رفته است. عبارت ارزان قیمت برای نشان دادن دیسک‌های ارزان قیمت آرایه RAID در مقابل یک دیسک بزرگ و گران قیمت واحد استفاده شده است. در واقع در گذشته، RAID از دیسک‌های کوچک و ارزانی تشکیل می‌شد که به عنوان یک جایگزین کم‌هزینه برای دیسک‌های بزرگ و گران، به کار برده می‌شدند. اما گران بودن دیسک‌ها مربوط به گذشته است و امروزه دیسک‌های بزرگ نیز ارزان قیمت هستند. بنابراین امروزه، صنعت واژه «**Independent**» را به جای واژه «**Inexpensive**» به کار می‌برد. تا بر افزایش نرخ انتقال از طریق موازی‌سازی و افزایش قابلیت اطمینان از طریق افزونگی آرایه RAID بیشتر تاکید کند. مقاله مزبور پیکربندی‌ها و کاربردهای متعدد RAID را ارائه می‌کند و به معرفی سطوح مختلف RAID می‌پردازد که هنوز هم مورد استفاده است. راهبرد پیشنهادی RAID عبارت از جایگزینی گرداننده‌های دیسک دارای ظرفیت زیاد با گرداننده‌های چندگانه کم ظرفیت است و داده‌ها به گونه‌ای توزیع می‌شوند که دسترسی موازی و همزمان به آنها از گرداننده‌های چندگانه امکان‌پذیر باشد. به این ترتیب کارایی خواندن و نوشتن افزایش پیدا می‌کند و افزایش تدریجی ظرفیت با سهولت بیشتری انجام می‌گیرد. گرچه عملیات همزمان و موازی هدها و محرک‌ها، باعث خواندن و نوشتن سریعتر و افزایش نرخ انتقال می‌شود، اما استفاده از گرداننده‌های چندگانه احتمال شکست کلیت راهبرد را افزایش می‌دهد، چون داده‌ها پراکنده شده‌اند و وقتی بخشی از داده‌ها از دسترس خارج شود آنگاه کل اطلاعات ناخوانا و از دسترس خارج می‌شود و نامعتبر می‌شود. احتمال اینکه تعدادی دیسک از یک مجموعه از N دیسک خراب شوند، بسیار بالاتر از این احتمالی است که تنها یک دیسک، خراب شود. با توجه به میانگین زمان برای خرابی یک دیسک که 100,000 ساعت است، پس زمان میانگین برای خرابی و از کارافتادگی تعدادی دیسک در یک ردیف یا آرایشی از 100 دیسک، برابر با $\frac{100,000}{100}=1000$ ساعت یا حدود $42\simeq\frac{1000}{24}$ روز است که چندان هم طولانی نیست، پس اگر فقط یک نسخه از داده‌های اصلی روی دیسک‌ها ذخیره شود و هیچ پشتیبان اطلاعاتی از آنها نباشد، آنگاه چون داده‌ها توزیع شده‌اند اگر بخشی از اطلاعات از دسترس خارج شود آنگاه کل اطلاعات ناخوانا و از دسترس خارج می‌شود و نامعتبر

khalilifar.ir

می‌شود، و این سرعت بالای گم شدن اطلاعات، غیر قابل قبول است. برای جبران این کاهش قابلیت اطمینان، RAID از داده‌های افزونه (اطلاعات کنترلی) ذخیره شده روی دیسک‌ها استفاده می‌کند که بازیابی داده‌های از دست رفته از خرابی یک دیسک را ممکن می‌سازد. در نتیجه، خرابی یک دیسک منجر به از دست دادن کل داده‌ها نمی‌شود.

چهار تکنیک در RAID مورد استفاده قرار می‌گیرد:

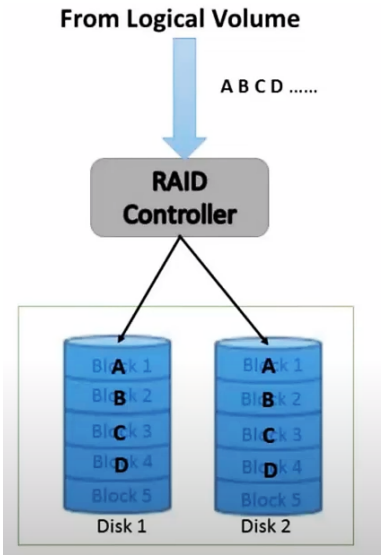
تکنیک Striping: جهت **افزایش کارایی خواندن و نوشتن (read & write performance)** یعنی افزایش سرعت خواندن و نوشتن یا همان افزایش نرخ انتقال خواندن و نوشتن، مورد استفاده قرار می‌گیرد. (Striping improves performance) در این تکنیک یعنی data striping، داده‌های اصلی به بخش‌های مختلف تکه تکه شده و تقسیم (splitting) می‌شود. و تکه‌ها به طور نوبت گردشی بین دیسک‌ها توزیع و پخش (spreading) می‌شوند. به دنباله‌ای سریالی از بخش‌های تقسیم شده داده‌های اصلی، که دقیقا هر بخش (باریکه) به یک دیسک از آرایه نگاشت می‌شود، یک **نوار** گفته می‌شود. در یک آرایه n دیسکی، n باریکه منطقی اول به طور فیزیکی به صورت اولین نوار بر روی n دیسک ذخیره می‌شود. n باریکه منطقی دوم، بر روی نوار دوم دیسک‌ها توزیع می‌شود و به همین ترتیب. مزیت این طرح این است که اگر اندازه یک درخواست خواندن و نوشتن متشکل از چند باریکه منطقی پیوسته و متوالی باشد، تا n باریکه از این درخواست می‌تواند به صورت موازی و همزمان انجام شود و نرخ انتقال خواندن و نوشتن افزایش پیدا کند و به تبع آن زمان انتقال خواندن و نوشتن کاهش یابد. در این حالت یک درخواست خواندن و نوشتن، موجب انتقال موازی داده‌ها از دیسک‌های چندگانه شده است، که نرخ انتقال آن، در مقایسه با انتقال از یک دیسک، افزایش یافته‌است.



khalilifar.ir

توجه: تکنیک Striping منجر به افزایش کارایی خواندن و نوشتن (read & write performance) یعنی افزایش سرعت خواندن و نوشتن یا همان افزایش نرخ انتقال خواندن و نوشتن می‌شود. در چنین ساختاری، همه دیسک‌ها، در هر درخواست خواندن و نوشتن مشارکت می‌کند، بنابراین، تعداد درخواست‌هایی که می‌توانند در هر ثانیه پردازش شود، مشابه با یک دیسک منفرد است، اما در هر درخواست، چند برابر، اطلاعات بیشتری در زمان مساوی با دیسک منفرد، خوانده می‌شود. برای مثال مطابق شکل فوق در یک درخواست واحد، کاراکترهای ABCDEF به طور همزمان و موازی خوانده یا نوشته می‌شود. این تقسیم در data striping می‌تواند در سطوح bit، byte یا block از یک فایل انجام شود. برای مثال در سطح bit اگر از یک آرایش 8 دیسکی برای فقط داده‌های اصلی استفاده شود، بیت i ام، از هر بایت به دیسک i ام منتقل می‌شود.

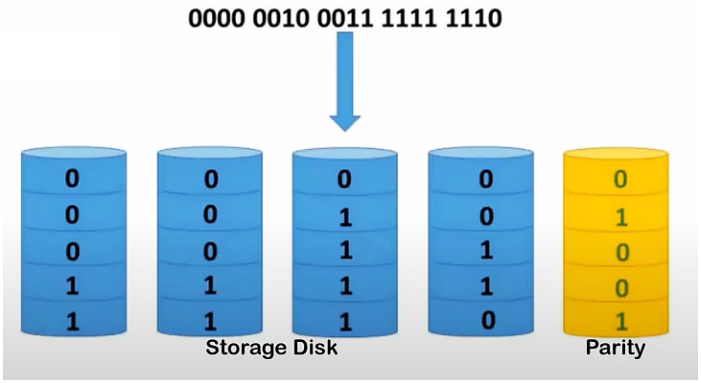
تکنیک Mirroring: جهت افزایش قابلیت اطمینان (reliability) مورد استفاده قرار می‌گیرد. در این تکنیک داده‌های اصلی روی دو دیسک، به طور یکسان نوشته می‌شوند. (mirrored set). در این تکنیک داده‌های افزونه از **تکرار کاملا یکسان** از داده‌های اصلی ایجاد می‌شود. (Mirroring provides data redundancy) در نتیجه هر دیسک آرایه، دارای یک دیسک آینه و حاوی همان داده‌هاست.



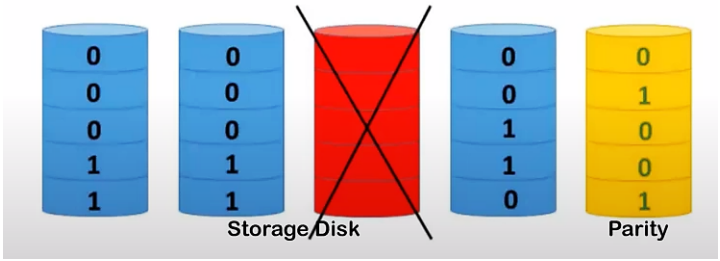
تکنیک Parity: جهت افزایش قابلیت اطمینان (reliability) مورد استفاده قرار می‌گیرد. در این تکنیک داده‌های افزونه از **محاسبه توازن** از داده‌های اصلی ایجاد می‌شود. در صورت استفاده از توازن زوج، بیت **parity** طوری مقداردهی می‌شود که تعداد **1** ها در کد، زوج بماند و در صورت استفاده از توازن فرد، بیت **parity** طوری مقداردهی می‌شود که تعداد **1** ها در کد، فرد بماند. اگر یکی از بیت‌ها در داده‌های اصلی هم ردیف خودش خراب شود، برای مثال یک بیت **1** به **0** و یا

khalilifar.ir

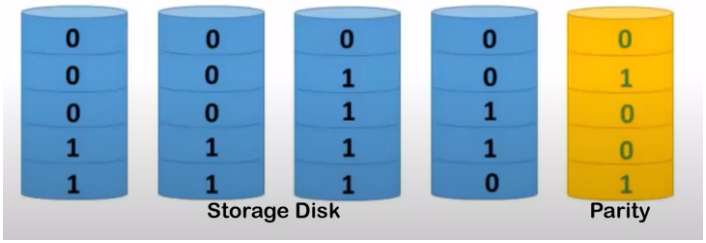
یک بیت **0** به **1** تبدیل شود، آنگاه توازن بیت‌های هم ردیف تغییر می‌کند و با توازن توافق شده و محاسبه شده، هماهنگ نیست. به طور مشابه، اگر خود بیت توازن ذخیره شده، خراب شود، آنگاه با توازن توافق شده و محاسبه شده، هماهنگ نیست. بنابراین همه خطاهای تک بیتی چه در داده‌های اصلی و چه در داده‌های افزونه (توازن) قابل کشف و تصحیح است. اگر توازن بیت‌های **1** باقی‌مانده، با توازن ذخیره‌شده سازگار باشد، بیت از دست رفته و خراب شده، مقدار **0** داشته و در غیر اینصورت مقدار **1** داشته‌است.



اگر یکی از دیسک‌ها خراب شود به صورت زیر:



امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه (توازن) وجود دارد، به صورت زیر:



تکنیک کد Hamming: جهت افزایش قابلیت اطمینان (reliability) مورد استفاده قرار می‌گیرد. در این تکنیک داده‌های افزونه از **محاسبه کد Hamming** از داده‌های اصلی ایجاد می‌شود.

khalilifar.ir

کد همینگ (Hamming code)

برای اینکه بتوان خطا در هر داده اطلاعاتی را تشخیص داد و یک بیت خطا در آن را تصحیح نمود. از روش کد همینگ استفاده می‌شود. کد همینگ را با یک مثال توضیح می‌دهیم:

فرض کنید داده اصلی 1001 است و طبق توافق قرار است از کد همینگ برای کشف خطا استفاده شود. برای اینکه بتوان در داده‌های اصلی خطا را تشخیص داد باید برای هر m بیت داده اصلی، r بیت داده کنترلی یا افزونه اضافه کرد. به این شکل که داده‌های کنترلی در بیت‌هایی با اندیس‌هایی از توان دو (1 و 2 و 4 و 8 و ...) و داده‌های اصلی در اندیس‌های باقی‌مانده (3 و 5 و 6 و 7) قرار می‌گیرند. به شکل زیر توجه کنید:

r_1	r_2	m_3	r_4	m_5	m_6	m_7
?	?	1	?	0	0	1

برای بدست آوردن مقادیر بیت‌های افزونه به شکل زیر عمل می‌شود:

ابتدا باید شماره اندیس بیت‌های داده اصلی را با استفاده از اعداد توان 2 بدست آوریم، به عنوان مثال عدد 7 از مجموع اعداد 4 و 2 و 1 که همه آنها اعداد توان 2 هستند، بدست می‌آید:

$1 + 2 + 4 = 7$

$2 + 4 = 6$

$1 + 4 = 5$

$1 + 2 = 3$

حال برای بدست آوردن مقدار بیت‌های افزونه کافی است مقدار اندیس‌های داده اصلی که اندیس بیت افزونه در بدست آوردن شماره اندیس آنها نقش داشته است، با هم XOR شوند و در اندیس مورد نظر قرار گیرند. به عنوان مثال برای بدست آوردن r_1 باید مقادیر بیت‌های با اندیس 7 و 5 و 3 را با هم XOR نمود:

$$r_1 = m_3 \oplus m_5 \oplus m_7 \Rightarrow r_1 = 1 \oplus 0 \oplus 1 = 0$$

$$r_2 = m_3 \oplus m_6 \oplus m_7 \Rightarrow r_2 = 1 \oplus 0 \oplus 1 = 0$$

$$r_4 = m_5 \oplus m_6 \oplus m_7 \Rightarrow r_4 = 0 \oplus 0 \oplus 1 = 1$$

در نتیجه کد همینگ برابر می‌شود با:

r_1	r_2	m_3	r_4	m_5	m_6	m_7
0	0	1	1	0	0	1

khalilifar.ir

کنترل خطا در کد همینگ

برای عمل کنترل خطا مقدار اندیس‌های داده که اندیس بیت افزونه در بدست آوردن شماره اندیس آنها نقش دارد با مقدار همان بیت افزونه XOR می‌شود، اگر نتیجه 0 باشد خطایی رخ نداده است:

$s_1 = r_1 \oplus m_3 \oplus m_5 \oplus m_7 \Rightarrow s_1 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$

$s_2 = r_2 \oplus m_3 \oplus m_6 \oplus m_7 \Rightarrow s_2 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$

$s_4 = r_4 \oplus m_5 \oplus m_6 \oplus m_7 \Rightarrow s_4 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$

در صورتی که در یک بیت، خطا رخ داده باشد محل وقوع خطا را می‌توان با معادل مقدار باینری $s_4s_2s_1$ بدست آورد. به عنوان مثال فرض کنید بیت 5 اُم خراب شود و از 0 به 1 دچار خطا شود، به صورت زیر:

r_1	r_2	m_3	r_4	m_5	m_6	m_7
0	0	1	1	1	0	1

$s_1 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$

$s_2 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$

$s_4 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$

$s_4s_2s_1 = (101)_2 = (5)_{10}$

در نتیجه بیت شماره 5 دچار خطا شده است و قابل تصحیح می‌باشد. و باید از 1 به 0 تصحیح شود، به صورت زیر:

r_1	r_2	m_3	r_4	m_5	m_6	m_7
0	0	1	1	0	0	1

اهداف سه‌گانه تکنیک RAID به صورت زیر است:

هدف اول: همانطور که گفتیم یکی از اهداف تکنیک RAID، **افزایش نرخ انتقال دیسک** (افزایش سرعت خواندن و نوشتن اطلاعات) است. در واقع زمانی که نیاز به ذخیره‌سازی اطلاعات بر روی دیسک است، می‌توان بخشی از اطلاعات را بر روی یک دیسک و بخشی دیگر را به طور همزمان و موازی بر روی دیسک دیگری ذخیره نمود تا سرعت دسترسی بالا رود. تکنیک Striping جهت تحقق این هدف مورد استفاده قرار می‌گیرد. در این تکنیک توسط موازی‌سازی و موازی‌کاری چند صف خواندن و نوشتن در چند دیسک مختلف، جایگزین یک صف خواندن و نوشتن در یک دیسک واحد می‌شود که به تبع آن افزایش سرعت دسترسی و افزایش نرخ انتقال را شاهد خواهیم بود.

khalilifar.ir

هدف دوم: یکی دیگر از اهداف تکنیک RAID، **افزایش قابلیت اطمینان (reliability)** است. برای این منظور اطلاعات یک دیسک می‌تواند به طور خودکار بر روی دیسک دیگری ذخیره شود تا در صورت خراب شدن دیسک اول، بتوان از دیسک دوم به عنوان پشتیبان اطلاعات استفاده نمود.

توجه: پشتیبان اطلاعات در کنار اطلاعات اصلی نوعی اطلاعات کنترلی و افزونگی (redundancy) محسوب می‌شود. در پشتیبان اطلاعات که به تبع افزونگی ایجاد می‌کند، اطلاعات اضافه‌تری ذخیره می‌شود که در حالت عادی و شرایط طبیعی لازم نمی‌شود و کاربردی هم ندارد، اما با وقوع یک حادثه یا خرابی یک دیسک (داده‌های اصلی)، می‌توان با استفاده از آنها (داده‌های افزونه)، اطلاعات اصلی از دست رفته را بازسازی کرد.

افزایش قابلیت اطمینان (reliability) به دو روش زیر انجام می‌شود:

□ **بدون صرفه‌جویی در فضای ذخیره‌سازی**

در این روش اطلاعات اصلی در دیسک اصلی عینا و با افزونگی 100 درصد در دیسک پشتیبان اطلاعات، تکرار و ذخیره‌سازی می‌شود. تکنیک Mirroring جهت تحقق این هدف مورد استفاده قرار می‌گیرد. تکنیک Mirroring (آینه‌سازی) ساده‌ترین و در عین حال گران‌ترین روش برای داشتن افزونگی و ساخت پشتیبان اطلاعات است. در آینه‌سازی، هر دیسک منطقی از دو دیسک فیزیکی تشکیل شده است. و هر عمل نوشتن، بر روی هر دو دیسک انجام می‌شود. اگر یکی از نسخه‌های دیسک به دلیل وقوع حادثه یا خرابی از دسترس خارج شود، داده‌ها را می‌توان از دیسک دیگر خواند. داده‌های اصلی فقط در صورتی به طور کامل از بین خواهند رفت که پس از خراب شدن دیسک اول، قبل از جایگزینی دیسک، دیسک دوم نیز خراب شود و از دسترس خارج شود.

□ **با صرفه‌جویی در فضای ذخیره‌سازی**

در این روش اطلاعات اصلی در دیسک اصلی با افزونگی در حد چند بیت توسط کدهای تصحیح خطا در دیسک پشتیبان اطلاعات، ذخیره‌سازی می‌شود. تکنیک Parity و تکنیک کد Hamming جهت تحقق این هدف مورد استفاده قرار می‌گیرد.

هدف سوم: یکی دیگر از اهداف تکنیک RAID، **افزایش فضای ذخیره‌سازی** به معنی گنجایشی بیش از یک دیسک بزرگ و گران است. برای این منظور می‌توان با بکارگیری چند دیسک در کنار یکدیگر، یک دیسک بزرگ مجازی از دیسک‌های کوچک‌تر ایجاد کرد.

توجه: تکنیک RAID در سطوح مختلفی برای تحقق اهداف فوق می‌تواند وجود داشته باشد که در هر سطح، بخشی یا تمام اهداف سه‌گانه فوق محقق می‌شود.

توجه: معمولا دیسک‌هایی که در RAID مورد استفاده قرار می‌گیرند، قابلیت تعویض فوری دارند، یعنی بدون قطع سیستم می‌توان آنها را تعویض کرد.

khalilifar.ir

توجه: با دیسک‌های چندگانه، یک درخواست خواندن و نوشتن می‌تواند به صورت موازی انجام شود، به شرطی که داده‌های مورد دسترسی، بر روی دیسک‌های مختلف توزیع و پخش شده باشد که منجر به «**افزایش نرخ انتقال خواندن و نوشتن**» در یک درخواست می‌شود. همچنین درخواست‌های خواندن و نوشتن جداگانه و مستقل هم می‌توانند تا جایی که داده‌های مورد نیاز بر روی دیسک‌های جداگانه قرار داشته باشند، به طور موازی و همزمان به اجرا درآیند که منجر به «**افزایش نرخ درخواست خواندن و نوشتن**» می‌شود.

توجه: به تفاوت «**نرخ درخواست خواندن و نوشتن**» و «**نرخ انتقال خواندن و نوشتن**» هم دقت نمایید. در نرخ درخواست خواندن و نوشتن تاکید روی توانایی اجرای چندین درخواست جداگانه و مستقل و موازی و همزمان باهم است. اما در نرخ انتقال خواندن و نوشتن تاکید روی میزان انتقال داده در یک درخواست واحد است.

توجه: به تعداد دسترسی‌های موازی جداگانه و مستقل به دیسک‌های مختلف در واحد زمان، نرخ درخواست خواندن و نوشتن گفته می‌شود.

توجه: به میزان داده منتقل شده برای فقط یک درخواست جاری در واحد زمان، نرخ انتقال خواندن و نوشتن گفته می‌شود.

نرخ درخواست خواندن و نوشتن

توجه: هرچه قدر اندازه Striping و تقسیم داده‌های اصلی **بزرگتر** (block-level Striping) باشد، نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **بیشتر** می‌شود، چون اندازه یک block در یک دیسک، **بزرگ** است، پس یک درخواست جاری، مدت زیادی را فقط در یک دیسک مشغول به کار خواهد بود، در نتیجه جهت خواندن و نوشتن اطلاعات بعدی با تاخیر سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های جداگانه و مستقل دیگر خواندن و نوشتن به طور موازی از سایر دیسک‌ها امکان‌پذیر خواهد بود، که منجر به **افزایش** نرخ درخواست خواندن و نوشتن در دیسک‌های مختلف می‌شود.

توجه: هرچه قدر اندازه Striping و تقسیم داده‌های اصلی **کوچکتر** (bit , byte-level Striping) باشد، نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **کمتر** می‌شود، چون اندازه یک bit , byte در یک دیسک، **کوچک** است، پس یک درخواست جاری، مدت کمی را فقط در یک دیسک مشغول به کار خواهد بود، در نتیجه جهت خواندن و نوشتن اطلاعات بعدی فورا سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های جداگانه و مستقل دیگر خواندن و نوشتن به طور موازی از سایر دیسک‌ها امکان‌پذیر نخواهد بود، چون همه دیسک‌ها به طور ترتیبی جهت خواندن و نوشتن، مشغول پاسخ به درخواست جاری هستن و امکان پاسخ به درخواست جدید تا پایان درخواست جاری وجود ندارد، که منجر به **کاهش** نرخ درخواست خواندن و نوشتن در دیسک‌های مختلف می‌شود.

khalilifar.ir

توجه: البته دقت نمایید که **نرخ درخواست نوشتن** در دیسک به دلیل محاسبات داده‌های افزونه در مقابل **نرخ درخواست خواندن** از دیسک بسته به میزان محاسبات و کیفیت سخت‌افزار می‌تواند **کمتر** باشد.

نرخ انتقال خواندن و نوشتن

توجه: هرچه قدر اندازه Striping و تقسیم داده‌های اصلی **بزرگتر** (block-level Striping) باشد، نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **کمتر** می‌شود، چون اندازه یک block در یک دیسک، **بزرگ** است، پس یک درخواست جاری، مدت زیادی را فقط در یک دیسک مشغول به کار خواهد بود، در نتیجه جهت خواندن و نوشتن اطلاعات بعدی با تاخیر سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های دیگر خواندن و نوشتن به طور موازی از سایر دیسک‌ها امکان‌پذیر خواهد بود، که منجر به **کاهش** نرخ انتقال خواندن و نوشتن درخواست جاری می‌شود. به عبارت دیگر در این حالت از قدرت نرخ انتقال و چند برابری تمام دیسک‌ها باهم برای پاسخ به یک درخواست جاری استفاده نمی‌شود.

توجه: هرچه قدر اندازه Striping و تقسیم داده‌های اصلی **کوچکتر** (bit , byte-level Striping) باشد، نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **بیشتر** می‌شود، چون اندازه یک bit , byte در یک دیسک، **کوچک** است، پس یک درخواست جاری، مدت کمی را فقط در یک دیسک مشغول به کار خواهد بود، در نتیجه جهت خواندن و نوشتن اطلاعات بعدی فورا سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های دیگر خواندن و نوشتن به طور موازی از سایر دیسک‌ها امکان‌پذیر نخواهد بود، چون همه دیسک‌ها به طور ترتیبی جهت خواندن و نوشتن، مشغول پاسخ به درخواست جاری هستن و امکان پاسخ به درخواست جدید تا پایان درخواست جاری وجود ندارد، که منجر به **افزایش** نرخ انتقال خواندن و نوشتن درخواست جاری می‌شود. به عبارت دیگر در این حالت از قدرت نرخ انتقال و چند برابری تمام دیسک‌ها باهم برای پاسخ به یک درخواست جاری استفاده می‌شود.

توجه: البته دقت نمایید که **نرخ انتقال نوشتن** در دیسک به دلیل محاسبات داده‌های افزونه در مقابل **نرخ انتقال خواندن** از دیسک بسته به میزان محاسبات و کیفیت سخت‌افزار می‌تواند **کمتر** باشد.

توجه: در محیط‌های تعامل‌گرا، در اغلب موارد، کاربر نرخ درخواست خواندن و نوشتن بالا را به نرخ انتقال خواندن و نوشتن بالا ترجیح می‌دهد. کارآمدی افزایش نرخ درخواست خواندن و نوشتن تحت تاثیر اندازه Striping یا باریکه‌سازی و قطعه‌سازی است. اگر اندازه باریکه بزرگ باشد، به طوری که یک درخواست خواندن و نوشتن فقط شامل یک دسترسی به دیسک باشد، درخواست‌های خواندن و نوشتن بعدی می‌توانند به صورت موازی و همزمان انجام گیرند و به این ترتیب زمان صف‌بندی هر درخواست را کاهش دهد.

khalilifar.ir

RAID سطح 0

در RAID سطح 0 ، تکنیک Block-level striping without parity or mirroring مورد استفاده قرار می‌گیرد، در نتیجه **افزایش کارایی خواندن** (read performance: n) و **افزایش کارایی نوشتن** (write performance: n) برای «**داده‌های اصلی**» محقق می‌شود. یعنی افزایش سرعت خواندن و نوشتن بالایی دارد. البته در صورتی که اندازه بلوک داده کوچک در نظر گرفته شود.

توجه: در این سطح، داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود. مزیت این سطح این است که توان عملیات خواندن و نوشتن در تعداد دیسک‌ها ضرب می‌شود، زیرا خواندن و نوشتن به طور همزمان و موازی انجام می‌شود.

همچنین هیچکدام از تکنیک‌های افزونگی (redundancy) مثل Mirroring، Hamming و Parity مورد استفاده قرار نمی‌گیرد، در نتیجه **افزایش قابلیت اطمینان** محقق نمی‌شود.

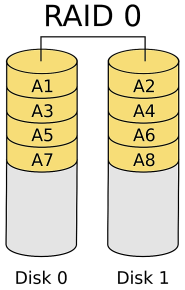
توجه: در این سطح، داده‌های افزونه (redundancy) روی دیسک‌های مختلف توزیع (distributed) نمی‌شود. چون اصلا برای این سطح داده افزونه تعریف نشده و وجود ندارد.

توجه: اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد، چون اصلا برای این سطح داده افزونه تعریف نشده و وجود ندارد. به عبارت دیگر تحمل‌پذیری در برابر خطا را ندارد (fault tolerance: None). از آنجا که striping محتوای داده‌های اصلی را بین همه دیسک‌های موجود توزیع می کند، و هیچ تکنیک افزونه و پشتیبان اطلاعات هم وجود ندارد، بنابراین خرابی هر دیسک باعث می‌شود که کل حجم RAID 0 و همه داده‌های اصلی از بین بروند.

توجه: این سطح، کمترین میزان افزونگی را ایجاد می‌کند، در نتیجه کمترین میزان قابلیت اطمینان را نیز ایجاد می‌کند. 100 درصد کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی و 0 درصد دیگر فضای ذخیره‌سازی برای نگهداری داده‌های افزونه (پشتیبان اطلاعات) مورد استفاده قرار می‌گیرد. یعنی اگر فضای ذخیره‌سازی دیسک اول برابر 50GB و فضای ذخیره‌سازی دیسک دوم هم 50GB باشد، آنگاه کل 100GB فضای ذخیره‌سازی برای ذخیره داده‌های اصلی مورد استفاده قرار می‌گیرد. (space efficiency:1= 100%).

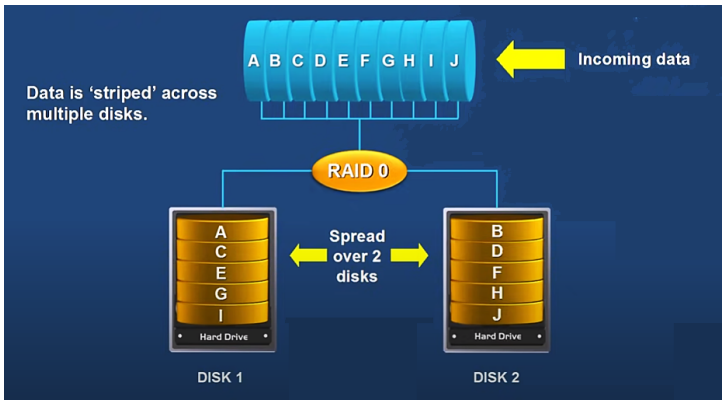
توجه: این سطح، هیچ خرابی از دیسک‌ها را پشتیبانی نمی‌کند. (fault tolerance: None)

توجه: این سطح، در حداقل 2 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 2)

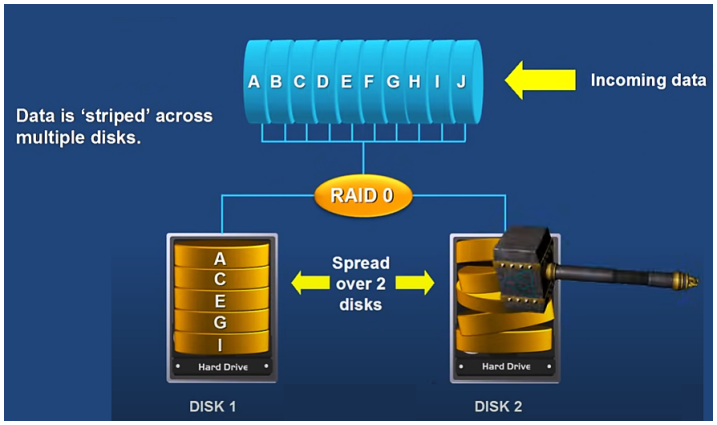


khalilifar.ir

شکل زیر گویای عملکرد RAID سطح 0 است:



اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد، چون اصلا برای این سطح داده افزونه تعریف نشده و وجود ندارد. شکل زیر گویای مطلب است:



پُرواضح است که داده‌های دیسک دوم پس از خرابی، قابل بازیابی نیست.

توجه: در RAID 0، تکنیک blok-level Striping مورد استفاده قرار می‌گیرد (**البته اگر در RAID 0 اندازه بلوک داده، کوچک در نظر گرفته شود**) و Strping و تقسیم داده‌های اصلی کوچک و در سطح بلوک کوچک باشد، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **کم** است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **زیاد** است.

توجه: در RAID 0، تکنیک blok-level Striping مورد استفاده قرار می‌گیرد (**البته اگر در RAID 0 اندازه بلوک داده، بزرگ در نظر گرفته شود**) و Strping و تقسیم داده‌های اصلی بزرگ و در سطح بلوک بزرگ باشد، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف

khalilifar.ir

جداگانه و مستقل موازی **زیاد** است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **کم** است.

توجه: در حالت درخواست خواندن و نوشتن با بلوک بزرگ، اگر دو درخواست برای دو بلوک مختلف از داده‌ها وجود داشته باشد. احتمال زیادی وجود دارد که بلوک‌های درخواست شده روی دو دیسک متفاوت قرار داشته باشد. در نتیجه دو درخواست می‌توانند به طور موازی دنبال شوند و زمان صف‌بندی کاهش یابد.

RAID سطح 1

در RAID سطح 1 ، تکنیک Mirroring without parity or striping مورد استفاده قرار می‌گیرد، در نتیجه **افزایش کارایی خواندن** (read performance: 1) و **افزایش کارایی نوشتن** (write performance: 1) برای «**داده‌های اصلی**» محقق نمی‌شود. در تئوری وجود n دیسک می‌تواند منجر به خواندن n برابری بشود ولی در اینجا و در عمل و بسته به شرایط تعریف شده و عدم استفاده از تکنیک Striping در RAID 1 مقدار واقعی read performance برابر همان مقدار 1 است، اما در تئوری حداکثر مقدار آن برابر n می‌تواند باشد. درخواست خواندن می‌تواند توسط هر یک از دو دیسکی که بیشترین سرعت را دارد صورت گیرد. همچنین اگر دیسک‌هایی با سرعت‌های مختلف در آرایه RAID 1 استفاده شود، عملکرد کلی نوشتن برابر با سرعت کندترین دیسک است، برای درخواست نوشتن هر دو دیسک باید بهنگام شوند. در RAID 1 هیچگونه «**جریمه نوشتن**» **به معنی محاسبات داده‌های افزونه** وجود ندارد. اما سطوح 2 تا 6 شامل استفاده از بیت‌های توازن است و به تبع «**جریمه نوشتن**» به معنی محاسبات داده‌های افزونه را دارند. در نتیجه، در بهنگام‌کردن و نوشتن روی دیسک‌ها، نرم‌افزار مدیریت آرایه، باید ابتدا بیت‌های توازن را محاسبه و بهنگام کند.

توجه: در این سطح، داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) نمی‌شود.

همچنین تکنیک Mirroring مورد استفاده قرار می‌گیرد، در نتیجه **افزایش قابلیت اطمینان** محقق می‌شود.

توجه: در این سطح، داده‌های افزونه (Mirroring) روی دیسک‌های مختلف توزیع (distributed) نمی‌شود. بلکه در مکان‌های مشخص روی دیسک‌ها ذخیره می‌شوند.

توجه: اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط فقط داده‌های افزونه (Mirroring) وجود دارد. (fault tolerance: mirrored set).

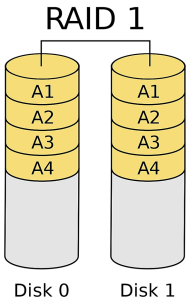
توجه: این سطح، بالاترین میزان افزونگی را ایجاد می‌کند، در نتیجه بالاترین میزان قابلیت اطمینان را نیز ایجاد می‌کند. 50 درصد از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی و 50 درصد دیگر فضای ذخیره‌سازی برای نگهداری داده‌های افزونه (پشتیبان اطلاعات) مورد استفاده قرار می‌گیرد. و این یعنی ایجاد 100 درصد افزونگی از روی داده‌های اصلی. یعنی اگر فضای ذخیره‌سازی دیسک اول برابر 50 GB و فضای ذخیره‌سازی دیسک دوم هم 50 GB باشد، آنگاه

khalilifar.ir

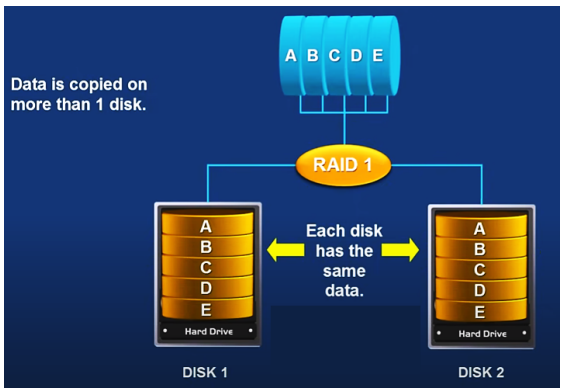
فقط 50 GB فضای ذخیره‌سازی یکی از دیسک‌ها برای ذخیره داده‌های اصلی مورد استفاده قرار می‌گیرد. (space efficiency: $\frac{1}{n} = 50\%$)

توجه: این سطح، خرابی (n-1) دیسک را پشتیبانی می‌کند. (fault tolerance: n-1 drive failures)

توجه: این سطح، در حداقل 2 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 2)



شکل زیر گویای عملکرد RAID سطح 1 است:



اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود دارد، شکل زیر گویای مطلب است:



khalilifar.ir

پُرواضح است که داده‌های دیسک دوم پس از خرابی، از دیسک اول به طور کامل و صحیح قابل بازیابی است.

توجه: در RAID 1، تکنیک Striping مورد استفاده قرار نمی‌گیرد.

RAID سطح 2

در RAID سطح 2 ، تکنیک Bit-level striping with Hamming code for error correction مورد استفاده قرار می‌گیرد، در نتیجه **افزایش کارایی خواندن** (read performance: Depends) و **افزایش کارایی نوشتن** (write performance: Depends) برای «**داده‌های اصلی**» تا حدی محقق می‌شود.

توجه: در این سطح، داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود.

همچنین تکنیک Hamming مورد استفاده قرار می‌گیرد، در نتیجه **افزایش قابلیت اطمینان** محقق می‌شود.

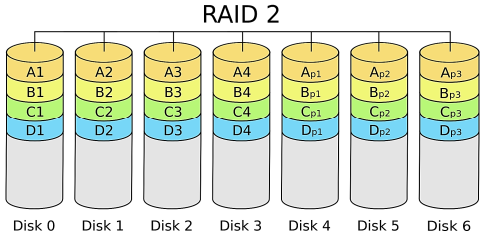
توجه: در این سطح، داده‌های افزونه (کد Hamming) روی دیسک‌های مختلف توزیع (distributed) نمی‌شود، بلکه در مکان‌های مشخص روی دیسک‌ها ذخیره می‌شوند. کد همینگ خطاهای تک بیتی را می‌تواند کشف و تصحیح نماید و خطاهای دوبیتی را فقط کشف نماید.

توجه: در کد Hamming یک دیسک خراب معادل یک بیت خراب است، در نتیجه اگر یکی از دیسک‌ها خراب شود و نیاز به تعویض با یک دیسک جدید داشته باشد، امکان بازیابی اطلاعات اصلی توسط داده‌های اصلی و افزونه (کد Hamming) وجود دارد. به عبارت دیگر با خراب شدن یک دیسک، دیسک‌های کد Hamming دسترسی شده و داده‌های از دست رفته از بقیه دیسک‌ها، بازسازی می‌گردد.

توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Hamming درصدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و درصدی دیگر از فضای ذخیره‌سازی برای نگهداری داده‌های افزونه (کد Hamming) یا پشتیبان اطلاعات، مورد استفاده قرار می‌گیرد. (space efficiency: $1-\frac{1}{n}\log_2(n+1)$)

توجه: این سطح خرابی یک دیسک را پشتیبانی می‌کند. (fault tolerance: one drive failure)

توجه: این سطح در حداقل 3 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 3)



khalilifar.ir

توجه: در افزونگی RAID 2 و استفاده از کد Hamming زیاده‌روی شده است. در RAID 2 تعداد دیسک‌های افزونه متناسب با لگاریتم تعداد دیسک‌های داده‌های اصلی است.
توجه: سطح RAID 2 به طور عملی پیاده‌سازی نشده است و مورد استفاده قرار نمی‌گیرد.
توجه: در RAID 2، تکنیک bit-level Striping مورد استفاده قرار می‌گیرد و Strping و تقسیم داده‌های اصلی کوچک و در سطح بیت است، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **کم** است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **زیاد** است.
توجه: در RAID 2 در یک خواندن واحد، همزمان تمام دیسک‌ها مورد دسترسی قرار می‌گیرند. داده‌های درخواست‌شده و کد تصحیح خطای مربوطه به کنترل‌کننده آرایه فرستاده می‌شود. اگر یک خطای تک بیتی موجود باشد، کنترل‌کننده می‌تواند بی‌درنگ آنرا تشخیص داده و تصحیح نماید، در نتیجه زمان دسترسی برای خواندن کند نمی‌شود. برای یک نوشتن واحد نیز، باید تمام دیسک‌های داده اصلی و دیسک‌های داده افزونه مورد دسترسی قرار گیرند.

RAID سطح 3

در RAID سطح 3 ، تکنیک Byte-level striping with dedicated parity مورد استفاده قرار می‌گیرد، در نتیجه **افزایش کارایی خواندن** (read performance: n-1) و **افزایش کارایی نوشتن** (write performance: n-1) برای «**داده‌های اصلی**» تا حدی محقق می‌شود. البته در بخش کارایی نوشتن، کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشتن) وجود دارد، بنابراین سخت‌افزاری را فرض کنید که می‌تواند محاسبات مرتبط با بیت توازن را با سرعت کافی انجام دهد.
RAID 3 نیز شبیه RAID 2 عمل می‌کند، ولی از افزونگی کمتری استفاده می‌کند و فقط از یک تک بیت افزونه (parity) به عبارت دیگر یک تک بیت **توازن منفرد** برای بازیابی اطلاعات متناظر استفاده می‌کند. **یعنی فقط از یک دیسک اضافی برای ذخیره‌سازی تک بیت افزونه بهره می‌گیرد** که نسبت به RAID 2 ارزان‌تر است. درحالی‌که در RAID 2 تعداد دیسک‌های افزونه به دلیل استفاده از کد Hamming، متناسب با لگاریتم تعداد دیسک‌های داده‌های اصلی بود. بنابراین در کاربرد و عمل RAID 2 مورد استفاده قرار نمی‌گیرد.
توجه: البته دقت کنید که تک بیت‌های افزونه در یک بایت شامل هشت بیت افزونه، نتیجه تناظر و توازن بیت به بیت در بایت‌های داده‌های اصلی است. برای مثال یک بایت داده افزونه، شامل هشت تک بیت افزونه، نتیجه تناظر و توازن بیت به بیت در بایت داده‌های اصلی یعنی بایت اول دیسک اول، بایت اول دیسک دوم و بایت اول دیسک n اُم است.
توجه: در این سطح، داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود.
همچنین تکنیک Dedicated Parity به معنی Parity اختصاصی مورد استفاده قرار می‌گیرد، در نتیجه **افزایش قابلیت اطمینان** محقق می‌شود.

khalilifar.ir

توجه: در این سطح، داده افزونه (Parity) روی دیسک‌های مختلف توزیع (distributed) نمی‌شود. بلکه در یک مکان مشخص روی یک دیسک مشخص ذخیره می‌شود. این کد، خطاهای تک بیتی را می‌تواند کشف و تصحیح نماید.

توجه: در این کد، یک دیسک خراب از داده‌های اصلی معادل یک بایت خراب است و یک دیسک خراب از داده افزونه (مجموع تک بیت‌های افزونه) معادل یک بایت خراب است، در نتیجه اگر یکی از دیسک‌ها خراب شود و نیاز به تعویض با یک دیسک جدید داشته باشد، امکان بازیابی اطلاعات اصلی توسط داده‌های اصلی و افزونه (Dedicated Parity) وجود دارد. به عبارت دیگر با خراب شدن یک دیسک، دیسک توازن دسترسی شده و داده‌های از دست رفته از بقیه دیسک‌ها، بازسازی می‌گردد.

توجه: بازسازی داده‌ها ساده است. آرایه‌ای از چهار دیسک را در نظر بگیرید که در آن X0 تا X2 حاوی داده‌های اصلی و X3 دیسک توازن است. توازن بیت i اُم به روش زیر محاسبه می‌گردد:

$$X3(i) = X2(i) \oplus X1(i) \oplus X0(i)$$

توجه: عملگر XOR با نماد $\oplus$ نشانه استفاده از توازن زوج است.

فرض کنید دیسک X1 خراب شود، در این شرایط هر بیت از داده‌های اصلی بر روی دیسک X1 می‌تواند از محتویات بیت‌های متناظر و هم ردیف خودش از دیسک‌های باقی‌مانده داده اصلی و داده افزونه در آرایه، به روش زیر بازسازی گردد:

$$X1(i) = X3(i) \oplus X2(i) \oplus X0(i)$$

روابط فوق برای سطوح RAID 3 تا RAID 6 برقرار است.

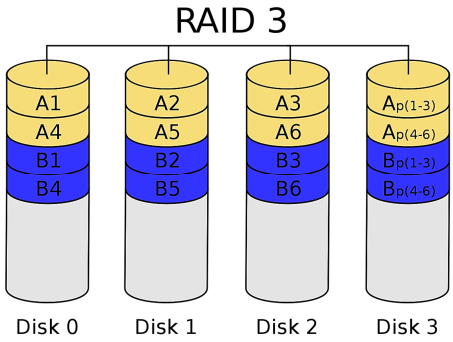
توجه: در هنگام خرابی یک دیسک، تمامی داده‌ها، در حالتی که «**حالت کاهشی**» نامیده می‌شود، موجودند. در این حالت داده‌های از دست رفته با استفاده از محاسبه XOR بازسازی می‌گردند.

توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Parity درصدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و درصدی دیگر از فضای ذخیره‌سازی معادل یک تک دیسک برای نگهداری داده‌های افزونه (parity) یا پشتیبان اطلاعات شامل مجموع تک بیت‌های افزونه(معادل یک بایت)، مورد استفاده قرار می‌گیرد. (space efficiency: $1-\frac{1}{n}$).

توجه: این سطح، خرابی یک دیسک را پشتیبانی می‌کند. (fault tolerance: one drive failure)

توجه: این سطح، در حداقل 3 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 3)

khalilifar.ir



همانطور که در تصویر فوق واضح است، n-1 دیسک برای ذخیره‌سازی داده‌های اصلی و 1 دیسک برای ذخیره‌سازی تک بیت داده افزونه (بایت افزونه) به عنوان پشتیبان اطلاعات مورد استفاده قرار می‌گیرد.

توجه: در RAID 3، تکنیک byte-level Striping مورد استفاده قرار می‌گیرد و Strping و تقسیم داده‌های اصلی کوچک و در سطح بایت است، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **کم** است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **زیاد** است.

RAID سطح 4

در RAID سطح 4، تکنیک Block-level striping with dedicated parity مورد استفاده قرار می‌گیرد، در نتیجه **افزایش کارایی خواندن** (read performance: n-1) و **افزایش کارایی نوشتن** (write performance: n-1) برای «**داده‌های اصلی**» تا حدی محقق می‌شود. البته در بخش کارایی نوشتن، کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشتن) وجود دارد، بنابراین سخت‌افزاری را فرض کنید که می‌تواند محاسبات مرتبط با بیت توازن را با سرعت کافی انجام دهد.

RAID 4 نیز شبیه RAID 3 عمل می‌کند، و فقط از یک تک بیت افزونه (parity) به عبارت دیگر یک تک بیت **توازن منفرد** برای بازیابی اطلاعات متناظر استفاده می‌کند. **یعنی فقط از یک دیسک اضافی برای ذخیره‌سازی تک بیت افزونه بهره می‌گیرد** که نسبت به RAID 2 ارزان‌تر است.

توجه: البته دقت کنید که تک بیت‌های افزونه در یک بلوک افزونه، نتیجه تناظر و توازن بیت به بیت در بلوک‌های داده‌های اصلی است. برای مثال یک بلوک داده افزونه، نتیجه تناظر و توازن بیت به بیت در بلوک‌های داده‌های اصلی یعنی بلوک اول دیسک اول، بلوک اول دیسک دوم و بلوک اول دیسک n اُم است.

توجه: در این سطح داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود.

khalilifar.ir

همچنین تکنیک Dedicated Parity به معنی Parity اختصاصی مورد استفاده قرار می‌گیرد، در نتیجه **افزایش قابلیت اطمینان** محقق می‌شود.

توجه: در این سطح، داده افزونه (Parity) روی دیسک‌های مختلف توزیع (distributed) نمی‌شود. بلکه در یک مکان مشخص روی یک دیسک مشخص ذخیره می‌شود. این کد، خطاهای تک بیتی را می‌تواند کشف و تصحیح نماید.

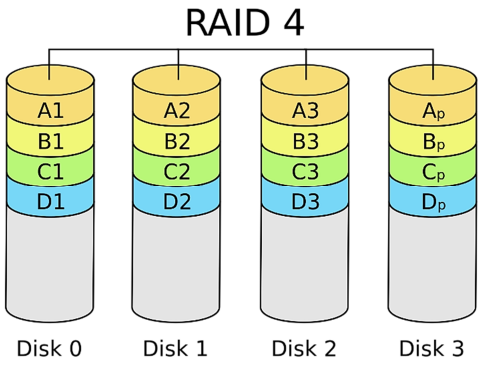
توجه: در این کد، یک دیسک خراب از داده‌های اصلی معادل یک بلوک خراب است و یک دیسک خراب از داده افزونه (مجموع تک بیت‌های افزونه) معادل یک بلوک خراب است، در نتیجه اگر یکی از دیسک‌ها خراب شود و نیاز به تعویض با یک دیسک جدید داشته باشد، امکان بازیابی اطلاعات اصلی توسط داده‌های اصلی و افزونه (Dedicated Parity) وجود دارد. به عبارت دیگر با خراب شدن یک دیسک، دیسک توازن دسترسی شده و داده‌های از دست رفته از بقیه دیسک‌ها، بازسازی می‌گردد.

توجه: بازسازی داده‌ها ساده است. و همانند RAID 3 با استفاده از عملگر XOR انجام می‌گردد.

توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Parity درصدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و درصدی دیگر از فضای ذخیره‌سازی معادل یک تک دیسک برای نگهداری داده‌های افزونه (parity) یا پشتیبان اطلاعات شامل مجموع تک بیت‌های افزونه(معادل یک بلوک)، مورد استفاده قرار می‌گیرد. ($1-\frac{1}{n}$:space efficiency).

توجه: این سطح خرابی یک دیسک را پشتیبانی می‌کند. (fault tolerance: one drive failure)

توجه: این سطح در حداقل 3 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 3)



همانطور که در تصویر فوق واضح است، n-1 دیسک برای ذخیره‌سازی داده‌های اصلی و 1 دیسک برای ذخیره‌سازی تک بیت داده افزونه (بلوک افزونه) به عنوان پشتیبان اطلاعات مورد استفاده قرار می‌گیرد.

khalilifar.ir

توجه: سطح RAID 4 به طور عملی پیاده‌سازی نشده است و مورد استفاده قرار نمی‌گیرد.
توجه: در RAID 4، تکنیک block-level Striping مورد استفاده قرار می‌گیرد و Strping و تقسیم داده‌های اصلی بزرگ و در سطح بلوک است، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **زیاد** است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **متوسط** است.
توجه: البته دقت نمایید که **نرخ انتقال نوشتن** در دیسک به دلیل محاسبات داده‌های افزونه در مقابل **نرخ انتقال خواندن** از دیسک بسته به میزان محاسبات و کیفیت سخت‌افزار می‌تواند **کمتر** باشد.
توجه: البته دقت نمایید که **نرخ درخواست نوشتن** در دیسک به دلیل محاسبات داده‌های افزونه در مقابل **نرخ درخواست خواندن** از دیسک بسته به میزان محاسبات و کیفیت سخت‌افزار می‌تواند **کمتر** باشد.
توجه: از RAID 4 تا RAID 6 به دلیل استفاده از تکنیک block-level Striping و به تبع استفاده از باریکه‌های بزرگ یک روش دسترسی مستقل ایجاد می‌شود. یک بلوک خواندن و نوشتن، فقط به یک دیسک دستیابی دارد و به درخواست‌های دیگر امکان می‌دهد تا به وسیله دیسک‌های دیگر پردازش شوند. نرخ انتقال خواندن و نوشتن در هر دستیابی پایین است چون از قدرت نرخ انتقال همه دیسک‌ها به طور موازی استفاده نمی‌شود، اما چندین دستیابی خواندن و نوشتن می‌تواند به صورت موازی صورت بگیرد که منجر به افزایش نرخ درخواست خواندن و نوشتن می‌شود. در آرایه‌ای با دسترسی مستقل، هر دیسک عضو به طور مستقل عمل می‌کند. پس می‌توان درخواست‌های خواندن و نوشتن مجزا را به صورت موازی برآورده کرد. به همین دلیل، آرایه‌ای با دسترسی مستقل، بیشتر برای کاربردهای با نرخ بالای درخواست خواندن و نوشتن (**محیط‌های تعامل‌گرا**) مناسبند و برای کاربردهای با نرخ بالای انتقال داده‌ها مناسب نیستند.
توجه: دقت کنید که اگر حجم داده اصلی برای خواندن و نوشتن در یک درخواست واحد بیشتر از یک بلوک باشد، آنگاه نرخ انتقال خواندن و نوشتن هم افزایش پیدا می‌کند و بالا خواهد بود، چون از قدرت نرخ انتقال همه دیسک‌ها به طور موازی استفاده می‌شود.
توجه: در تکنیک RAID 4 اگر تک دیسک داده‌های افزونه خراب شود کل سیستم بازیابی داده‌ها مختل می‌شود. اما در RAID 5 داده‌های افزونه (Parity) روی دیسک‌های مختلف، توزیع (distributed) می‌شود. یعنی در یک مکان مشخص روی یک دیسک مشخص ذخیره نمی‌شود، بلکه در دیسک‌های مختلف ذخیره می‌شوند، بنابراین از وقوع «**مختل شدن کل سیستم بازیابی داده‌ها**» جلوگیری می‌کند.
توجه: در تکنیک RAID 4 یک ستون از آرایه دیسک‌ها مربوط به داده‌های افزونه (بیت توازن) است. بنابراین کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشتن) وجود دارد، از آنجاکه هر عمل نوشتن باید با دیسک توازن درگیر باشد، پس سرعت سخت‌افزاری که محاسبات مرتبط با

khalilifar.ir

بیت توازن را انجام می‌دهد می‌تواند «**گلوگاه**» ایجاد کند. در RAID 5 داده‌های افزونه (Parity) روی دیسک‌های مختلف، توزیع (distributed) می‌شود. یعنی در یک مکان مشخص روی یک دیسک مشخص ذخیره نمی‌شود، بلکه در دیسک‌های مختلف ذخیره می‌شوند، بنابراین از وقوع «**گلوگاه**» جلوگیری می‌کند.

RAID سطح 5

در RAID سطح 5 ، تکنیک Block-level striping with distributed parity مورد استفاده قرار می‌گیرد، در نتیجه **افزایش کارایی خواندن** (read performance: n) و **افزایش کارایی نوشتن** (write performance: n-1) در یک full stripe یعنی نوار کامل برای «**داده‌های اصلی**» تا حدی محقق می‌شود. البته در بخش کارایی نوشتن، کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشتن) وجود دارد، بنابراین سخت‌افزاری را فرض کنید که می‌تواند محاسبات مرتبط با بیت توازن را با سرعت کافی انجام دهد.

RAID 5 نیز شبیه RAID 4 عمل می‌کند، و فقط از یک تک بیت افزونه (parity) به عبارت دیگر یک تک بیت **توازن منفرد** برای بازیابی اطلاعات متناظر استفاده می‌کند. **یعنی فقط از یک دیسک اضافی برای ذخیره‌سازی تک بیت افزونه بهره می‌گیرد**، اما به صورت **توزیع شده** است.

توجه: البته دقت کنید که تک بیت‌های افزونه در یک بلوک افزونه، نتیجه تناظر و توازن بیت به بیت در بلوک‌های داده‌های اصلی است. برای مثال یک بلوک داده افزونه، نتیجه تناظر و توازن بیت به بیت در بلوک‌های داده‌های اصلی یعنی بلوک اول دیسک اول، بلوک اول دیسک دوم و بلوک اول دیسک n اُم است.

توجه: در این سطح داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود.

همچنین تکنیک **Distributed Parity** به معنی **Parity توزیع شده** مورد استفاده قرار می‌گیرد، در نتیجه **افزایش قابلیت اطمینان** محقق می‌شود.

توجه: در این سطح داده‌های افزونه (Parity) روی دیسک‌های مختلف، توزیع (distributed) می‌شود. یعنی در یک مکان مشخص روی یک دیسک مشخص ذخیره نمی‌شود، بلکه در دیسک‌های مختلف ذخیره می‌شوند. این کد، خطاهای تک بیتی را می‌تواند کشف و تصحیح نماید.

توجه: در این کد، یک دیسک خراب از داده‌های اصلی معادل یک بلوک خراب است و یک دیسک خراب از داده افزونه (مجموع تک بیت‌های افزونه) معادل یک بلوک خراب است، در نتیجه اگر یکی از دیسک‌ها خراب شود و نیاز به تعویض با یک دیسک جدید داشته باشد، امکان بازیابی اطلاعات اصلی توسط داده‌های اصلی و افزونه (Distributed Parity) وجود دارد. به عبارت دیگر با خراب شدن یک دیسک، دیسک توازن دسترسی شده و داده‌های از دست رفته از بقیه دیسک‌ها، بازسازی می‌گردد.

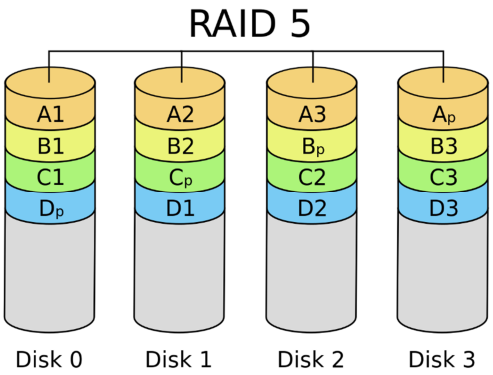
توجه: بازسازی داده‌ها ساده است. و همانند RAID 4 با استفاده از عملگر XOR انجام می‌گردد.

khalilifar.ir

توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Parity درصدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و درصدی دیگر از فضای ذخیره‌سازی معادل یک تک دیسک برای نگهداری داده‌های افزونه (parity) یا پشتیبان اطلاعات شامل مجموع تک بیت‌های افزونه(معادل یک بلوک)، مورد استفاده قرار می‌گیرد. ($1-\frac{1}{n}$:space efficiency).

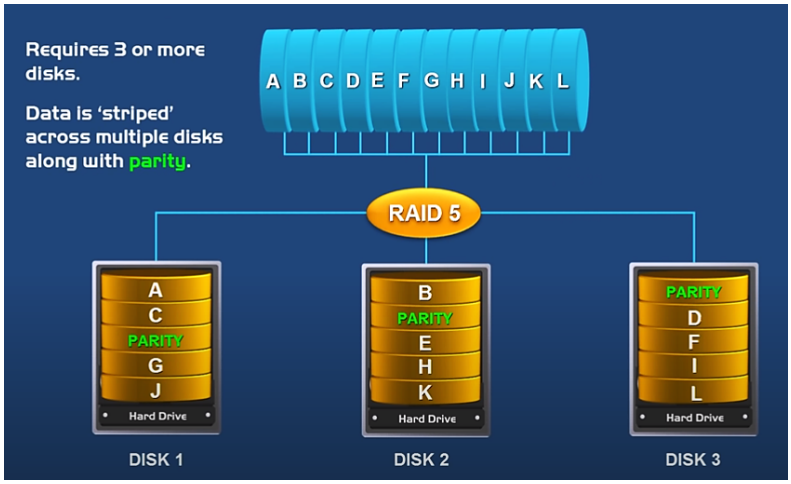
توجه: این سطح خرابی یک دیسک را پشتیبانی می‌کند. (fault tolerance: one drive failure)

توجه: این سطح در حداقل 3 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 3)



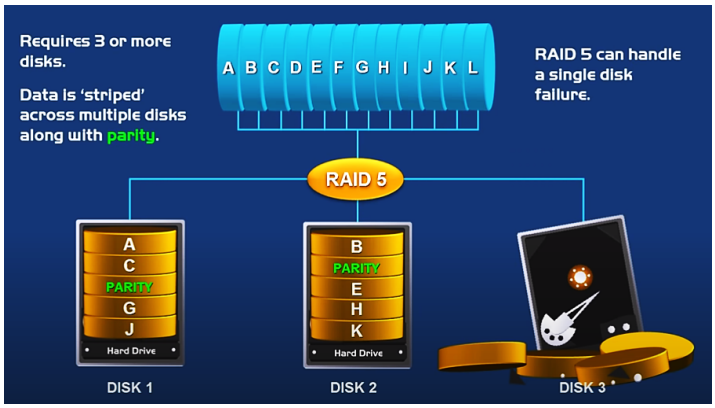
همانطور که در تصویر فوق واضح است، n-1 دیسک برای ذخیره‌سازی داده‌های اصلی و 1 دیسک برای ذخیره‌سازی تک بیت داده افزونه (بلوک افزونه) به عنوان پشتیبان اطلاعات مورد استفاده قرار می‌گیرد.

شکل زیر گویای عملکرد RAID سطح 5 است:

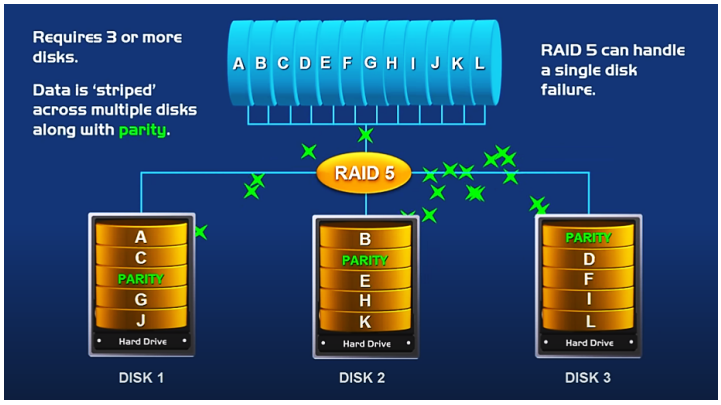


khalilifar.ir

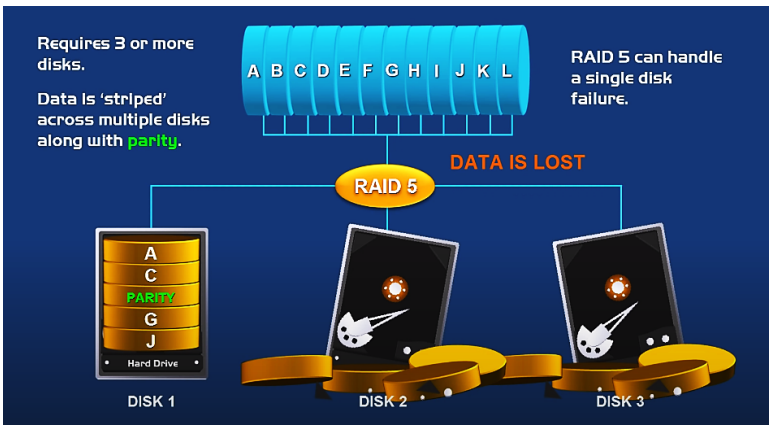
اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود دارد:



شکل زیر گویای مطلب است:

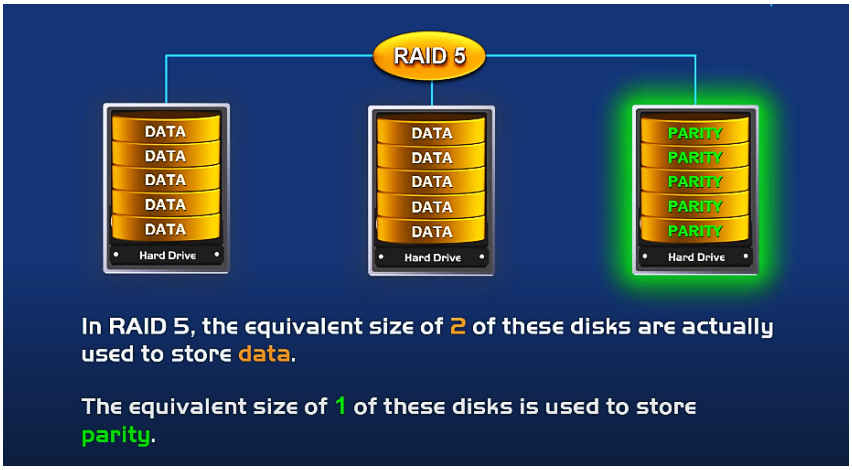


اگر دو عدد از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد، شکل زیر گویای مطلب است:



khalilifar.ir

این سطح افزونگی ایجاد می‌کند، برای مثال از کل 3 دیسک، 2 دیسک شامل داده اصلی و 1 دیسک برای داده افزونه مورد استفاده قرار می‌گیرد، شکل زیر گویای مطلب است:



توجه: در RAID 5، تکنیک block-level Striping مورد استفاده قرار می‌گیرد و Strping و تقسیم داده‌های اصلی بزرگ و در سطح بلوک است، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **زیاد** است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **متوسط** است.

توجه: البته دقت نمایید که **نرخ انتقال نوشتن** در دیسک به دلیل محاسبات داده‌های افزونه در مقابل **نرخ انتقال خواندن** از دیسک بسته به میزان محاسبات و کیفیت سخت‌افزار می‌تواند **کمتر** باشد.

توجه: البته دقت نمایید که **نرخ درخواست نوشتن** در دیسک به دلیل محاسبات داده‌های افزونه در مقابل **نرخ درخواست خواندن** از دیسک بسته به میزان محاسبات و کیفیت سخت‌افزار می‌تواند **کمتر** باشد.

توجه: دقت کنید که اگر حجم داده اصلی برای خواندن و نوشتن در یک درخواست واحد بیشتر از یک بلوک باشد، آنگاه نرخ انتقال خواندن و نوشتن هم افزایش پیدا می‌کند و بالا خواهد بود، چون از قدرت نرخ انتقال همه دیسک‌ها به طور موازی استفاده می‌شود.

توجه: در تکنیک RAID 4 اگر تک دیسک داده‌های افزونه خراب شود کل سیستم بازیابی داده‌ها مختل می‌شود. اما در RAID 5 داده‌های افزونه (Parity) روی دیسک‌های مختلف، توزیع (distributed) می‌شود. یعنی در یک مکان مشخص روی یک دیسک مشخص ذخیره نمی‌شود، بلکه در دیسک‌های مختلف ذخیره می‌شوند، بنابراین از وقوع «**مختل شدن کل سیستم بازیابی داده‌ها**» جلوگیری می‌کند.

khalilifar.ir

توجه: در تکنیک RAID 4 یک ستون از آرایه دیسک‌ها مربوط به داده‌های افزونه (بیت توازن) است. بنابراین کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشتن) وجود دارد، از آنجاکه هر عمل نوشتن باید با دیسک توازن درگیر باشد، پس سرعت سخت‌افزاری که محاسبات مرتبط با بیت توازن را انجام می‌دهد می‌تواند «**گلوگاه**» ایجاد کند. در RAID 5 داده‌های افزونه (Parity) روی دیسک‌های مختلف، توزیع (distributed) می‌شود. یعنی در یک مکان مشخص روی یک دیسک مشخص ذخیره نمی‌شود، بلکه در دیسک‌های مختلف ذخیره می‌شوند، بنابراین از وقوع «**گلوگاه**» جلوگیری می‌کند.

RAID سطح 6

در RAID سطح 6 ، تکنیک Block-level striping with double distributed parity مورد استفاده قرار می‌گیرد، در نتیجه **افزایش کارایی خواندن** (read performance: n) و **افزایش کارایی نوشتن** (write performance: n-2) در یک full stripe یعنی نوار کامل برای «**داده‌های اصلی**» تا حدی محقق می‌شود. البته در بخش کارایی نوشتن، کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشتن) وجود دارد، بنابراین سخت‌افزاری را فرض کنید که می‌تواند محاسبات مرتبط با بیت توازن را با سرعت کافی انجام دهد.

RAID 6 نیز شبیه RAID 5 عمل می‌کند، اما با این تفاوت که از دو تک بیت افزونه (parity) به عبارت دیگر دو تک بیت **توازن منفرد** برای بازیابی اطلاعات متناظر استفاده می‌کند. **یعنی از دو دیسک اضافی برای ذخیره‌سازی دو تک بیت افزونه بهره می‌گیرد**، و البته همچنان همانند RAID 5 به صورت **توزیع شده** است. در واقع دو محاسبه توازن متفاوت از طریق دو الگوریتم محاسبه توازن متمایز انجام می‌شود و بر روی بلوک‌های مجزا در دیسک‌های متفاوت ذخیره می‌شوند. بنابراین حتی اگر دو دیسک از داده‌های اصلی کاربر دچار خرابی شود، امکان بازسازی مجدد داده‌ها وجود دارد. واضح است که آرایه RAID 6 که کاربر آن نیازمند N دیسک داده اصلی باشد از N+2 دیسک تشکیل شده‌است. دقت کنید که RAID 6 جریمه قابل توجهی برای نوشتن دارد، زیرا هر نوشتن، بر روی 2 بلوک توازن تاثیر می‌گذارد. به عبارت دیگر محاسبات دو برابری بیت توازن، به تبع جریمه نوشتن دو برابری هم در پی خواهد داشت.

توجه: در این سطح داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود.

همچنین تکنیک **Double Distributed Parity** به معنی **Parity توزیع شده دوبرابری** مورد استفاده قرار می‌گیرد، در نتیجه **افزایش قابلیت اطمینان** محقق می‌شود.

توجه: بازسازی داده‌ها ساده است. توازن اول همانند RAID 5 با استفاده از عملگر XOR و توازن دوم توسط الگوریتم متمایز دیگری انجام می‌گردد.

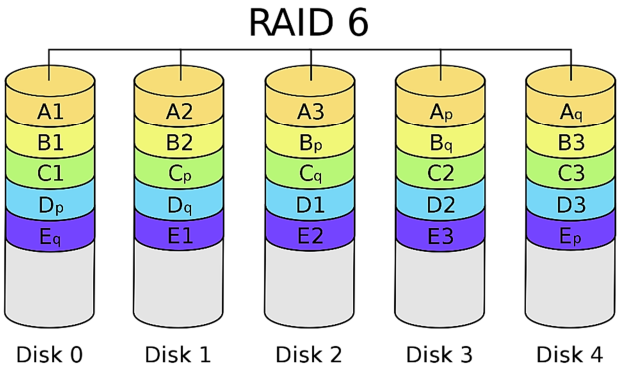
توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Parity درصدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های

khalilifar.ir

اصلی (data) و درصدی دیگر از فضای ذخیره‌سازی معادل دو تک دیسک برای نگهداری داده‌های افزونه (parity) یا پشتیبان اطلاعات شامل مجموع تک بیت‌های افزونه (معادل دو بلـوک مسـتقل)، مورد استفاده قرار می‌گیرد. ($1-\frac{2}{n}$:space efficiency).

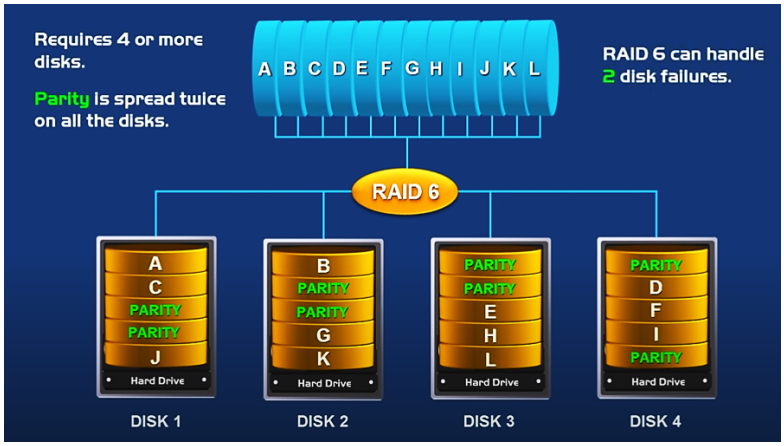
توجه: این سطح، خرابی دو دیسک را پشتیبانی می‌کند. (fault tolerance: two drive failure)

توجه: این سطح، در حداقل 4 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 4)



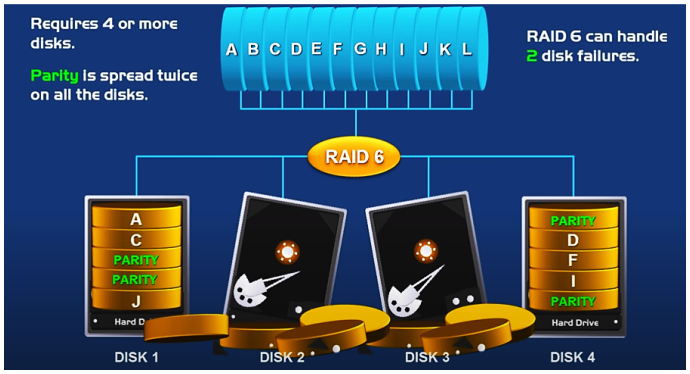
همانطور که در تصویر فوق واضح اسـت، n-2 دیسـک بـرای ذخیـره‌سـازی داده‌هـای اصـلی و 2 دیسک برای ذخیره‌سازی دو تک بیت داده افزونه (دو بلوک افزونه) بـه عنـوان پشـتیبان اطلاعـات مورد استفاده قرار می‌گیرد.

شکل زیر گویای عملکرد RAID سطح 6 است:

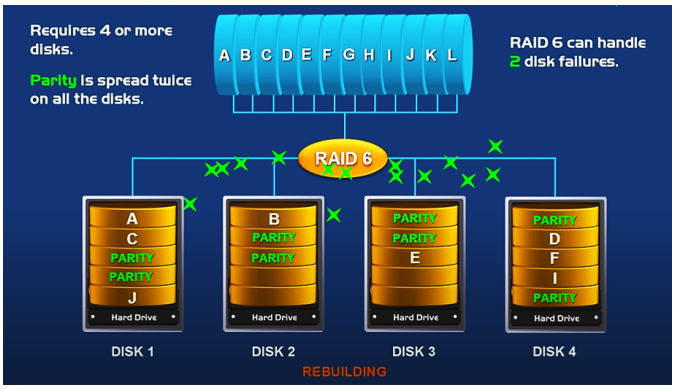


اگر دو عدد از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونـه وجـود دارد:

khalilifar.ir

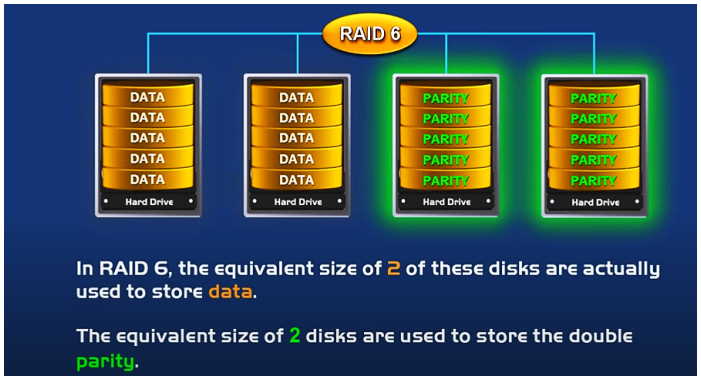


شکل زیر گویای مطلب است:



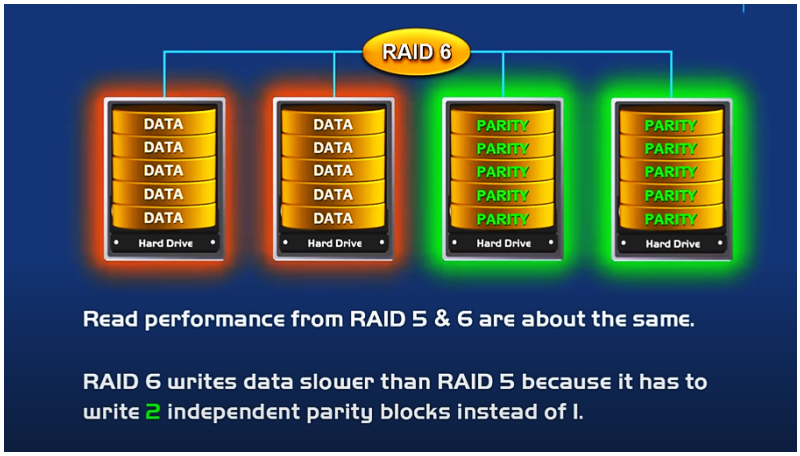
اگر سه عدد و بیشتر از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد.

این سطح افزونگی ایجاد می‌کند، برای مثال از کل 4 دیسک، 2 دیسک شامل داده اصلی و 2 دیسک برای داده افزونه توزیع شده دو برابری مورد استفاده قرار می‌گیرد، شکل زیر گویای مطلب است:



khalilifar.ir

همانطور که گفتیم RAID 6 جریمه قابل توجهی برای نوشتن دارد، زیرا هر نوشتن بر روی 2 بلوک توازن تاثیر می‌گذارد. به عبارت دیگر محاسبات دو برابری بیت توازن به تبع جریمه نوشتن دو برابری هم در پی خواهد داشت، شکل زیر گویای مطلب است:

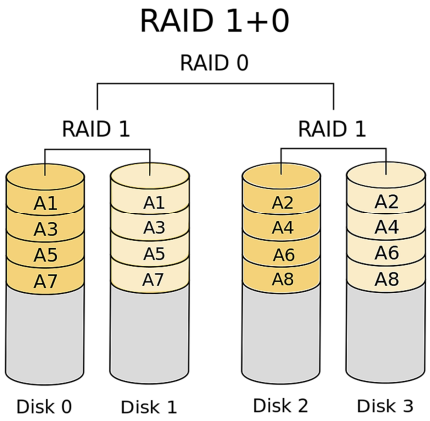


توجه: در RAID 6، تکنیک block-level Striping مورد استفاده قرار می‌گیرد و Strping و تقسیم داده‌های اصلی بزرگ و در سطح بلوک است، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی **زیاد** است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری **متوسط** است.

توجه: سایر مشخصات RAID 6 همانند RAID 5 است.

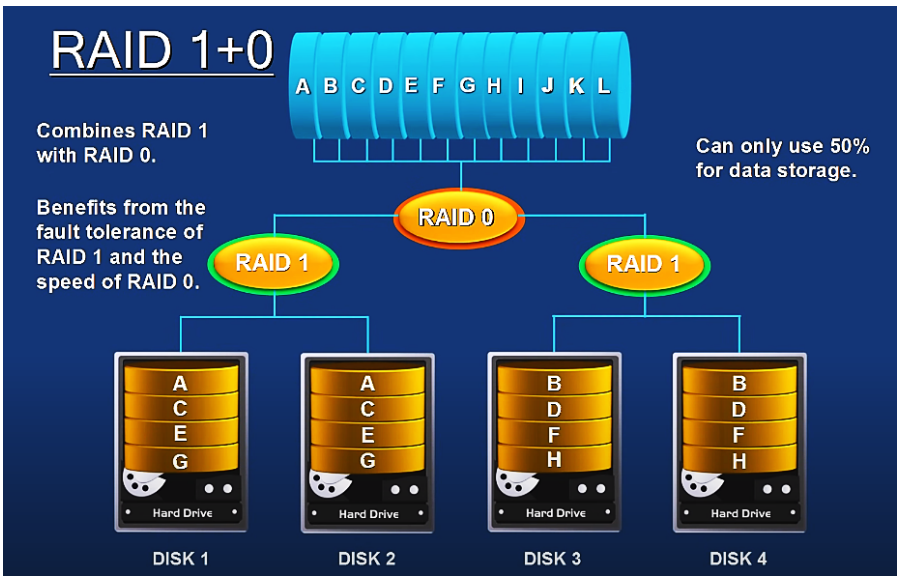
RAID سطح 10

در RAID سطح 10، تکنیک Mirroring without parity, and block-level striping مورد استفاده قرار می‌گیرد.



khalilifar.ir

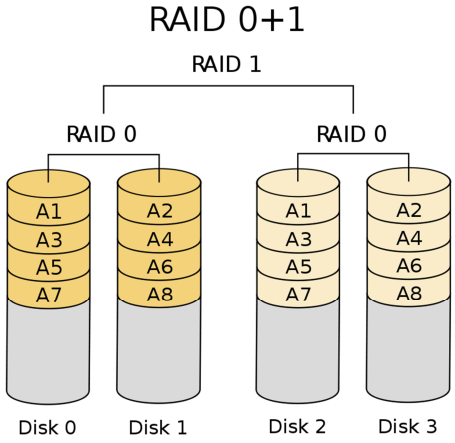
شکل زیر گویای مطلب است:



توجه: این سطح در حداقل 4 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 4)

RAID سطح 01

در RAID سطح 01، تکنیک Block-level striping, and mirroring without parity مورد استفاده قرار می‌گیرد.



توجه: این سطح در حداقل 4 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 4)

khalilifar.ir

۲- گزینه (۱) صحیح است.

از آنجاکه مطابق فرض صورت سوال، سیستم عامل باید مجبور شود دستورات S_1 ، S_2 ، S_3 و S_4 را به همین ترتیب اجرا کند، بنابراین این سوال یک مسئله همگام‌سازی توسط سمافور است و نه حل مسئله شرایط رقابتی و انحصار متقابل. دقت کنید که از ابزار سمافور هم برای حل مسئله شرایط رقابتی و انحصار متقابل و هم برای حل مسئله همگام‌سازی استفاده می‌شود.

سمافور عمومی

سمافور عمومی s از یک شمارنده و یک صف تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
Int count;
Queue Type Queue;
} s;
```

شمارنده‌ی سمافور (s.count)

برای برقراری شرط انحصار متقابل از این شمارنده با مقدار اولیه یک استفاده می‌گردد.

صف سمافور (s.queue)

برای برقراری شرط پیشروی، انتظار محدود، حل مسأله‌ی انتظار مشغول و حل مسأله‌ی اولویت معکوس از این صف استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صف نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صف به ترتیب ورود باشد، اصطلاحاً به آن **سمافور قوی** می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن **سمافور ضعیف** گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این‌گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

بر روی سمافور عمومی s دو تابع wait(s) و signal(s) عملیات ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

khalilifar.ir

```
wait (semaphore)
    {
s.count = s.count – 1;
if (s.count< 0)
{
add  this process to s.queue;
block ();
  }
}
```

توجه: راه‌حل سمافور و تابع اتمیک wait باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند علاقه‌مند به ورود به ناحیه‌ی بحرانی، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if (s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر اینصورت، فرآیند، وارد ناحیه‌ی بحرانی می‌گردد.

توجه: در سمافور عمومی، وقتی شمارنده سمافور، مقدار منفی دارد، قدر مطلق این مقدار، معرف تعداد فرآیندهای بلوکه شده در صف سمافور است.

تابع signal(s) : عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
signal (semaphore s)
{
s.count = s.count + 1
   if (s.count  < = 0)
   {
    remove a process from queue;
    wake up ();
     }
}
```

توجه: راه‌حل سمافور و تابع اتمیک signal باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه‌حل را توسط سرویس‌های سیستم عامل شبیه سازی کرد.

khalilifar.ir

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند علاقه‌مند به خروج از ناحیه‌ی بحرانی، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count<= 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به ناحیه‌ی بحرانی که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake up() بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند که از ناحیه‌ی بحرانی خارج شود، با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

توجه: با توجه به مقادیر شمارنده‌های سمافور مطرح شده در سوال، $a = 0$، $b = 0$ و $c = 0$، فقط و فقط فرآیند S_1 می‌تواند در ابتدا اجرا شود.

توجه: فرآیندهای P_1، P_2 ، P_3 و P_4 فاقد حلقه هستند، و توسط شرایطی که برای هم فراهم می‌کنند، در صورت امکان فقط و فقط می‌توانند **یکبار** اجرا شوند و تمام شوند.

گزینه اول پاسخ سوال است. زیرا: اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 فقط و فقط منجر به اجرای دستورات S_1 ، S_2 ، S_3 و S_4 به همین ترتیب می‌شود.

فرآیند	دستور	a	b	c	خروجی
P_1 ، P_2 ، P_3 و P_4	-	0	0	0	-
P_1	S_1 signal (a)	1	0	0	S_1
P_2	wait (a) S_2 signal (b)	0	1	0	S_2
P_3	wait (b) S_3 signal (c)	0	0	1	S_3
P_4	wait (c) S_4	0	0	0	S_4

توجه: در گزینه اول اگر فرآیند P_2 تحت هر شرایطی، قبل از اتمام کامل فرآیند P_1 و انجام signal(a) اجرا شود چون اول wait(a) با مقدار 0 را اجرا می‌کند، پس می‌خوابد، همچنین اگر فرآیند P_3 تحت هر شرایطی، قبل از اتمام کامل فرآیند P_2 و انجام signal(b) اجرا شود چون اول wait(b) با مقدار 0 را اجرا می‌کند، پس می‌خوابد. همچنین اگر فرآیند P_4 تحت هر شرایطی، قبل از

khalilifar.ir

اتمام کامل فرآیند P_3 و انجام signal(c) اجرا شود چون اول wait(c) با مقدار 0 را اجرا می‌کند، پس می‌خوابد. نتیجه اینکه در گزینه اول اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 در نهایت فقط و فقط منجر به اجرای دستورات S_1 ، S_2 ، S_3 و S_4 به همین ترتیب می‌شود و هیچ ترتیب و سناریوی دیگری برای اجرای دستورات وجود ندارد.

گزینه دوم پاسخ سوال نیست. زیرا: هرچند که اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 می‌تواند منجر به اجرای دستورات S_1 ، S_2 ، S_3 و S_4 به همین ترتیب شود.

فرآیند	دستور	a	b	c	خروجی
P_1 ، P_2 ، P_3 و P_4	-	0	0	0	-
P_1	S_1 signal (a) signal (b)	1	1	0	S_1
P_2	wait (b) S_2 signal (a)	2	0	0	S_2
P_3	wait (a) S_3 signal (b)	1	1	0	S_3
P_4	wait (a) wait (b) S_4	0	0	0	S_4

اما سناریوی دیگری هم وجود دارد که اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 می‌تواند منجر به اجرای دستورات S_1 ، S_2 ، S_3 و S_4 به همین ترتیب نشود. و برای مثال منجر به اجرای دستورات S_1 ، S_3 ، S_2 و S_4 شود.

فرآیند	دستور	a	b	c	خروجی
P_1 ، P_2 ، P_3 و P_4	-	0	0	0	-
P_1	S_1 signal (a) signal (b)	1	1	0	S_1
P_3	wait (a) S_3 signal (b)	0	2	0	S_3
P_2	wait (b) S_2 signal (a)	1	1	0	S_2
P_4	wait (a) wait (b) S_4	0	0	0	S_4

گزینه سوم پاسخ سوال نیست. زیرا: هرچند که اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 می‌تواند منجر به اجرای دستورات S_1 ، S_2 ، S_3 و S_4 به همین ترتیب شود.

khalilifar.ir

فرآیند	دستور	a	b	c	خروجی
P_1 ، P_2 ، P_3 و P_4	-	0	0	0	-
P_1	S_1 signal (a)	1	0	0	S_1
P_2	wait (a) S_2 signal (a) signal (a)	2	0	0	S_2
P_3	wait (a) wait (a) S_3 signal (a) signal (a) signal (a)	3	0	0	S_3
P_4	wait (a) wait (a) wait (a) S_3 signal (a) signal (a) signal (a) signal (a)	4	0	0	S_4

اما سناریوی دیگری هم وجود دارد که اجـرای فرآینـدهای P_1 ، P_2 ، P_3 و P_4 مـی‌توانـد منجـر بـه اجرای دستورات S_1 ، S_2 ، S_3 و S_4 به همین ترتیب نشود. و برای مثال منجر به خواب ابدی شود.

فرآیند	دستور	a	b	c	خروجی
P_1 ، P_2 ، P_3 و P_4	-	0	0	0	-
P_1	S_1 signal (a)	1	0	0	S_1
P_3	wait (a) wait (a) S_3 signal (a) signal (a) signal (a)	-1	0	0	می‌خوابد.
P_2	wait (a) S_2 signal (a) signal (a)	-2	0	0	می‌خوابد.
P_4	wait (a) wait (a) wait (a) S_4 signal (a) signal (a) signal (a) signal (a)	-3	0	0	می‌خوابد.

khalilifar.ir

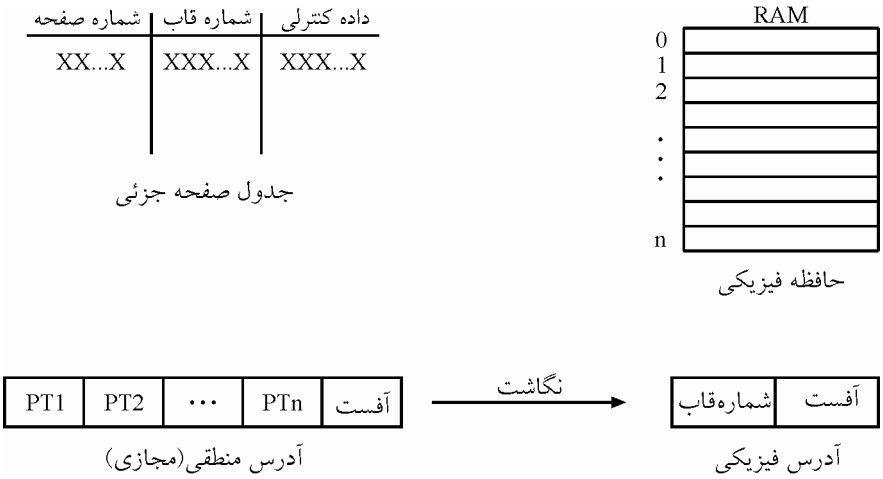
گزینه چهارم پاسخ سوال نیست. زیرا: اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 فقط و فقط منجر به خواب ابدی می‌شود.

فرآیند	دستور	a	b	c	خروجی
P_1 ، P_2 ، P_3 و P_4	-	0	0	0	-
P_1	S_1 wait (a) signal (b) signal (c)	-1	0	0	S_1 می‌خوابد.
P_2	wait (a) S_2 signal (b) signal (c)	-2	0	0	می‌خوابد.
P_3	wait (a) signal (b) S_3 signal (c)	-3	0	0	می‌خوابد.
P_4	wait (a) signal (b) signal (c) S_4	-4	0	0	می‌خوابد.

۳- گزینه (۲) صحیح است.

در اینجا برای جدول صفحه جزئی محدودیتی به اندازه یک قاب (صفحه) داریم.

بنابراین اندازه جدول صفحه جزئی برابر اندازه قاب (صفحه) می‌باشد. بنابراین برای محاسبه تعداد سطرهای جدول صفحه جزئی، کافی است، اندازه قاب که برابر اندازه جدول صفحه جزئی است بر اندازه عرض جدول صفحه جزئی تقسیم گردد. به شکل زیر توجه کنید:



khalilifar.ir

توجه: عرض جدول صفحه همواره برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب است، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد.

بنابراین داریم:

تعداد بیت‌های کنترلی + تعداد بیت‌های شماره قاب= عرض جدول صفحه جزئی

$8B$ = عرض جدول صفحه جزئی

توجه: مطابق فرض سؤال، هر مدخل جدول صفحه (عرض جدول صفحه جزئی) 8 بایت در نظر گرفته شده است.

$$\text{تعداد سطرهای جدول صفحه جزئی} = \frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^4 \times 2^{10} B}{2^3 B} = 2^{11} = 2048$$

$2^{47} B$ = اندازه فرآیند (فضای آدرس مجازی)

$$\text{اندازه قاب} = \text{اندازه صفحه} = 16KB = 16384B = 2^4 \times 2^{10} B = 2^{14} B$$

$$f : \text{تعداد صفحات فرآیند} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه}} = \frac{2^{47}}{2^{14}} = 2^{33}$$

$$r : \text{تعداد سطرهای جدول صفحه جزئی} = 2048 = 2^{11}$$

حال اطلاعات کافی برای محاسبه تعداد سطوح جدول صفحه چند سطحی را در اختیار داریم:

روش تجزیه

تعداد صفحات فرآیند باید در اندازه r(2^{11}) تجزیه گردد:

$$\text{تعداد صفحات فرآیند} = 2^{33} = 2^{\overset{\uparrow}{\textcircled{11}}} \times 2^{\overset{\uparrow}{\textcircled{11}}} \times 2^{\overset{\uparrow}{\textcircled{11}}}$$

PT1 PT2 PT3

توجه: تعداد عملوندها برابر 3 است، بنابراین تعداد سطوح جدول چند سطحی نیز برابر 3 است.

روش لگاریتم

$$\text{تعداد سطوح جدول چند سطحی} = d = \left\lceil \log_r^f \right\rceil = \left\lceil l0g_{2^{11}}^{2^{33}} \right\rceil = 3$$

روش تقسیم متوالی

$$\text{تعداد جداول صفحه جزئی در سطح سوم} = \frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{2^{33}}{2^{11}} = 2^{22}$$

$$\text{تعداد جداول صفحه جزئی در سطح دوم} = \frac{\text{تعداد جداول صفحه جزئی در سطح سوم}}{r} = \frac{2^{22}}{2^{11}} = 2^{11}$$

khalilifar.ir

$$\text{تعداد جداول صفحه جزئی در سطح اول} = \frac{\text{تعداد جداول صفحه جزئی در سطح دوم}}{r} = \frac{2^{11}}{2^{11}} = 1$$

توجه: سطح اول، یک جدول به حساب می‌آید، که 2^{11} سطر دارد.

توجه: تعداد تقسیم متوالی برابر 3 است، بنابراین تعداد سطوح جدول صفحه چند سطحی برابر 3 است.

$$\text{تعداد بیت آفست} = \log_2^{\text{اندازه صفحه}} = \log_2^{2^{14}} = 14 \text{ بیت}$$

بنابراین شکل آدرس منطقی (مجازی) به صورت زیر خواهد بود:

PT1	PT2	PT3	آفست
11 بیت	11 بیت	11 بیت	14 بیت
33 بیت			

۴- گزینه (۴) صحیح است.

یک ریاضی‌دان هلندی به‌نام Decker اولین کسی بود که یک راه‌حل نرم‌افزاری «دو فرآیندی» درست برای حل مسئله شرایط رقابتی ارائه داد. و سر انجام این راه‌حل در سال 1965 توسط Dijkstra منتشر شد. آنچه در صورت سوال مطرح شده است همان راه حل درست Decker است که هر سه شرط انحصار متقابل، انتظار محدود و پیشرفت همواره تضمین می‌شود.

ابتدا کد مطرح شده در صورت سوال را برای دو فرآیند P_0 و P_1 به صورت زیر بازنویسی می‌کنیم:

```
P0:
 {
(1) flag [0] = TRUE;
(2) while (flag [1]) {
(3) if (turn = = 1) {
(4)  flag [0] = FALSE;
(5) while (turn = = 1);
(6) flag [0] = TRUE;
}}
/ * critical section */
(7) turn = 1;
(8) flag [0] = FALSE;
/ * reminder section */
}
```

```
P1:
 {
(1) flag [1] = TRUE;
(2) while (flag [0]) {
(3) if (turn = = 0) {
(4)  flag [1] = FALSE;
(5) while (turn = = 0);
(6) flag [1] = TRUE;
}}
/ * critical section */
(7) turn = 0;
(8) flag [1] = FALSE;
/ * reminder section */
}
```

توجه: مقادیر اولیه به صورت زیر است:

turn = 0, flag [0] = FALSE, flag [1] = FALSE

khalilifar.ir

توجه: اگر فرآیند P_0 علاقه‌مند ورود به ناحیه بحرانی باشد، ابتدا flag مربوط به خود را بـه TRUE مقداردهی می‌کند (پرچم خود را بالا می‌برد) و سپس flag مربوط به فرآیند P_1 را بررسی مـی‌کنـد. همچنین اگر فرآیند P_1 علاقه‌مند ورود به ناحیه بحرانـی باشـد، ابتـدا flag مربـوط بـه خـود را بـه TRUE مقداردهی می‌کند (پرچم خود را بالا می‌برد) و سپس flag مربوط به فرآینـد P_0 را بررسـی می‌کند.

توجه: متغیر turn گویای این است که در هر لحظه کدام فرآیند حق اصرار و پافشاری جهت ورود به ناحیه بحرانی دارد. به عبارت دیگر مقدار متغیر turn حق مسلم ماست. اگر مقـدار turn برابـر 0 باشد، ورود به ناحیه بحرانی حق مسلم فرآیند P_0 است و به آن باید اصرار و پافشاری کنـد. و اگـر مقدار turn برابر 1 باشد، ورود به ناحیه بحرانی حق مسلم فرآیند P_1 و به آن باید اصرار و پافشاری کند.

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

بـرای کنتـرل برقـراری شـرط انحصـار متقابـل، شـرط پیشـرفت و شـرط انتظـار محـدود (گرسنگی و بن‌بست) از آزمون‌های زیر استفاده می‌کنیم:

توجه: ما نام این آزمون‌ها را به عنوان مبدع آن «**قوانین ارسطو**» نام‌گذاری کردیم، ایـن قـوانین بـه «**قوانین چهارگانه ارسطو**» نیز موسوم است.

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در ایـن حالت شرط انحصار متقابل نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

(1) flag [0] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

(2) while (flag [1]) {

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآینـد P_0 قرار می‌گیرد، به صورت زیر:

P0:

/*critical section*/

khalilifar.ir

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون اول وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، یعنی:

در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.

فرض کنید فرآیند P_1 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) flag [1] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = TRUE و turn = 0 است.

(2) while (flag [0]) {

توجه: شرط حلقه TRUE است، پس بدنه حلقه یعنی دستور {...} if (turn = = 0) اجرا می‌شود.

(3) if (turn = = 0) {

(4) flag [1] = FALSE;

توجه: هم اکنون مقدار turn برابر 0 است، پس شرط دستور if (turn = = 0) برابر TRUE و به تبع مقدار flag [1] برابر FALSE می‌شود.

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

(5) while (turn = = 0);

توجه: هم اکنون مقدار turn برابر 0 است، پس شرط حلقه while (turn = = 0); برابر TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P_1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P_1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

توجه: دقت کنید که انتهای حلقه دوم یعنی while (turn = = 0); نماد «;» قرار گرفته‌است. به این معنی که حلقه بدنه ندارد، به عبارت دیگر دستور flag [1] = TRUE; جزو بدنه حلقه دوم یعنی while (turn = = 0); محسوب نمی‌شود.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، خب موفق نشد. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط اول انحصار متقابل برقرار است.

khalilifar.ir

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است.

```
P0:
 {
(1) flag [0] = TRUE;
(2) while (flag [1]) {
(3) if (turn = = 1) {
(4)  flag [0] = FALSE;
(5) while (turn = = 1);
(6) flag [0] = TRUE;
}}
/ * critical section */
(7) turn = 1;
(8) flag [0] = FALSE;
/ * reminder section */
}
```

```
P1:
 {
(1) flag [1] = TRUE;
(2) while (flag [0]) {
(3) if (turn = = 0) {
(4)  flag [1] = FALSE;
(5) while (turn = = 0);
(6) flag [1] = TRUE;
}}
/ * critical section */
(7) turn = 0;
(8) flag [1] = FALSE;
/ * reminder section */
}
```

توجه: مقادیر اولیه به صورت زیر است:

turn = 0, flag [0] = FALSE, flag [1] = FALSE

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

(1) flag [0] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

همچنین فرض کنید فرآیند P_1 نیز به شکل همروند یا موازی با فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) flag [1] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = TRUE و turn = 0 است.

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین آرایه فرآیندها هر دو flag [0]= TRUE و flag [1]= TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند.

در ادامه پردازنده را از فرآیند P_1 بگیرید و به فرآیند P_0 بدهید.

khalilifar.ir

توجه: هم اکنون flag [0] = TRUE و flag [1] = TRUE و turn = 0 است.

(2) while (flag [1]) {

توجه: شرط حلقه TRUE است، پس بدنه حلقه یعنی دستور {...} if (turn = = 1) اجرا می‌شود.

(3) if (turn = = 1) {

توجه: هم اکنون مقدار turn برابر 0 است، پس شرط دستور if (turn = = 1) برابر FALSE است، پس کنترل برنامه از کل بدنه دستور if خارج شده و کنترل برنامه مجددا به ابتدای حلقه جاری یعنی while (flag [1]) باز می‌گردد.

توجه: هم اکنون مقدار flag [1] برابر TRUE است، پس شرط حلقه while (flag [1]) برابر TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P_0 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P_0 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.

توجه: هم اکنون flag [0] = TRUE و flag [1] = TRUE و turn = 0 است.

(2) while (flag [0]) {

توجه: شرط حلقه TRUE است، پس بدنه حلقه یعنی دستور {...} if (turn = = 0) اجرا می‌شود.

(3) if (turn = = 0) {

(4) flag [1] = FALSE;

توجه: هم اکنون مقدار turn برابر 0 است، پس شرط دستور if (turn = = 0) برابر TRUE و به تبع مقدار flag [1] برابر FALSE می‌شود.

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

(5) while (turn = = 0);

توجه: هم اکنون مقدار turn برابر 0 است، پس شرط حلقه while (turn = = 0); برابر TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P_1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P_1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

در ادامه پردازنده را از فرآیند P_1 بگیرید و به فرآیند P_0 بدهید.

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

(2) while (flag [1]) {

khalilifar.ir

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:
/*critical section*/

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفق نشدند. فرآیند اول و دوم نتوانستند هر دو باهم وارد ناحیه بحرانی خودشان بشوند. بنابراین شرط دوم انحصار متقابل نیز برقرار است.

توجه: برای برقرار بودن شرط انحصار متقابل باید **قانون اول ارسطو** (آزمون اول شرط انحصار متقابل) و **قانون دوم ارسطو** (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشند. بنابراین شرط انحصار متقابل در سوال مطرح شده برقرار است.

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۳)** در ادامه فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، **(گام ۴)** سپس همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۵)** در نهایت همان فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست.

توجه: مقادیر اولیه به صورت زیر است:

turn = 0, flag [0] = FALSE, flag [1] = FALSE

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
(1) flag [0] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

(2) while (flag [1]) {

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:
/*critical section*/

khalilifar.ir

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P_0:

/*critical section*/

(7) turn = 1;

(8) flag [0] = FALSE;

حال در ادامه فرآیند P_0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P_0:

/*remainder_section*/

همانطور که در (گام ۲) گفتیم قرار شد که همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P_0 مشغول حرکت است.

(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.

فرض کنید فرآیند P_1 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) flag [1] = TRUE;

توجه: هم اکنون flag [0] = FALSE و flag [1] = TRUE و turn = 1 است.

(2) while (flag [0]) {

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P_1:

/*critical section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند دوم را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۴) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_1 مشغول حرکت است.

(گام ۴): همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P_1 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P_1:

/*critical section*/

khalilifar.ir

(7) turn = 0;
(8) flag [1] = FALSE;
حال در ادامه فرآیند P_1 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P_1:
/*remainder_section
همانطور که در (گام ۴) گفتیم قرار شد که همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۵) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P_1 مشغول حرکت است.
(گام ۵): فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست. یعنی:
فرض کنید فرآیند P_1 نیز قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:
(1) flag [1] = TRUE;

توجه: هم اکنون flag [0] = FALSE و flag [1] = TRUE و turn = 0 است.

(2) while (flag [0]) {
شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P_1:
/*critical section*/
همانطور که در (گام ۵) گفتیم قرار شد که فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، خب شد، فرآیند دوم مجددا وارد ناحیه بحرانی خودش شد. بنابراین شرط پیشرفت برقرار است.

قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، **(گام ۳)** در ادامه فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۴)** در نهایت همان فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.
توجه: مقادیر اولیه به صورت زیر است:

khalilifar.ir

turn = 0, flag [0] = FALSE, flag [1] = FALSE

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

(1) flag [0] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

(2) while (flag [1]) {

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:

/*critical section*/

(گام ۲): فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.

فرض کنید فرآیند P_1 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) flag [1] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = TRUE و turn = 0 است.

(2) while (flag [0]) {

توجه: شرط حلقه TRUE است، پس بدنه حلقه یعنی دستور {...} if (turn = = 0) اجرا می‌شود.

(3) if (turn = = 0) {

(4) flag [1] = FALSE;

توجه: هم اکنون مقدار turn برابر 0 است، پس شرط دستور if (turn = = 0) برابر TRUE و به تبع مقدار flag [1] برابر FALSE می‌شود. یعنی اگر مقدار کنونی turn برابر 0 است، پس ورود به ناحیه بحرانی حق مسلم فرآیند P_0 است و به آن باید اصرار و پافشاری کند و فرآیند P_1 باید flag به معنی شمشیر و پرچم خود را غلاف کند و پایین بکشد.

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 0 است.

(5) while (turn = = 0);

توجه: هم اکنون مقدار turn برابر 0 است، پس شرط حلقه while (turn = = 0); برابر TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P_1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P_1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

khalilifar.ir

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۳) می‌شویم. هم اکنون پردازنده پشت ناحیه بحرانی فرآیند P_1 در یک حلقه انتظار، دچار انتظار مشغول است.

(گام ۳): فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P_1 بگیرید و به فرآیند P_0 بدهید.

فرض کنید فرآیند P_0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P_0:
/*critical section*/
(7) turn = 1;
(8) flag [0] = FALSE;

توجه: هم اکنون flag [0] = FALSE و flag [1] = FALSE و turn = 1 است.

حال در ادامه فرآیند P_0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P_0:
/*remainder section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۴) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P_0 مشغول حرکت است.

توجه: به طور اخلاقی، درحال حاضر مقدار کنونی turn برابر 1 است، پس ورود به ناحیه بحرانی حق مسلم فرآیند P_1 است و به آن باید اصرار و پافشاری کند و البته فرآیند P_0 مقدار flag به معنی شمشیر و پرچم خود را غلاف کرده‌است و پایین کشیده‌است یعنی flag [0] = FALSE است.

توجه بسیار مهم: اما اگر فرآیند P_0 با وجود اینکه شمشیر و پرچم خود را غلاف کرده‌است و پایین کشیده‌است، بخواهد بدعهدی، بداخلاقی و زرنگی کند و نوبت را نادیده بگیرد و از فرصت باقی‌مانده در اختیار داشتن پردازنده (CPU) سوء استفاده کند و بخواهد بارها و بارها در مدت باقی‌مانده یعنی تا زمانی که پردازنده را در اختیار دارد، ناحیه بحرانی خودش را طی کند، آنگاه مدت زمان این سوء استفاده محدود است و ابدی نیست، چون در نهایت وقت و کوانتوم فعلی پردازنده برای فرآیند P_0 تمام می‌شود و در نهایت حق به حق‌دار می‌رسد و به حساب وضعیت فرآیند P_0 رسیدگی می‌شود. و این ضرب المثل هست که اگر فردی چندبار کارهای غیراخلاقی کرده باشد و به تصادف از بازخواست نجات یافته‌باشد و به همین سبب مغرور شده و با گستاخی بخواهد بازهم به چنین کارها دست بزند به او گویند: یک بار جستی ملخک، دوبار جستی ملخک، آخر به دستی ملخک!

khalilifar.ir

(گام ۴): فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود (در این مورد خاص یعنی فرآیند P_0 کار غیراخلاقی کند چون به طور اخلاقی، درحال حاضر مقدار کنونی turn برابر 1 است، پس ورود به ناحیه بحرانی حق مسلم فرآیند P_1 است و به آن باید اصرار و پافشاری کند و نه فرآیند P_0)، آنگاه در این حالت فرآیند دوم به **ظاهر و به طور موقت و البته به ناحق** دچار «**گرسنگی موقت**» شده است، اما این گرسنگی ابدی و همیشگی نیست، یعنی:

فرض کنید فرآیند P_0 قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

(1) flag [0] = TRUE;

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE و turn = 1 است.

(2) while (flag [1]) {

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:

/*critical section*/

توجه: همانطور که واضح است فرآیند P_0 بی‌توجه به اینکه مقدار turn برابر 1 است و نوبتی هم که باشد نوبت فرآیند P_1 است که از critical section خودش استفاده کند، با بدعهدی و زرنگی توانست از critical section خودش برای «**بار دوم**» هم استفاده کند. ولی نکته اینجاست که شاید فرآیند P_0 برای «**بار دوم**» ، «**بار سوم**» و «**بار n اُم**» به طور پی در پی بتواند وارد critical section خودش به دلیل داشتن فرصت پردازنده (CPU) بشود، اما این ورود به critical section برای فرآیند P_0 ابدی و همیشگی نیست چون سرانجام زمان پردازنده منتسب شده به فرآیند P_0 پس از «**بار دوم**» به بعد تمام می‌شود و به ناچار پردازنده به فرآیند P_1 تعویض متن می‌کند.

توجه: و عدالت و حق‌خواهی دقیقا از همانجا شروع می‌شود که وقت و کوانتوم و برهه زمانی اجرای فرآیند P_0 پس از «**بار دوم**» به بعد به سر می‌رسد و تمام می‌شود و پردازنده (CPU) به فرآیند P_1 داده می‌شود.

توجه: شاید پش خودتان فکر کنید که چرا «**بار دوم**» به بعد؟! چون قرار است برای مثال یکبار «**گرسنگی موقت**» برای فرآیند P_1 ایجاد شود، اما می‌بینیم که در نهایت «**گرسنگی دائم**» برای فرآیند P_1 ایجاد نمی‌شود.

توجه: گرسنگی موقت اول برای فرآیند P_1 در بار دوم ملاقات critical section توسط فرآیند P_0 رخ می‌دهد.

توجه: بسته به اینکه فرآیند P_0 در چه شرایطی وقت پردازنده آن تمام شده باشد و پردازنده (CPU) را رها کرده‌باشد، یعنی فرآیند P_0 در چه موقعیت زمانی و مکانی از قطعه کد خود به بازی

khalilifar.ir

ناجونمردانه خود پس از «**بار دوم**» به بعد پایان می‌دهد، آنگاه شرط حلقه ;(turn = = 0) while در فرآیند P_1 مورد بررسی قرار می‌گیرد.

یکی از دو حالت زیر برای پایان مهلت زمانی پردازنده برای فرآیند P_0 وجود دارد:

حالت اول: مهلت زمانی پردازنده فرآیند P_0 در نقطه critical section برای «**بار دوم**» به بعد تمام شود:

در این حالت پردازنده از فرآیند P_0 به فرآیند P_1 تعویض متن می‌کند و در ادامه شرط حلقه جاری یعنی ;(turn = = 0) while در فرآیند P_1 مورد بررسی قرار می‌گیرد. در این حالت فرآیند P_1 از حلقه ;(turn = = 0) while خارج می‌شود، چون درحال حاضر مقدار متغیر turn برابر 1 است (متغیر turn قبلا در فرآیند P_0 مقداردهی 1 شده بود) و سپس مقدار flag [1] خود را برابر TRUE قرار می‌دهد، یعنی پرچم خود را بالا می‌برد و شمشیر خود را بیرون می‌کشد. دقت کنید که بعد انجام دستور ;flag [1] = TRUE در فرآیند P_1 کنترل برنامه به ناحیه بحرانی (critical section) نمی‌رود چون قبل critical section نماد «{» مربوط به کنترل حلقه جاری یعنی (flag [0]) while بسته شده‌است. بنابراین بعد انجام دستور ;flag [1] = TRUE در فرآیند P_1 کنترل برنامه به ابتدا و بررسی شرط حلقه (flag [0]) while می‌رود. از آنجا که مقدار flag[0] قبلا و قبل از ورود به critical section در بار دوم در فرآیند P_0 برابر TRUE شده است خط (1) و همچنین متغیر turn قبلا در فرآیند P_0 مقداردهی 1 شده است، در نتیجه بررسی شرط حلقه (flag [0]) while در فرآیند P_1 به صورت زیر است:

توجه: هم اکنون flag [0] = TRUE و flag [1] = TRUE و turn = 1 است.

(2) while (flag [0]) {

توجه: شرط حلقه TRUE است، پس بدنه حلقه یعنی دستور {...} (turn = = 0) if اجرا می‌شود.

(3) if (turn = = 0) {

(4) flag [1] = FALSE;

توجه: هم اکنون مقدار turn برابر 1 است، پس شرط دستور (turn = = 0) if برابر FALSE و به تبع خط ;flag [1] = FALSE اجرا نمی‌شود.

توجه: هم اکنون flag [0] = TRUE و flag [1] = TRUE و turn = 1 است.

(5) while (turn = = 0);

توجه: هم اکنون مقدار turn برابر 1 است، پس شرط حلقه ;(turn = = 0) while برابر FALSE است، پس کنترل برنامه از حلقه خارج شده و دستور زیر اجرا می‌شود:

(6) flag [1] = TRUE;

توجه: هم اکنون مقدار flag [0] برابر TRUE است، پس شرط حلقه (flag [0]) while برابر TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار

khalilifar.ir

نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

حالت دوم: مهلت زمانی پردازنده فرآیند P_0 در نقطه **remainder section** برای «**بار دوم**» به بعد تمام شود:

در این حالت پردازنده از فرآیند P_0 به فرآیند P_1 تعویض متن می‌کند و در ادامه شرط حلقه جاری یعنی while (turn = = 0); در فرآیند P_1 مورد بررسی قرار می‌گیرد. در این حالت فرآیند P_1 از حلقه while (turn = = 0); خارج می‌شود، چون درحال حاضر مقدار متغیر turn برابر 1 است (متغیر turn قبلا در فرآیند P_0 مقداردهی 1 شده بود) و سپس مقدار flag [1] خود را برابر TRUE قرار می‌دهد، یعنی پرچم خود را بالا می‌برد و شمشیر خود را بیرون می‌کشد. دقت کنید که بعد انجام دستور flag [1] = TRUE; در فرآیند P_1 کنترل برنامه به ناحیه بحرانی (critical section) نمی‌رود چون قبل critical section نماد «{}» مربوط به کنترل حلقه جاری یعنی while (flag [0]) بسته شده‌است. بنابراین بعد انجام دستور flag [1] = TRUE; در فرآیند P_1 کنترل برنامه به ابتدا و بررسی شرط حلقه while (flag [0]) می‌رود. از آنجا که مقدار flag[0] قبلا و قبل از ورود به remainder section در بار دوم در فرآیند P_0 برابر FALSE شده است خط (8)، پس شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P1:

/*critical section*/

که در نهایت حق به حق‌دار می‌رسد و سرانجام فرآیند P_1 نیز می‌تواند وارد ناحیه بحرانی خودش شود.

همانطور که در (گام ۴) گفتیم قرار شد که فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که به **طور مکرر** مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب نشد، فرآیند اول نتوانست به طور مکرر مجددا وارد ناحیه بحرانی خودش بشود. بنابراین شرط انتظار محدود به دلیل نبود گرسنگی برقرار است.

قانون دوم ارسطو(آزمون بن‌بست)

جهت بررسی بن‌بست از همان **قانون دوم** استفاده می‌شود. در واقع روال بررسی همان قانون دوم است، اما نتیجه قانون متفاوت است.

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت

khalilifar.ir

ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند که در این حالت «**بن‌بست**» رخ داده است.

همانطور که در **آزمون دوم** گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. خب همانطور که در آزمون دوم دیدید هر دو فرآیند باهم پشت ناحیه بحرانی خودشان مسدود نشدند. فرآیند اول توانست وارد ناحیه بحرانی خودش شود، اما فرآیند دوم نتوانست وارد ناحیه بحرانی خودش شود. بنابراین بن‌بست رخ نداده است.

توجه: برای برقرار بودن شرط انتظار محدود باید **قانون دوم ارسطو** (آزمون بن‌بست) و **قانون چهارم ارسطو**(آزمون گرسنگی) هر دو باهم برقرار باشند. بنابراین شرط انتظار محدود در سوال مطرح شده برقرار است.

توجه: پُر واضح است که گزینه چهارم پاسخ سوال است، زیرا هر سه شرط انحصار متقابل، انتظار محدود و پیشرفت همواره تضمین می‌شود.

۵- گزینه (۲) صحیح است.

در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

$\sum_{i=1}^{n} \text{Request}[i] < m+n$: مجموع درخواست‌های فرآیندها برای منابع

$$\rightarrow 3+4+6 < m+n \rightarrow 13 < m+3 \rightarrow m > 10 \rightarrow m_{min} = 11$$

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند

khalilifar.ir

اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را بـه سیسـتم برمـی‌گردانـد. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.
اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد احتمال بن‌بست به صورت زیر است:

$$2+3+5=m$$

حال اگر مقدار m حداقل یک واحد بیشتر از مجموع $5+3+2$ یعنی 10 شود، آنگاه سیسـتم دیگـر دچار بن‌بست نمی‌شود، به صورت زیر:

$$m>10 \rightarrow m_{min}=11$$

۶- گزینه (۳) صحیح است.

در سیستم‌های بی‌درنگ، زمان پاسخگویی به فرآیندها نباید از یک حد مشخصـی (مهلـت زمـانی) بیش‌تر شود، در غیر اینصورت از بین خواهند رفت. سیسـتم‌هـای بـی‌درنـگ معمـولا بـه شـیوه‌ای پیاده‌سازی می‌شوند که باید به اتفاقات و حوادث خارجی سریعا پاسخ دهند. این وقایع به دو طبقه متناوب و غیرمتناوب تقسیم می‌شوند:

وقایع متناوب (مساوی): آن دسته از وقایعی که در فواصل زمانی مـنظم و مسـاوی رخ مـی‌دهنـد، مانند استراحت دستگاه فتوکپی پس از چاپ 500 برگه در فواصل زمانی منظم و مساوی.

وقایع غیرمتناوب (غیرمساوی): آن دسته از وقایعی که در فواصل زمانی نامنظم و غیرمسـاوی رخ می‌دهند، مانند فشردن پدال ترمز اتومبیل در فواصل زمانی نامنظم و غیرمساوی.

شرط زمان‌بندی

زمان‌بندی تمام وقایع متناوب باهم در یک سیستم بی‌درنـگ بـر اسـاس رابطـه زیـر ممکـن اسـت **امکان‌پذیر** باشد:

$$\sum_{i=1}^{m}\frac{t_i}{p_i}\leq 1$$

در رابطه فوق m تعداد وقایع متناوب در سیستم و i رخدادی که اتفاق افتاده است و t_i مدت زمانی که اجرای واقعه به پردازنده نیاز دارد و p_i دوره تناوب واقعه است.

توجه: کلمه **امکان‌پذیر بودن** در عبارت فوق، به معنی حتمی بودن و قطعی بودن نیست، بلکـه بـه این معنی است که زمان‌بندی تمام وقایع متناوب باهم در یک سیستم بی‌درنگ بر اسـاس برقـراری معیار فوق ممکن است شدنی باشد یا نباشد.

توجه: یک سیستم بی‌درنگ که منطبق با معیار فوق نباشد، به طور قطع غیرقابل زمـان‌بنـدی اسـت، اما اگر این معیار را داشته باشد یک الگوریتم خوب **ممکن است** بتواند آنرا زمان‌بندی کرده و همـه مهلت‌ها را محقق کند.

khalilifar.ir

سوال: در یک سیستم بی‌درنگ نرم، 3 واقعه متناوب با دوره‌های تناوب 100 و 200 و 500 میلی‌ثانیه وجود دارند، اگر زمان پردازش هر واقعه به ترتیب 50 و 30 و 100 میلی‌ثانیه باشد، آیا این سیستم قابل زمان‌بندی است؟

پاسخ:

$$\sum_{i=1}^{3}\frac{t_i}{p_i}=\frac{50}{100}+\frac{30}{200}+\frac{100}{500}=0.5+0.15+0.2=0.85\le 1$$

از آنجاکه شرط فوق برقرار است، ممکن است زمان‌بندی وقایع فوق در عمل امکان‌پذیر باشد.

توجه: به طور کلی الگوریتم‌های زمان‌بندی سیستم‌های بی‌درنگ به دو طبقه **ایستا** و **پویا** تقسیم می‌شود. در الگوریتم‌های ایستا، اولویت زمان‌بندی قبل از اجرای فرآیندها تعیین می‌شود (پیشا اجرا)، اما در الگوریتم‌های پویا، اولویت زمان‌بندی حین اجرای فرآیندها تعیین می‌شود (حینا اجرا). معروف‌ترین الگوریتم ایستا، نرخ یکنواخت (Rate-Monotonic Scheduling) و معروف‌ترین الگوریتم پویا، ابتدا زودترین مهلت (EDF: Earliest Deadline First) است.

الگوریتم نرخ یکنواخت

الگوریتم زمان‌بندی بی‌درنگ «نرخ یکنواخت» برای اجرای فرآیندهای متناوب از سیاست اولویت «ایستا» از نوع «غیرانحصاری» (preemptive) یا قبضه‌ای، قابل تخلیه پیش‌هنگام و قابل پس گرفتن استفاده می‌کند. در این الگوریتم، در ابتدای کار به هر فرآیند یک اولویت ایستا، ثابت، غیرقابل تغییر و برای همیشه بر اساس دوره تناوب داده می‌شود. که این اولویت با عکس دوره تناوب هر فرآیند رابطه مستقیم دارد، به صورت زیر:

$$\text{اولویت (فرکانس)}=\frac{1}{\text{دوره تناوب}}$$

توجه: مطابق رابطه فوق فرآیندهای با دوه تناوب کمتر، دارای اولویت بیشتر خواهند بود.

در زمان اجرای فرآیندها، زمان‌بند پردازنده، همیشه فرآیندی که بیشترین اولویت را دارد، انتخاب می‌کند و چنانچه فرآیندی با اولویت بیشتر در صف آماده حضور یابد، بلافاصله فرآیند جاری را قبضه خواهد کرد.

مثال اول: دو فرآیند متناوب P_1 و P_2 با دوره تناوب به ترتیب برابر p1=50 و p2=100 میلی‌ثانیه و زمان پردازش (CPU Burst) به ترتیب برابر t1=20 و t2=35 را در نظر بگیرید، همچنین فرض کنید مهلت زمانی انجام هر فرآیند برابر همان دوره تناوب باشد، آیا این سیستم قابل زمان‌بندی است؟

	Period	CPU Time
P_1	50	20
P_2	100	35

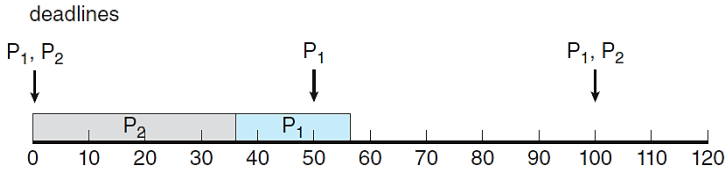
khalilifar.ir

پاسخ:

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{20}{50}+\frac{35}{100}=0.4+0.35=0.75\le 1$$

واضح است که در مجموع، بهره‌وری فرآیندهای متناوب P_1 و P_2 برابر 0.75 است. بنابراین از نظر تئوری مهلت‌های زمانی فرآیندها قابل زمان‌بندی است. اما برای بررسی قابل زمان‌بندی بودن آن به طور قطعی و عملی باید نمودار گانت آن رسم شود.

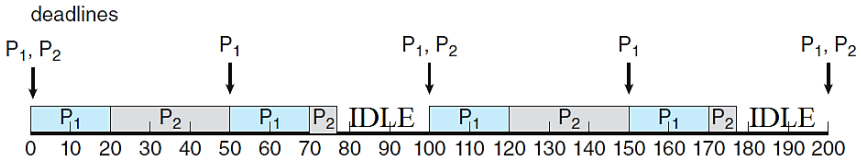
اگر مثال فوق را بر اساس تعریف الگوریتم نرخ یکنواخت در نظر بگیریم، آنگاه اولویت بیشتر به فرآیند P_1 داده می‌شود که دوره تناوب کمتر و به تبع فرکانس و اولویت بیشتری دارد. اما به جهت درک بیشتر ابتدا فرض کنید ملاک الگوریتم نرخ یکنواخت نباشد، و الگوریتمی ملاک باشد که اولویت را به P_2 می‌دهد. پس فرض کنید که به فرآیند P_2، اولویت بالاتری نسبت به اولویت P_1 می‌دهیم. در این صورت ترتیب اجرای فرآیندها به صورت شکل زیر است:



Scheduling of tasks when P_2 has a higher priority than P_1.

واضح است که ابتدا P_2، اجرای خود را آغاز نموده و در زمان 35 خاتمه می‌یابد. در این نقطه، P_1 کار خود را آغاز کرده و اجرای خود را در زمان 55 خاتمه می‌دهد، درحالیکه مهلت زمانی اول (تناوب اول) برای فرآیند P_1 در زمان 50 به پایان می‌رسد ولی اجرایش تا زمان 55 ادامه داشته‌است. بنابراین مهلت زمانی برای P_1 برآورده نشده‌است پس الگوریتم مذکور نتوانسته‌است مهلت‌ها را محقق کند. در سیستم‌های بی‌درنگ، زمان پاسخگویی به فرآیندها نباید از یک حد مشخصی (مهلت زمانی) بیش‌تر شود، در غیر اینصورت از بین خواهند رفت.

حال فرض کنید که **الگوریتم زمان‌بندی نرخ یکنواخت** مورد استفاده قرار گیرد. که در این شرایط فرآیند P_1 اولویت بالاتری را نسبت به P_2 دریافت می‌کند. زیرا فرآیند P_1 دوره تناوب کمتر و به تبع فرکانس و اولویت بیشتری دارد. در این صورت ترتیب اجرای فرآیندها به صورت شکل زیر است:



Rate-monotonic scheduling.

khalilifar.ir

واضح است که ابتدا P_1، اجرای خود را آغاز نموده و در زمان 20 خاتمه می‌یابد. و مهلت زمانی اول فرآینده P_1 هم برآورده می‌شود. در این نقطه، P_2 اجرای خود را آغاز کرده و تا لحظه 50 اجرا می‌شود. در لحظه 50 با اینکه 5 میلی‌ثانیه از زمان اجرای فرآیند P_2 باقی‌مانده‌است اما مطابق الگوریتم نرخ یکنواخت، پردازنده از P_2 (اولویت کمتر) گرفته می‌شود و به P_1 (اولویت بیشتر) داده می‌شود چون در لحظه 50 رویداد راه‌انداز بار دوم برای فرآیند P_1 از راه رسیده‌است. مطابق تعریف الگوریتم نرخ یکنواخت همواره اولویت با فرآیند با اولویت بیشتر و به صورت «غیرانحصاری» (preemptive) یا قبضه‌ای است. در ادامه فرآیند P_1 پردازش خود را در نقطه 70 کامل می‌کند که در این لحظه P_2 از پایان اجرای قبلی، کار خود را ادامه می‌دهد. P_2 پردازش خود را در لحظه 75 تکمیل می‌کند و اولین مهلت زمانی خودش را هم برآورده می‌کند. دقت کنید که فرآیند P_1 هر 50 میلی‌ثانیه یکبار و در لحظات 0 و 50 و 100 و 150 و 200 و ... و فرآیند P_2 هر 100 میلی‌ثانیه یکبار و در لحظات 0 و 100 و 200 و ... و این چرخه ادامه دارد و در نهایت مهلت زمانی هر دو فرآیند محقق می‌شود.

توجه: الگوریتم نرخ یکنواخت بین تمام الگوریتم‌های بی‌درنگ ایستا از همه «**بهینه‌تر**» است، و اگر این الگوریتم نتواند مهلت‌های زمانی فرآیندها را محقق کند، هیچ الگوریتم دیگری قادر به تحقق مهلت‌های زمانی نخواهد بود.

مثال دوم: دو فرآیند متناوب P_1 و P_2 با دوره تناوب به ترتیب برابر p1=50 و p2=80 میلی‌ثانیه و زمان پردازش (CPU Burst) به ترتیب برابر t1=25 و t2=35 را در نظر بگیرید، همچنین فرض کنید مهلت زمانی انجام هر فرآیند برابر همان دوره تناوب باشد، آیا این سیستم قابل زمان‌بندی است؟

	Period	CPU Time
P_1	50	25
P_2	80	35

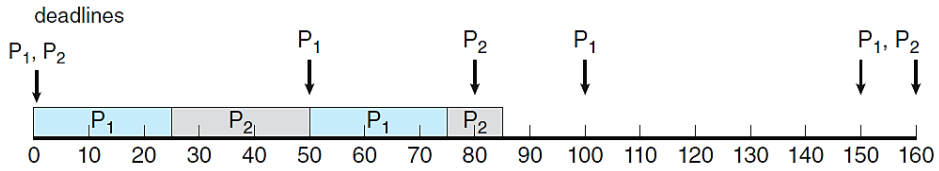
پاسخ:

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{35}{80}=0.5+0.44=0.94\le 1$$

واضح است که در مجموع، بهره‌وری فرآیندهای متناوب P_1 و P_2 برابر 0.94 است. بنابراین از نظر تئوری مهلت‌های زمانی فرآیندها قابل زمان‌بندی است. اما برای بررسی قابل زمان‌بندی بودن آن به طور قطعی و عملی باید نمودار گانت آن رسم شود.

فرض کنید که **الگوریتم زمان‌بندی نرخ یکنواخت** مورد استفاده قرار گیرد. که در این شرایط فرآیند P_1 اولویت بالاتری را نسبت به P_2 دریافت می‌کند. زیرا فرآیند P_1 دوره تناوب کمتر و به تبع فرکانس و اولویت بیشتری دارد. در این صورت ترتیب اجرای فرآیندها به صورت شکل زیر است:

khalilifar.ir



Missing deadlines with rate-monotonic scheduling.

واضح است که ابتدا P_1، اجرای خود را آغاز نموده و در زمان 25 خاتمه می‌یابد. و مهلت زمانی اول فرآینده P_1 هم برآورده می‌شود. در این نقطه، P_2 اجرای خود را آغاز کرده و تا لحظه 50 اجرا می‌شود. در لحظه 50 با اینکه 10 میلی‌ثانیه از زمان اجرای فرآیند P_2 باقی‌مانده‌است اما مطابق الگوریتم نرخ یکنواخت، پردازنده از P_2 (اولویت کمتر) گرفته می‌شود و به P_1 (اولویت بیشتر) داده می‌شود چون در لحظه 50 رویداد راه‌انداز بار دوم برای فرآیند P_1 از راه رسیده‌است. مطابق تعریف الگوریتم نرخ یکنواخت همواره اولویت با فرآیند با اولویت بیشتر و به صورت «غیرانحصاری» (preemptive) یا قبضه‌ای است. در ادامه فرآیند P_1 پردازش خود را در نقطه 75 کامل می‌کند که در این لحظه P_2 از پایان اجرای قبلی، کار خود را ادامه می‌دهد. P_2 پردازش خود را در لحظه 85 تکمیل می‌کند غافل از اینکه در لحظه 80 رویداد راه انداز P_2 مجدد رخ داده است و بنابراین مهلت P_2 تا لحظه 80 بوده و نه لحظه 85 و پایان اجرایش در لحظه 85 به معنی دیر رسیدن همانند هرگز نرسیدن است. در سیستم‌های بی‌درنگ، زمان پاسخگویی به فرآیندها نباید از یک حد مشخصی (مهلت زمانی) بیش‌تر شود، در غیر اینصورت از بین خواهند رفت.

دقت کنید که فرآیند P_1 هر 50 میلی‌ثانیه یکبار و در لحظات 0 و 50 و 100 و 150 و ... و فرآیند P_2 هر 80 میلی‌ثانیه یکبار و در لحظات 0 و 80 و 160 و ... و این چرخه ادامه دارد و در نهایت مهلت زمانی هر دو فرآیند محقق نمی‌شود.

توجه: ممکن است در مسئله‌ای معیار قابلیت زمان‌بندی در تئوری برقرار باشد، یعنی بهره‌وری پردازنده کمتر یا مساوی 1 باشد، اما در عمل و در نمودار گانت مهلت‌های زمانی فرآیندها محقق نشود.

توجه: حداقل مقدار بهره‌وری پردازنده (CPU) در الگوریتم زمان‌بندی نرخ یکنواخت برای سیستمی شامل N فرآیند از رابطه زیر محاسبه می‌شود:

$$N(2^{\frac{1}{N}}-1)$$

مقادیر زیر برای رابطه فوق برقرار است:

$N(2^{\frac{1}{N}}-1)$	N	Utilization
	1	1
	2	0.83
	∞	0.69

khalilifar.ir

توجه: مطابق جدول فوق، واضح است که با در نظر گرفتن فقط و فقط یک فرآیند در سیستم بی‌درنگ، بهره‌وری پردازنده (CPU) برابر 100 درصد خواهد بود.

توجه: در مثال اول، بهره‌وری پردازنده (CPU) در دو حالت بدون الگوریتم نرخ یکنواخت و با الگوریتم نرخ یکنواخت برابر مقدار 0.75 بود و دیدید که در حالت با الگوریتم نرخ یکنواخت، سیستم توانست مهلت زمانی فرآیندها را برآورده سازد. البته در حالت با الگوریتم نرخ یکنواخت، مقدار 0.75 از مقدار 0.83 به عنوان **حداقل مقدار بهره‌وری پردازنده** (CPU) در حالت 2 فرآیندی **کمتر** هم بود، مطابق رابطه‌ی زیر:

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{20}{50}+\frac{35}{100}=0.4+0.35=0.75\le N(2^{\frac{1}{N}}-1)=0.83\ \le 1$$

توجه: بنابراین در حالت کمتر یا مساوی حداقل مقدار بهره‌وری پردازنده (CPU)، الگوریتم زمان‌بندی نرخ یکنواخت، «تضمین» می‌کند که فرآیندها بگونه‌ای زمان‌بندی شوند که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شود.

توجه: در مثال دوم، بهره‌وری پردازنده (CPU) در دو حالت بدون الگوریتم نرخ یکنواخت و با الگوریتم نرخ یکنواخت برابر مقدار 0.94 بود و دیدید که در حالت با الگوریتم نرخ یکنواخت، سیستم نتوانست مهلت زمانی فرآیندها را برآورده سازد. البته در حالت با الگوریتم نرخ یکنواخت، مقدار 0.94 از مقدار 0.83 به عنوان **حداقل مقدار بهره‌وری پردازنده** (CPU) در حالت 2 فرآیندی **بیشتر** هم بود، مطابق رابطه‌ی زیر:

$$N(2^{\frac{1}{N}}-1)=0.83\le\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{35}{80}=0.5+0.44=0.94\le 1$$

بنابراین در این حالت الگوریتم زمان‌بندی نرخ یکنواخت، تضمین نمی‌کند که فرآیندها بگونه‌ای زمان‌بندی شوند که مهلت‌های زمانی فرآیندها برآورده شود. دقت کنید که تضمین نمی‌کند یعنی ممکن است در این شرایط الگوریتم نرخ یکنواخت نتواند از پس تحقق مهلت‌های زمانی فرآیندها برآید، نه اینکه حتما نمی‌تواند، به عبارت دیگر شاید بتواند و شاید نتواند.

در صورت سوال مطرح شده‌است که دو پردازه متناوب با مشخصات زیر مفروض است. کدام گزینه بزرگترین مقدار x را برای پردازه 2 نشان می‌دهد به نحوی که زمان‌بندی قبضه‌ای (نرخ یکنواخت) Rate Monotonic امکان‌پذیر باشد؟

	Period	CPU Time
P_1	50	25
P_2	80	x

پاسخ: برای کشف بزرگترین مقدار x برای P_2 به نحوی که زمان‌بندی قبضه‌ای (نرخ یکنواخت) Rate Monotonic امکان‌پذیر باشد باید گزینه‌هایی که شرط زمان‌بندی را برقرار می‌کنند، در نمودار

khalilifar.ir

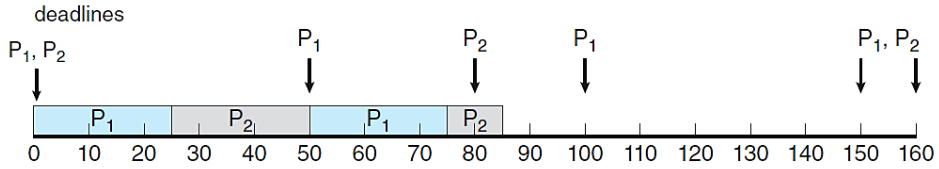
گانت نیز بررسی شوند. به عبارت دیگر هم از نظر تئوری مهلت‌های زمانی فرآیندها باید قابل زمان‌بندی باشد. و هم از نظر عملی نمودار گانت آن به طور کامل و دوره‌ای قابل زمان‌بندی باشد.

گزینه چهارم پاسخ سوال نیست.

	Period	CPU Time
P_1	50	25
P_2	80	x=35

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{35}{80}=0.5+0.44=0.94\le 1$$

همانطور که در مثال دوم بررسی کردیم، دیدید که مقادیر فوق در نمودار گانت قابل زمان‌بندی کامل و دوره‌ای نبود.



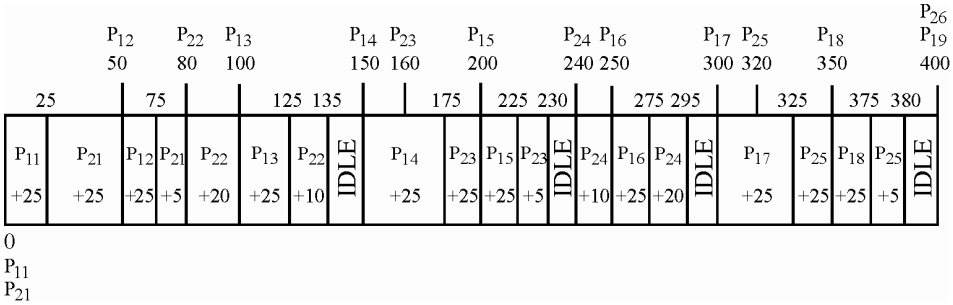
Missing deadlines with rate-monotonic scheduling.

گزینه سوم پاسخ سوال است.

	Period	CPU Time
P_1	50	25
P_2	80	x=30

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{30}{80}=0.5+0.375=0.875\le 1$$

مقادیر فوق از نظر تئوری درست است، اما باید از نظر عملی نیز نمودار گانت آن نیز بررسی شود:



khalilifar.ir

واضح است که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شده‌است، چون در لحظـه 400 به تناوب و ملاقات مجدد فرآیندها باهم رسیده‌است.

در گزینه سوم مهلت‌های زمانی فرآیندها از نظر **تئوری** و **عملی** قابـل زمـان‌بنـدی اسـت. و پاسـخ سوال هم هست چون صورت سوال، **بزرگترین** مقدار ممکن را خواسته‌است.

گزینه دوم پاسخ سوال نیست.

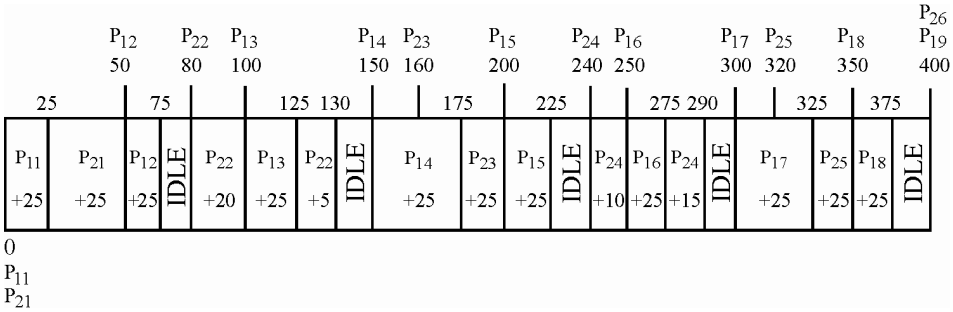
	Period	CPU Time
P_1	50	25
P_2	80	x=25

$$\sum_{i=I}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{25}{80}=0.5+0.3125=0.8125\le 1$$

بر اساس الگوریتم نرخ یکنواخت، مقدار 0.8125 از مقدار 0.83 به عنوان **حداقل مقدار بهـره‌وری پردازنده** (CPU) در حالت 2 فرآیندی **کمتر** است، مطابق رابطه‌ی زیر:

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{25}{80}=0.5+0.3125=0.8125\le N(2^{\frac{1}{N}}-1)=0.83\ \le 1$$

توجه: همانطور که پیش‌تر گفتیم، در حالت کمتر یا مساوی حداقل مقدار بهـره‌وری پردازنـده (CPU)، الگوریتم زمان‌بندی نرخ یکنواخت، «تضمین» می‌کند که فرآیندها بگونه‌ای زمان‌بندی شوند که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شود.



واضح است که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شده‌است، چون در لحظـه 400 به تناوب و ملاقات مجدد فرآیندها باهم رسیده‌است.

در گزینه دوم مهلت‌های زمانی فرآیندها از نظر **تئوری** و **عملی** قابل زمان‌بنـدی اسـت. امـا پاسـخ سوال نیست چون صورت سوال، **بزرگترین** مقدار ممکن را خواسته‌است.

khalilifar.ir

گزینه اول پاسخ سوال نیست.

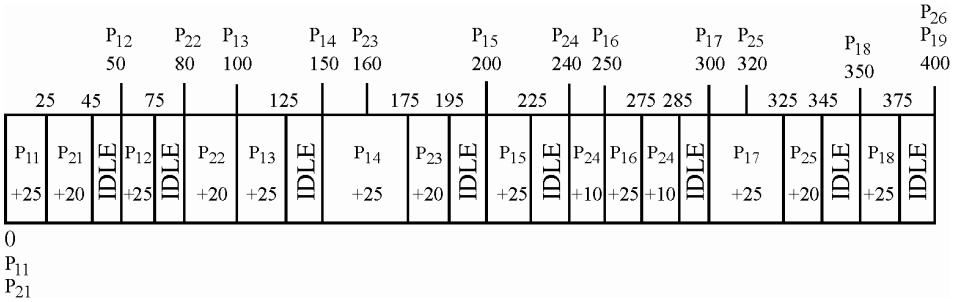
	Period	CPU Time
P_1	50	25
P_2	80	x=20

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{20}{80}=0.5+0.25=0.75\le 1$$

بر اساس الگوریتم نرخ یکنواخت، مقدار 0.75 از مقدار 0.83 به عنوان **حداقل مقدار بهره‌وری پردازنده** (CPU) در حالت 2 فرآیندی **کمتر** است، مطابق رابطه‌ی زیر:

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{20}{80}=0.5+0.25=0.75\le N(2^{\frac{1}{N}}-1)=0.83\ \le 1$$

توجه: همانطور که پیش‌تر گفتیم، در حالت کمتر یا مساوی حداقل مقدار بهره‌وری پردازنده (CPU)، الگوریتم زمان‌بندی نرخ یکنواخت، «تضمین» می‌کند که فرآیندها بگونه‌ای زمان‌بندی شوند که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شود.



واضح است که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شده‌است، چون در لحظه 400 به تناوب و ملاقات مجدد فرآیندها باهم رسیده‌است.

در گزینه اول مهلت‌های زمانی فرآیندها از نظر **تئوری** و **عملی** قابل زمان‌بندی است. اما پاسخ سوال نیست چون صورت سوال، **بزرگترین** مقدار ممکن را خواسته‌است.

مقدار X در حداکثر (سقف) بهره‌وری از رابطه زیر محاسبه می‌شود:

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{x}{80}\le 1$$

$$\frac{x}{80}\le\frac{1}{2}$$

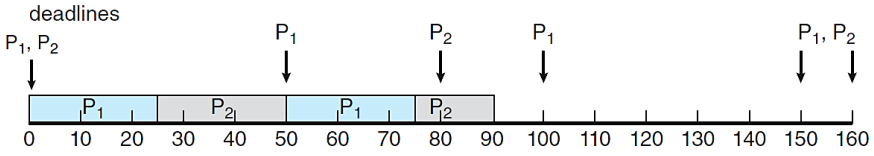
$$x\le 40\rightarrow x=40$$

khalilifar.ir

	Period	CPU Time
P_1	50	25
P_2	80	x=40

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{40}{80}=0.5+0.5=1\le 1$$

واضح است که در مجموع، بهره‌وری فرآیندهای متناوب P_1 و P_2 برابر 1 و در حالت حداکثری است. بنابراین از نظر تئوری مهلت‌های زمانی فرآیندها قابل زمان‌بندی است. اما برای بررسی قابل زمان‌بندی بودن آن به طور قطعی و عملی باید نمودار گانت آن رسم شود.



Missing deadlines with rate-monotonic scheduling.

واضح است که ابتدا P_1، اجرای خود را آغاز نموده و در زمان 25 خاتمه می‌یابد. و مهلت زمانی اول فرآینده P_1 هم برآورده می‌شود. در این نقطه، P_2 اجرای خود را آغاز کرده و تا لحظه 50 اجرا می‌شود. در لحظه 50 با اینکه 15 میلی‌ثانیه از زمان اجرای فرآیند P_2 باقی‌مانده‌است اما مطابق الگوریتم نرخ یکنواخت، پردازنده از P_2 (اولویت کمتر) گرفته می‌شود و به P_1 (اولویت بیشتر) داده می‌شود چون در لحظه 50 رویداد راه‌انداز بار دوم برای فرآیند P_1 از راه رسیده‌است. مطابق تعریف الگوریتم نرخ یکنواخت همواره اولویت با فرآیند با اولویت بیشتر و به صورت «غیرانحصاری» (preemptive) یا قبضه‌ای است. در ادامه فرآیند P_1 پردازش خود را در نقطه 75 کامل می‌کند که در این لحظه P_2 از پایان اجرای قبلی، کار خود را ادامه می‌دهد. P_2 پردازش خود را در لحظه 90 تکمیل می‌کند غافل از اینکه در لحظه 80 رویداد راه انداز P_2 مجدد رخ داده است و بنابراین مهلت P_2 تا لحظه 80 بوده و نه لحظه 90 و پایان اجرایش در لحظه 90 به معنی دیر رسیدن همانند هرگز نرسیدن است. در سیستم‌های بی‌درنگ، زمان پاسخگویی به فرآیندها نباید از یک حد مشخصی (مهلت زمانی) بیش‌تر شود، در غیر اینصورت از بین خواهند رفت.

دقت کنید که فرآیند P_1 هر 50 میلی‌ثانیه یکبار و در لحظات 0 و 50 و 100 و 150 و ... و فرآیند P_2 هر 80 میلی‌ثانیه یکبار و در لحظات 0 و 80 و 160 و ... و این چرخه ادامه دارد و در نهایت مهلت زمانی هر دو فرآیند محقق نمی‌شود.

مقدار X در حداقل (کف) بهره‌وری از رابطه زیر محاسبه می‌شود:

khalilifar.ir

$$\sum_{i=1}^{2}\frac{t_i}{p_i}=\frac{25}{50}+\frac{x}{80}\le N(2^{\frac{1}{N}}-1)$$

$$\frac{1}{2}+\frac{x}{80}\le 2(2^{\frac{1}{2}}-1)$$

$$0.5+\frac{x}{80}\le 0.83$$

$$\frac{x}{80}\le 0.83-0.5$$

$$\frac{x}{80}\le 0.33\rightarrow x\le 80\times 0.33\rightarrow x\le 26.4$$

توجه: همانطور که دیدید مقادیر x در گزینه اول x = 20 و گزینه دوم x = 25 کوچکتر از مقدار **حداقل** (کف) بهره‌وری یعنی $20 < 25 < 26.4$است. **پس همانطور که پیش‌تر گفتیم، در حالت کمتر یا مساوی حداقل مقدار بهره‌وری پردازنده (CPU)، الگوریتم زمان‌بندی نرخ یکنواخت، «تضمین» می‌کند که فرآیندها بگونه‌ای زمان‌بندی شوند که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شود، که دیدید چنین هم شد.**

۷- گزینه (۳) صحیح است.

الگوریتم‌های Priority (زمانبندی با اولویت)

در این روش زمانبندی، هریک از فرآیندها اولویت مخصوص به خود را دارند. این اولویت معمولاً از خارج سیستم مشخص می‌شود. منطقی به نظر می‌رسد که اولویت فرآیندی مربوط به رئیس از اولویت فرآیندی مربوط به کارمند بالاتر باشد. ایده اصلی این الگوریتم بسیار ساده و مشخص است. هر فرآیند باید یک اولویت داشته باشد و در هر لحظه فرآیندی اجرا می‌شود که بالاترین اولویت را دارد.

توجه: الگوریتم‌های Priority را می‌توان هم به صورت انحصاری و هم به صورت غیرانحصاری پیاده‌سازی کرد.

توجه: تنوع الگوریتم‌های زمانبندی با اولویت، بسیار زیاد است و انواع مختلفی از آن وجود دارد که به عنوان مثال می‌توان به SJF، SRT و HRRN اشاره کرد. البته در این سه الگوریتم، اولویت فرآیندها در داخل سیستم و براساس شرایط مشخص می‌شود.

توجه: در الگوریتم‌هایPriority، فرآیندهای با اولویت کمتر، دچار قحطی‌زدگی می‌شوند.

توجه: یک اولویت می‌تواند استاتیک یا دینامیک باشد:

- یک اولویت استاتیک هیچ‌گاه تغییر نمی‌کند، به همین دلیل پیاده‌سازی آن آسان است.
- یک اولویت دینامیک براثر تغییراتی که در محیط اتفاق می‌افتد تغییر می‌کند.

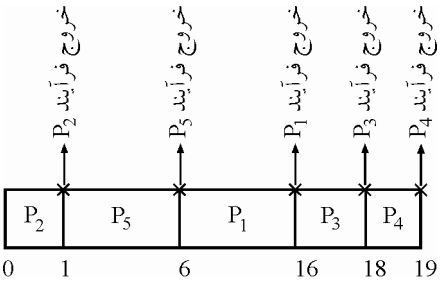
توجه: در الگوریتم‌های اولویت، جهت مقابله با مشکل قحطی‌زدگی برخی از فرآیندها، می‌توان از

khalilifar.ir

تکنیکی موسوم به سالخوردگی (Aging) استفاده کرد. در این تکنیک به تدریج اولویت پردازش‌هایی که مدت مدیدی در انتظار بوده‌اند، افزایش می‌یابد.
با توجه به مفروضات مطرح شده در صورت سؤال داریم:

پردازه	اولویت	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	3	0	10		
P_2	1	0	1		
P_3	4	0	2		
P_4	5	0	1		
P_5	2	0	5		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



توجه: مطابق مفروضات مطرح شده در صورت سوال، اولویت از نوع استاتیک و انحصاری است.

زمان ورود فرآیند – زمان خروج کامل فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 16 - 0 = 16$

P_2 زمان بازگشت $= 1 - 0 = 1$

P_3 زمان بازگشت $= 18 - 0 = 18$

P_4 زمان بازگشت $= 19 - 0 = 19$

P_5 زمان بازگشت $= 6 - 0 = 6$

میانگین زمان بازگشت $= \text{ATT} = \frac{16+1+18+19+6}{5} = \frac{60}{5} = 12$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 16 - 10 = 6$

P_2 زمان انتظار $= 1 - 1 = 0$

P_3 زمان انتظار $= 18 - 2 = 16$

khalilifar.ir

P_4 زمان انتظار $= 19 - 1 = 18$

P_5 زمان انتظار $= 6 - 5 = 1$

میانگین زمان انتظار $= AWT = \frac{6+0+16+18+1}{5} = \frac{41}{5} = 8.2$

میانگین زمان اجرا $= AST = \frac{10+1+2+1+5}{5} = \frac{19}{5} = 3.8$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

12= 3.8 + 8.2

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	اولویت	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	3	0	10	6	16
P_2	1	0	1	0	1
P_3	4	0	2	16	18
P_4	5	0	1	18	19
P_5	2	0	5	1	6

میانگین زمان بازگشت12 = میانگین زمان انتظار 8.2 + میانگین زمان اجرا 3.8

۸- گزینه (۲) صحیح است.

الگوریتم FIFO (First In First Out) ساده‌ترین الگوریتم از نظر پیاده‌سازی است. در این روش سیستم عامل لیستی از صفحات را به ترتیب ورود به حافظه نگه می‌دارد. وقتی یک خطای نقص صفحه رخ می‌دهد، سیستم عامل قدیمی‌ترین صفحه را برای بیرون رفتن انتخاب می‌کند. ایده این روش این است که قدیمی‌ترین صفحه شانس مورد استفاده قرار گرفتن را به اندازه کافی در اختیار داشته و اکنون باید این شانس به صفحه دیگری داده شود.

توجه: نقص الگوریتم FIFO این است که حتی اگر صفحه‌ای بارها و به طور مکرر استفاده شود، سرانجام به قدیمی‌ترین صفحه تبدیل و حذف می‌شود، در صورتی که احتمالاً بلافاصله باید دوباره به حافظه آورده شود.

khalilifar.ir

مثال: فرض کنید در سیستمی 3 قاب حافظه وجود دارد، اگر درخواست‌های زیر از چپ به راست، به این سیستم وارد شود، چند وقفه نقص صفحه رخ می‌دهد؟

4 ،3 ،2 ،1 ،5 ،4 ،3 ،5 ،4 ،3 ،2 ،1 ،5

حل: فرض می‌کنیم در ابتدا هر 3 قاب خالی هستند، با جدول زیر تعداد نقص صفحه را به دست می‌آوریم:

ورودی	4	3	2	1	4	3	5	4	3	2	1	5
قاب 1	4	4	4	1	1	1	5	5	5	5	5	5
قاب 2		3	3	3	4	4	4	4	4	2	2	2
قاب 3			2	2	2	3	3	3	3	3	1	1
وقفه خطای صفحه	×	×	×	×	×	×	×			×	×	

جمعاً 9 وقفه خطای صفحه رخ می‌دهد.

توجه: در نگاه اول به نظر می‌رسد با افزایش قاب‌هایی از حافظه که در اختیار یک فرآیند است، تعداد نقص صفحه‌ها همواره کاهش می‌یابد، اما در الگوریتم FIFO و در بعضی از الگوهای خاص ارجاع به صفحه‌ها، با افزایش تعداد قاب‌ها، تعداد وقفه‌های نقص صفحه نیز **افزایش** می‌یابد. این پدیده را **ناهنجاری FIFO (FIFO Anomaly) و یا ناهنجاری بی‌لیدی (Belady Anomaly)** گویند.

برای روشن شدن قضیه، همان مثال قبل را این‌بار با 4 قاب حافظه بررسی می‌کنیم:

ورودی	4	3	2	1	4	3	5	4	3	2	1	5
قاب 1	4	4	4	4	4	4	5	5	5	5	1	1
قاب 2		3	3	3	3	3	3	4	4	4	4	5
قاب 3			2	2	2	2	2	2	3	3	3	3
قاب 4				1	1	1	1	1	1	2	2	2
وقفه خطای صفحه	×	×	×	×			×	×	×	×	×	×

مشاهده می‌کنیم با همان دنباله ارجاع و با 4 قاب، تعداد نقص صفحه‌ای که رخ می‌دهد به 10 می‌رسد. البته اگر تعداد قاب‌ها را به 5 افزایش دهیم تعداد نقص صفحه در این مثال یکباره به 5 نقص صفحه کاهش می‌یابد.

۹- گزینه (۲) صحیح است.

در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

khalilifar.ir

$\sum_{i=1}^{n} Request[i] < m+n$: مجموع درخواست‌های فرآیندها برای منابع

$\rightarrow 3n < m+n \rightarrow 2n < m$

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد احتمال بن‌بست به صورت زیر است:

$n \times (r-1) = m$

حال اگر مقدار m حداقل یک واحد بیشتر از $n \times (r-1)$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

$n \times (r-1) < m$

$n \times r - n < m$

$n \times r < m+n$

$\sum_{i=1}^{n} Request[i] < m+n$

$\longrightarrow n \times 3 < 6+n \rightarrow 2n < 6 \rightarrow n < 3 \rightarrow n = 2$

۱۰- گزینه (۳) صحیح است.

معایب سیستم‌های لایه‌ای:

۱- کاهش کارایی

توجه: یک نقض سیستم‌های لایه‌ای این است که ممکن است قدری نسبت به سیستم‌های دیگر کند به نظر برسند و به تبع منجر به «**کاهش کارایی**» گردد، زیرا دستورات و فراخوانی‌ها از لایه

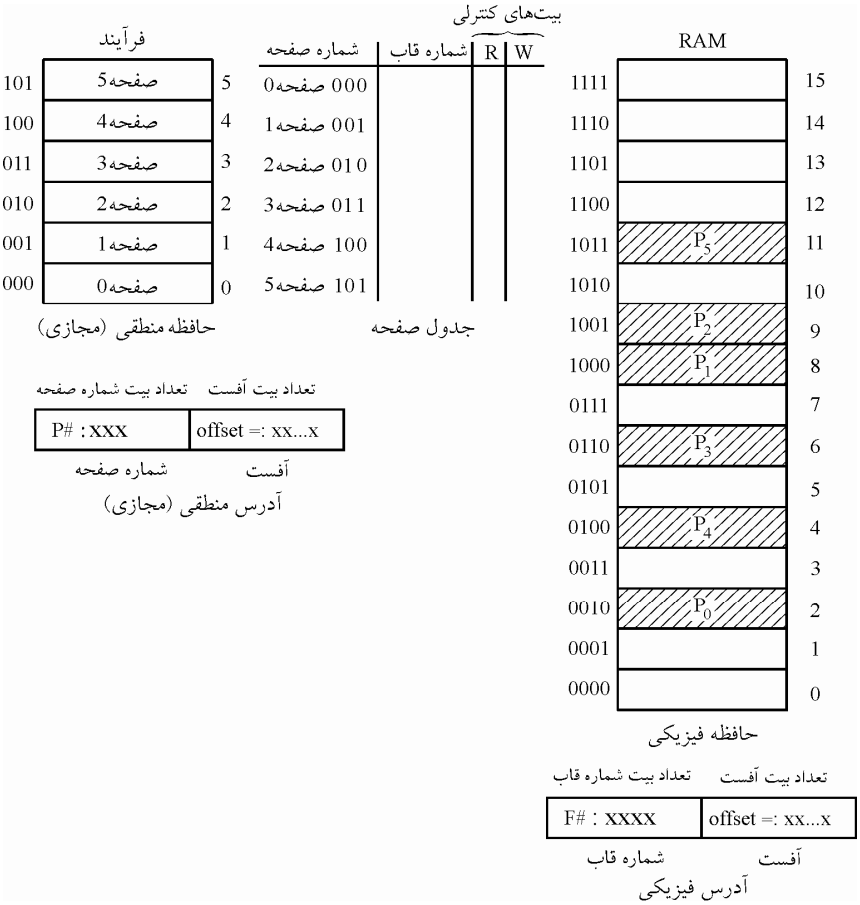
khalilifar.ir

بالا به سمت لایه پایین حرکت می‌کنند و زمان زیادی برای این روال صرف می‌شود. در واقع هر لایه قدری سربار (Overhead) به دستورات و فراخوانی‌ها می‌افزاید، در نتیجه یک فراخوانی سیستمی در این ساختار، زمان بیشتری نسبت به ساختار غیر لایه ای صرف می‌کند. برای رفع این نقص سعی می‌شود تعداد لایه‌های کمتری با قابلیت عمل بیشتری طراحی شود. به عنوان مثال محصول اولیه Windows NT با لایه‌های زیاد، کارایی کمتری نسبت به ویندوز 95 داشت. در NT 4.0 سعی شد لایه‌ها به همدیگر نزدیکتر و مجتمع‌تر شوند تا کارایی بیشتر گردد.

۲- نیاز به دقت بالا در پیمانه‌ای کردن و تقسیم‌بندی لایه‌ها دارد، مسأله اصلی در سیستم‌های لایه‌ای، تعریف مناسب هر لایه و سرویس‌های درون آن می‌باشد. از آنجا که هر لایه فقط می‌تواند از سرویس‌های لایه پایین‌تر استفاده کند، در طراحی لایه‌ها باید دقت بسیار به خرج داد.

۱۱- گزینه (۳) صحیح است.

توجه: مطابق اطلاعات مساله و **شکل حافظه اصلی صورت سوال**، شکل زیر گویای مطلب است:



khalilifar.ir

توجه: مطابق فرض سوال **قسمت‌های خاکستری** فریم‌های تخصیص داده شده به یک پردازه (فرآیند) هستند، بنابراین تعداد صفحات فرآیند برابر 6 عدد است.
توجه: در شکل صورت سوال 6 صفحه فرآیند (P_0 تا P_5) یا صفحه مجازی (Virtual Pages) در 6 قاب از 16 قاب حافظه اصلی یا حافظه فیزیکی (Physical Frames) به ترتیب در قاب‌های 2، 4، 6، 8، 9 و 11 درج شده است.

راه حل اول:

$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}}$$

$$\text{تعداد قاب‌های حافظه‌ی فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}}$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \lceil \log_2^6 \rceil = 3\text{bit}$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{16} = 4\text{bit}$$

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{2048} = 11\text{bit}$$

همچنین، اندازه آدرس‌های منطقی (مجازی) و فیزیکی به صورت زیر است:

$$\text{طول آدرس منطقی} = \text{تعداد بیت شماره صفحه} + \text{تعداد بیت آفست} = 3\text{bit} + 11\text{bit} = 14\,\text{bit}$$

$$\text{طول آدرس فیزیکی} = \text{تعداد بیت شماره قاب} + \text{تعداد بیت آفست} = 4\text{bit} + 11\text{bit} = 15\,\text{bit}$$

همچنین داریم:

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{\text{تعداد بیت آدرس منطقی}} = 2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{14} = 2^3 \times 2^{11} = 2^{14} = 16\,\text{KB}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{\text{تعداد بیت آدرس فیزیکی}} = 2^{\text{تعداد بیت شماره قاب}} \times 2^{\text{تعداد بیت آفست}}$$

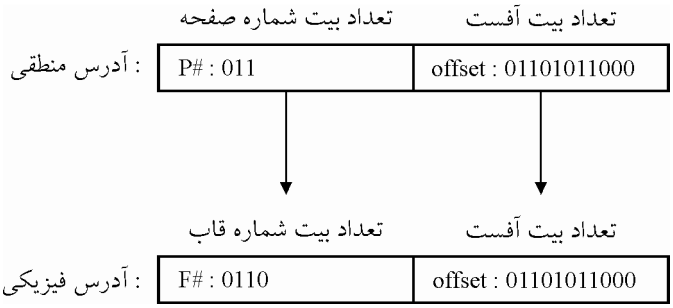
$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{15} = 2^4 \times 2^{11} = 2^{15} = 32\,\text{KB}$$

توجه: اندازه آدرس منطقی (مجازی) و فیزیکی (حقیقی) الزاماً برابر نیست.
توجه: آدرس منطقی (مجازی) توسط پردازنده (CPU) تولید می‌شود که مطابق صورت سوال، پردازنده آدرس 7000 را تولید کرده‌است، **باینری آدرس 7000 به صورت زیر است:**

8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	1	1	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**

khalilifar.ir

توجه: نگاشت آدرس منطقی (مجازی) به آدرس فیزیکی بر اساس مشخصات صورت سوال، به صورت زیر است:



توجه: در نمودار فوق، باینری 0110 به عنوان شماره قاب از آدرس فیزیکی، ارتباطی به آدرس منطقی 7000 و معادل باینری آن **01101011000**,011 ندارد، بلکه بر اساس شکل مطرح شده در صورت سوال واضح است که شماره صفحه 3 (P_3) و معادل باینری آن 011 در قاب شماره 6 و معادل باینری آن 0110 نگاشت حافظه‌ای شده‌است.

توجه: باینری آدرس 7000 برابر 14 بیت به عنوان طول کل آدرس منطقی به صورت **01101011000**,011 است. از آنجا که فرآیند موجود در این سیستم شامل 6 صفحه است، بنابراین 3 بیت سمت چپ $(b = \lceil \log_2^6 \rceil = 3\,bit)$ از آدرس منطقی 14 بیتی مربوط به شماره صفحه و 11 بیت سمت راست باقیمانده مربوط به آفست است، به صوت زیر:

شماره صفحه			آفست										
0	1	1	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**

باینری آدرس منطقی به صورت زیر است:

8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	1	1	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**

بر اساس اطلاعات جدول فوق دسیمال شماره صفحه از آدرس منطقی به صورت زیر است:

$$4096 + 2048 = (6144)_{10}$$

بر اساس اطلاعات جدول فوق دسیمال آفست از آدرس منطقی به صورت زیر است:

$$(01101011000)_2 = 512 + 256 + 64 + 16 + 8 = (856)_{10}$$

حاصل جمع دسیمال شماره صفحه و آفست برابر دسیمال آدرس منطقی است، به صورت زیر:

$$6144 + 856 = 7000$$

توجه: همچنین از آنجا که حافظه فیزیکی موجود در این سیستم نیز شامل 16 قاب است، بنابراین 4 بیت سمت چپ $(b = \log_2^{16} = 4\,bit)$ از آدرس فیزیکی نیز مربوط به تعداد بیت‌های شماره قاب

khalilifar.ir

می‌باشد. با توجه به اینکه در ترجمه آدرس منطقی به فیزیکی در سیستم‌های صفحه‌بندی شده، تعداد بیت‌های آفست ثابت مانده و فقط شماره صفحه با شماره قاب جای‌گزین می‌گردد، بنابراین آدرس فیزیکی نیز دارای طولی برابر 15 بیت خواهد بود، به صورت زیر:

شماره قاب				آفست										
0	1	1	0	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**

باینری آدرس فیزیکی به صورت زیر است:

16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	1	1	0	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**

بر اساس اطلاعات جدول فوق دسیمال شماره قاب از آدرس فیزیکی به صورت زیر است:

$8192 + 4096 = (12288)_{10}$

بر اساس اطلاعات جدول فوق دسیمال آفست از آدرس فیزیکی به صورت زیر است:

$(01101011000)_2 = 512 + 256 + 64 + 16 + 8 = (856)_{10}$

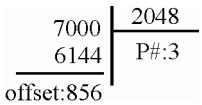
حاصل جمع دسیمال شماره قاب و آفست برابر دسیمال آدرس فیزیکی است، به صورت زیر:

$12288 + 856 = 13144$

بنابراین پُرواضح است که گزینه سوم پاسخ سوال است.

راه حل دوم:

ابتدا باید بررسی کنیم که آدرس منطقی 7000 در کدام صفحه از فرآیند قرار دارد، که به صورت زیر محاسبه می‌شود:



بر اساس شکل مطرح شده در صورت سوال واضح است که شماره صفحه 3 یعنی P_3 در قاب شماره 6 از حافظه اصلی (فیزیکی) قرار گرفته است. دقت کنید که قاب شماره 6 یعنی 7 اُمین قاب حافظه اصلی، بنابراین 6 قاب قبل از قاب شماره 6 (7 اُمین قاب) قرار دارد که باید از آن جهت رسیدن به آدرس فیزیکی مورد نظر عبور کنیم، به صورت زیر:

$\text{Physical address} = [6 \times 2048] + [\text{offset}] = [12288] + [856] = 13144$

بنابراین پُرواضح است که گزینه سوم پاسخ سوال است.

۱۲- گزینه (۴) صحیح است.

توجه: در این سوال تابع release معادل تابع signal در نظر گرفته شده است.

khalilifar.ir

با توجه به مقادیر شمارنده‌های سمافور مطرح شده در سوال، $S_0 = 1$، $S_1 = 0$ و $S_2 = 0$، فرآیند P_0 ابتدا اجرا می‌شود.

توجه: فرآیندهای P_1 و P_2 فاقد حلقه هستند، و توسط شرایطی که فرآیند P_0 برای فرآیندهای P_1 و P_2 فراهم می‌کند، فرآیندهای P_1 و P_2 فقط و فقط می‌توانند **یکبار** اجرا شوند و تمام شوند.

سناریوی اول: (دو بار چاپ '0')

ابتدا فرآیند P_0 اجرا می‌شود و روی شمارنده سمافور S_0، عمل $wait(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $release(S_1)$ و $release(S_2)$ مقدار شمارنده سمافورهای S_1 و S_2 را برابر یک می‌کند.

حال فرآیندهای P_1 و P_2 می‌توانند به هر ترتیبی (اول P_1 بعد P_2 و یا اول P_2 بعد P_1)، با توجه به مقدار شمارنده سمافورهای S_1 و S_2، اجرا شوند. ولی از آنجا که شمارنده سمافور دودویی است، در انتهای اجرای دو فرآیند P_1 و P_2، حداکثر مقدار شمارنده سمافور دودویی S_0 به واسطه دو بار اجرای دستور $release(s_0)$ برابر یک خواهد بود. اگر سمافور S عمومی بود، مقدار شمارنده سمافور S_0 برابر 2 می‌بود، اما فرض سوال سمافور دودویی است!

در این لحظه فرآیندهای P_1 و P_2 تمام شده‌اند و دیگر اجرا نمی‌شوند، از آنجا که در انتهای کار فرآیندهای P_1 و P_2 مقدار شمارنده سمافور S_0، برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یک بار دیگر اجرا شود و دومین '0' را نیز چاپ کند. در ادامه فرآیند P_0، با اجرای دستور $wait(S_0)$، با سرنوشتی که دارد، برای همیشه می‌خوابد.

دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ دومین '0'، تغییری در روند اجرای کار ندارند، چون فرآیندهای P_1 و P_2 دیگر نیستند و تمام شده‌اند.

سناریوی دوم: (سه بار چاپ '0')

ابتدا فرآیند P_0 اجرا می‌شود و روی شمارنده سمافور S_0، عمل $wait(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $release(S_1)$ و $release(S_2)$ مقدار شمارنده سمافورهای S_1 و S_2 را برابر یک می‌کند.

حال اگر فرآیند P_1، با توجه به مقدار شمارنده سمافور S_1 اجرا شود، در انتهای اجرای فرآیند P_1، مقدار شمارنده سمافور دودویی S_0 به واسطه $release(S_0)$ برابر یک خواهد شد. در این لحظه فرآیند P_1 تمام شده است و دیگر اجرا نمی‌شود. از آنجا که در انتهای کار فرآیند P_1، مقدار شمارنده سمافور S_0 برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یکبار دیگر اجرا شود و دومین '0' را چاپ کند. دقت کنید، اجرای دو دستور $release(S_1)$ و $release(S_2)$ پس از چاپ دومین '0'، تغییری در روند اجرای کار ندارد، چون فرآیند P_1، دیگر نیست و تمام شده است و برای فرآیند P_2 نیز مقدار شمارنده سمافور S_2 قبلاً و بعد از چاپ

khalilifar.ir

اولین '0' برابر یک شده است و چون شمارنده سمافور دودویی است، انجام دستور release(S_2) بعد از چاپ دومین '0' تأثیری در مقدار شمارنده سمافور S_2 نخواهد داشت و در همان مقدار یک باقی خواهد ماند.

توجه: دقت کنید که حداکثر مقدار شمارنده سمافور دودویی برابر یک می‌باشد.

حال اگر فرآیند P_2، با توجه به مقدار شمارنده سمافور S_2، اجرا شود، در انتهای اجرای فرآیند P_2، مقدار شمارنده سمافور دودویی S_0 به واسطه release(S_0) برابر یک خواهد شد. در این لحظه فرآیند P_2 تمام شده است و دیگر اجرا نمی‌شود. از آنجا که در انتهای کار فرآیند P_2 مقدار شمارنده سمافور S_0 برابر یک شد، بنابراین فرآیند P_0 مجدداً این شانس را خواهد داشت که یکبار دیگر اجرا شود و سومین '0' را چاپ کند. دقت کنید، اجرای دو دستور release(S_1) و release(S_2) پس از چاپ سومین '0'، تغییری در روند اجرای کار ندارد، چون فرآیندهای P_1 و P_2 تمام شده‌اند و دیگر اجرا نمی‌شوند.

در ادامه، فرآیند P_0، با اجرای دستور wait(S_0)، با سرنوشتی که دارد، برای همیشه می‌خوابد.

بنابراین گزینه چهارم درست خواهد بود.

۱۳- گزینه () صحیح است.

الگوریتم SJF (Shortest Job First)

در این روش ابتدا کاری برای اجرا انتخاب می‌شود که از همه کوتاهتر باشد (زمان اجرای کمتری داشته باشد).

توجه: این الگوریتم، **SPN** (Shortest Process Next) و

SPT (Shortest Process Time) نیز نامیده می‌شود.

توجه: SJF یک الگوریتم **انحصاری** (Non Preemptive) است. در سایر متون فارسی به الگوریتم انحصاری، الگوریتم «**غیرقبضه‌ای**» یا «**غیرقابل پس گرفتن**» یا «**غیرقابل تخلیه پیش هنگام**» نیز گفته می‌شود.

توجه: یک نقص عمده الگوریتمSJF این است که ممکن است باعث قحطی‌زدگی فرآیندهای طولانی شود. به این ترتیب که اگر همواره تعدادی فرآیند کوچک وارد سیستم شوند، اجرای فرآیندهای بزرگ به طور متناوب به تعویق می‌افتد. این روال حتی می‌تواند تا بینهایت ادامه یابد و هیچگاه نوبت به فرآیندهای بزرگ نرسد!!!!!

توجه: در این روش اگر دو فرآیند مدت زمان اجرای برابر داشته باشند، بر اساس FCFS زمان‌بندی می‌شوند.

توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسخ و میانگین زمان گردش کار (بازگشت) فرآیندهاست.

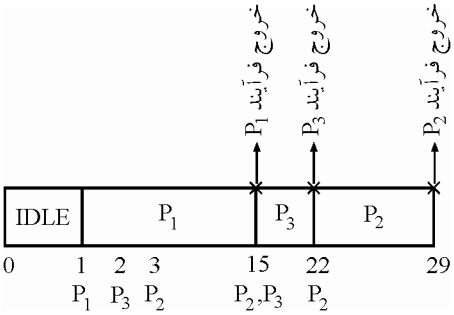
khalilifar.ir

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآیندها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجرای فرآیندها را فقط حدس زده و به طور تقریبی بدست آورد.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	1	14		
P_2	3	7		
P_3	2	7		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



توجه: در لحظه 15، زمان باقی‌مانده فرآیند P_2 و P_3 هر دو برابر 7 واحد زمانی است که به دلیل ورود زودتر فرآیند P_3 پردازنده به همان فرآیند P_3 اختصاص می‌یابد. که البته بهتر بود طراح محترم فرض این شرایط را بیان می‌کرد.

زمان ورود فرآیند – زمان خروج کامل فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 15 - 1 = 14$

P_2 زمان بازگشت $= 29 - 3 = 26$

P_3 زمان بازگشت $= 22 - 2 = 20$

میانگین زمان بازگشت $= ATT = \frac{14 + 26 + 20}{3} = \frac{60}{3} = 20$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 14 - 14 = 0$

P_2 زمان انتظار $= 26 - 7 = 19$

P_3 زمان انتظار $= 20 - 7 = 13$

khalilifar.ir

$$\text{میانگین زمان انتظار} = \text{AWT} = \frac{0+19+13}{3} = \frac{32}{3} = 10.67$$

$$\text{میانگین زمان اجرا} = \text{AST} = \frac{14+7+7}{3} = \frac{28}{3} = 9.33$$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

20 = 9.33 + 10.67

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	1	14	0	14
P_2	3	7	19	26
P_3	2	7	13	20

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

20 = 10.67 + 9.33

توجه: بیشینه زمان بازگشت یا چرخش (Turnaround Time) برابر 26 و مربوط به فرآیند P_2 است، همانطور که پُر واضح است، پاسخ سوال در گزینه‌ها موجود نیست.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه سوم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

۱۴- گزینه (۱) صحیح است.

ابتدا کد مطرح شده در صورت سوال را برای دو فرآیند P_0 و P_1 به صورت زیر بازنویسی می‌کنیم:

```
P0:
 {
(1) turn = 0;
(2) while (turn = = 1);
/ * critical section */
(3) turn = 1;
/ * reminder section */
}
```

```
P1:
 {
(1) turn = 1;
(2) while (turn = = 0);
/ * critical section */
(3) turn = 0;
/ * reminder section */
}
```

khalilifar.ir

توجه: متغیر turn در این مسئله به معنی متغیر نوبت نیست. هر گردی گردو نیست.
حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:
برای کنترل برقراری شرط انحصار متقابل، شرط پیشرفت و شرط انتظار محدود (گرسنگی و بن‌بست) از آزمون‌های زیر استفاده می‌کنیم:
توجه: ما نام این آزمون‌ها را به عنوان مبدع آن «**قوانین ارسطو**» نام‌گذاری کردیم، این قوانین به «**قوانین چهارگانه ارسطو**» نیز موسوم است.

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)
(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است.
(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:
فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
(1) turn = 0;

توجه: هم اکنون turn = 0 است.

(2) while (turn = = 1);

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:
/*critical section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون اول وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.
(گام ۲): فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، یعنی:
در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.
فرض کنید فرآیند P_1 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:
(1) turn = 1;

توجه: هم اکنون turn = 1 است.

(2) while (turn = = 0);

khalilifar.ir

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآینـد P_1 قرار می‌گیرد، به صورت زیر:

P_1:
/*critical section*/

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، خب موفق شد. فرآیند دوم توانست وارد ناحیه بحرانی خودش بشـود. بنـابراین شـرط اول انحصار متقابل برقرار نیست.

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

فرآیند اول و دوم را به طور همرونـد در سیسـتم تـک پردازنـده‌ای و یـا مـوازی در سیسـتم چنـد پردازنده‌ای حرکت بدید اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در ایـن حالت شرط انحصار متقابل نقض شده است.

P_0:
{
(1) turn = 0;
(2) while (turn = = 1);
/ * critical section */
(3) turn = 1;
/ * reminder section */
}

P_1:
{
(1) turn = 1;
(2) while (turn = = 0);
/ * critical section */
(3) turn = 0;
/ * reminder section */
}

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
(1) turn = 0;

توجه: هم اکنون turn = 0 است.

همچنین فرض کنید فرآیند P_1 نیز به شکل همروند یا موازی با فرآینـد P_0 قصـد دارد وارد ناحیـه بحرانی خودش شود، به صورت زیر:

P_1:
(1) turn = 1;

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارنـد، متغیـر turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآینـد، شـماره‌ی فرآینـد خود را در متغیر turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیره کند، فرآیندی است که شماره‌اش در متغیر turn باقی می‌مانـد و دیگـری اثـرش در متغیـر turn از بـین مـی‌رود. در واقـع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغییر turn گره خورده است، بنابراین در ایـن مسـئله

khalilifar.ir

فرآیندی که دیرتر متغیر turn را مقداردهی کرده است، وارد ناحیه بحرانی می‌شود. و فرآیندی که زودتر متغیر turn را مقداردهی کرده است، باید صبر پیشه کند و در حلقه‌ی انتظار بچرخد.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر turn کنند، بنابراین مقدار متغیر turn برابر با یک خواهد بود (turn=1)، وقتی که دو فرآیند به دستور while می‌رسند، خط **(2)** برای فرآیند P_0 برقرار است پس باید در یک حلقه انتظار مشغول، مشغول باشد. اما برای فرآیند P_1 برقرار نیست و وارد ناحیه بحرانی می‌شود. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

در ادامه پردازنده را از فرآیند P_1 بگیرید و به فرآیند P_0 بدهید.

P_0:
(2) while (turn = = 1);

توجه: هم اکنون turn = 1 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P_0 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P_0 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.

P_1:
(2) while (turn = = 0);

توجه: هم اکنون turn = 1 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P_1:
/*critical_section*/

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفق نشدند. فرآیند اول و دوم نتوانستند هر دو باهم وارد ناحیه بحرانی خودشان بشوند. بنابراین شرط دوم انحصار متقابل نیز برقرار است.

توجه: برای برقرار بودن شرط انحصار متقابل باید **قانون اول ارسطو** (آزمون اول شرط انحصار متقابل) و **قانون دوم ارسطو** (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشند. بنابراین شرط انحصار متقابل در سوال مطرح شده برقرار نیست.

khalilifar.ir

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۳)** در ادامه فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، **(گام ۴)** سپس همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۵)** در نهایت همان فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

```
P0:                                  P1:
 {                                    {
(1) turn = 0;                        (1) turn = 1;
(2) while (turn = = 1);              (2) while (turn = = 0);
/ * critical section */              / * critical section */
(3) turn = 1;                        (3) turn = 0;
/ * remainder section */             / * remainder section */
}                                    }
```

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
(1) turn = 0;
```

توجه: هم اکنون turn = 0 است.

```
(2) while (turn = = 1);
```

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

```
P0:
/*critical section*/
```

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

```
P0:
/*critical section*/
(3) turn = 1;
```

khalilifar.ir

حال در ادامه فرآیند P_0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P_0:
/*remainder_section*/

همانطور که در (گام ۲) گفتیم قرار شد که همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P_0 مشغول حرکت است.

(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.

فرض کنید فرآیند P_1 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:
(1) turn = 1;

توجه: هم اکنون turn = 1 است.

(2) while (turn = = 0);

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P_1:
/*critical section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند دوم را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۴) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_1 مشغول حرکت است.

(گام ۴): همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P_1 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P_1:
/*critical section*/
(3) turn = 0;

حال در ادامه فرآیند P_1 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P_1:
/*remainder_section

همانطور که در (گام ۴) گفتیم قرار شد که همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۵) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P_1 مشغول حرکت است.

khalilifar.ir

(گام ۵): فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست. یعنی:

فرض کنید فرآیند P_1 نیز قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) turn = 1;

توجه: هم اکنون turn = 1 است.

(2) while (turn = = 0);

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P_1:

/*critical section*/

همانطور که در (گام ۵) گفتیم قرار شد که فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، خب شد، فرآیند دوم مجددا وارد ناحیه بحرانی خودش شد. بنابراین شرط پیشرفت برقرار است.

قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، **(گام ۳)** در ادامه فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۴)** در نهایت همان فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.

P_0:
{
(1) turn = 0;
(2) while (turn = = 1);
/ * critical section */
(3) turn = 1;
/ * reminder section */
}

P_1:
{
(1) turn = 1;
(2) while (turn = = 0);
/ * critical section */
(3) turn = 0;
/ * reminder section */
}

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

(1) turn = 0;

khalilifar.ir

توجه: هم اکنون turn = 0 است.

(2) while (turn = = 1);

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:
/*critical section*/

(گام ۲): فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P_0 بگیرید و به فرآیند P_1 بدهید.

فرض کنید فرآیند P_1 نیز قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:
(1) turn = 1;

توجه: هم اکنون turn = 1 است.

(2) while (turn = = 0);

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P_1:
/*critical section*/

فرآیند دوم هم توانست وارد ناحیه بحرانی شود و دیگر نیاز به بررسی **(گام ۳)** و **(گام ۴)** هم نیست، بنابراین شرط انتظار محدود به دلیل نبود گرسنگی برقرار است.

قانون دوم ارسطو (آزمون بن‌بست)

جهت بررسی بن‌بست از همان **قانون دوم** استفاده می‌شود. در واقع روال بررسی همان قانون دوم است، اما نتیجه قانون متفاوت است.

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند که در این حالت «**بن‌بست**» رخ داده است.

همانطور که در **آزمون دوم** گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. خب همانطور که در آزمون دوم

khalilifar.ir

دیدید هر دو فرآیند باهم پشت ناحیه بحرانی خودشان مسدود نشدند. فرآیند اول نتوانست وارد ناحیه بحرانی خودش شود، اما فرآیند دوم توانست وارد ناحیه بحرانی خودش شود. بنابراین بن‌بست رخ نداده است.

توجه: برای برقرار بودن شرط انتظار محدود باید **قانون دوم ارسطو** (آزمون بن‌بست) و **قانون چهارم ارسطو** (آزمون گرسنگی) هر دو باهم برقرار باشند. بنابراین شرط انتظار محدود در سوال مطرح شده برقرار است.

توجه: پُر واضح است که گزینه چهارم پاسخ سوال است، زیرا هر سه شرط انحصار متقابل، انتظار محدود و پیشرفت همواره تضمین می‌شود.

khalilifar.ir

مشاوره تخصصی رشته کامپیوتر و IT

در راستای رسالت **مؤسسهٔ فرهنگی و انتشاراتی بابان** مبنی بر ارتقای سطح علـم و دانش کشور و کمک همه جانبه به دانشجویان و داوطلبان گرامـی، در جهـت قبـولی در کنکور کارشناسی ارشد و دکتری مهندسـی کـامپیوتر و IT دو طـرح زیـر را پایـه‌ریـزی کرده‌ایم:

۱) ارائه مشاوره تخصصی **حضوری** و **غیرحضوری** (تلفنی و آنلاین)

۲) برگزاری کلاس‌های **حضوری** و **غیرحضوری** (فیلم آموزشی و کلاس آنلاین)

برای آشنایی بیشتر با خدمات ارائـه شـده توسـط **موسسـه بابـان** بـه وب سـایت **khalilifar.ir** یا کانال تلگرام **@arastookhalilifar** مراجعه فرمایید.

تلفن دفتر مرکزی موسسه بابان: ۷۷۹۷۲۸۶۸-۰۲۱

تلفن دفتر فروشگاه انتشارات بابان: ۷۷۹۷۳۳۸۶-۰۲۱

پایگاه اطلاع رسانی موسسه بابان: www.baban.ir

khalilifar.ir



khalilifar.ir



khalilifar.ir



khalilifar.ir

بهترین منابع

کنکور ارشد فناوری اطلاعات

	عنوان درس	منابع
زبان عمومی و تخصصی	زبان انگلیسی	کلاس آنلاین یا فیلم استاد زارع یا کتاب راهیان ارشد
دروس مشترک	ریاضیات گسسته	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	ساختمان داده‌ها	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	طراحی الگوریتم	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	مهندسی نرم‌افزار	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
	شبکه‌های کامپیوتری	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
دروس تخصصی	پایگاه داده‌ها	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
	هوش مصنوعی	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	سیستم عامل	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
	اصول و مبانی مدیریت	مبانی مدیریت و مبانی رفتار سازمانی استیفن رابینز یا کتاب راهیان ارشد

گروه بابان
BABAN.IR

khalilifar.ir

بهترین منابع
کنکور ارشد علوم کامپیوتر

	عنوان درس	منابع
زبان عمومی و تخصصی	زبان انگلیسی	کلاس آنلاین یا فیلم استاد زارع یا کتاب راهیان ارشد
دروس پایه	ریاضی عمومی ۱ و ۲	کلاس آنلاین یا فیلم استاد گیلک یا جزوه بابان
	مبانی علوم ریاضی	کتاب لیپن ولیپن ترجمه عمید رسولیان فصل های اول، پنجم و هفتم کلاس آنلاین ریاضیات گسسته یا فیلم استاد گیلک یا کتاب ریاضیات گسسته راهیان ارشد
	مبانی ماتریس ها و جبرخطی	کتاب جبرخطی هافمن یا کتاب جبرخطی اونان یا کتاب جبرخطی پیام نور
	مبانی آنالیز ریاضی	اصول آنالیز ریاضی رودین فصل یک تا شش یا کتاب آنالیز ریاضی یک پیام نور
	مبانی آنالیز عددی	کتاب راهیان ارشد بهزاد خداکرمی یا کتاب آنالیز عددی بابلیان
	مبانی احتمال	کلاس آنلاین یا فیلم استاد گیلک یا جزوه بابان کتاب آمار و احتمال شلدون راس
دروس مشترک	ساختمان داده	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	طراحی الگوریتم	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	مبانی نظریه محاسبه	کلاس آنلاین یا فیلم استاد گیلک یا کتاب علوم رایانه استاد شاپوری
	مبانی منطق و نظریه مجموعه ها	کلاس آنلاین یا فیلم استاد گیلک یا جزوه بابان
	ریاضیات گسسته و مبانی ترکیبیات	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد

گروه بابان
BABAN.IR

khalilifar.ir