

موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد
درس و کنکور ارشد

سیستم عامل

(حل تشریحی سوالات دولتی ۱۴۰۰)

ویژهی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

آبراهام سیلبرشاتز، ویلیام استالینگر و اندور اس تنباوم

ارسطو خلیلی فر

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

تست‌های فصل هشتم: مدیریت ورودی و خروجی و دیسک

۱۰۱- کدام سطح از RAID را Disk mirroring می‌گویند؟ (مهندسی کامپیوتر - دولتی ۱۴۰۰)

۳ (۴)

۲ (۳)

۱ (۲)

۰ (۱)

عنوان کتاب: سیستم عامل

مؤلف: ارسسطو خلیلی‌فر

ناشر: انتشارات راهیان ارشد

آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل هفتم: مدیریت ورودی و خروجی

۱۰۱- 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 نوشتمن (read & write performance) یعنی افزایش سرعت خواندن و نوشتمن یا همان افزایش نرخ انتقال خواندن و نوشتمن، افزایش قابلیت اطمینان (reliability) یا همان افزایش تحمل پذیری 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 بوده است. 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 دریافت‌های 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 شده است که به طور مستقل و موازی عمل می‌کنند. با دیسک‌های چندگانه، در خواست‌های 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 در خواست 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 اضافه کردن 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 ساخت کند. خوشبختانه صنعت کامپیوتر، روی طرح استانداردی برای طراحی بانک اطلاعاتی دیسک‌های چندگانه توافق کرده است و این طرح به نام RAID خوانده می‌شود. طرح RAID شامل هفت سطح (۰ 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 مرتبی نیستند، بلکه مشخص کننده معماری‌های مختلفی هستند که در سه خصوصیت مشترکند:

خصوصیت اول: RAID مجموعه‌ای از گردنده‌های دیسک فیزیکی است که از نظر سیستم عامل به صورت یک واحد گردانده دیسک منطقی دیده می‌شود.

خصوصیت دوم: داده‌ها بر روی گرداننده‌های فیزیکی یک آرایه توزیع می‌شوند.

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. ۰ RAID خصوصیت سوم را پشتیبانی نمی‌کند.

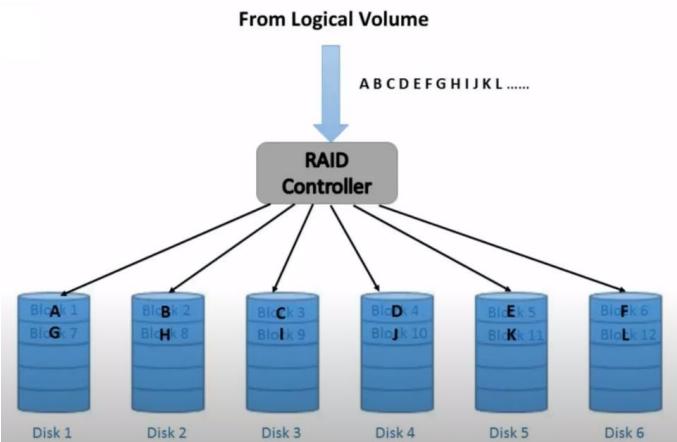
توجه: در تکنیک RIAD کارت کنترل کننده دیسک با یک کنترل کننده RAID جایگزین شده و داده در RAID کپی می‌شود و سپس عملیات عادی ادامه پیدا می‌کند.

واژه RAID ابتدا در مقاله‌ای توسط گروهی از پژوهشگران دانشگاه کالیفرنیا در برکلی به کار برده شد. در این مقاله سروازه 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 محرك‌ها، 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 $\frac{100,000}{100} = \frac{1000}{24}$ ساعت یا حدود 42 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 می‌شود، و این سرعت بالای گم شدن اطلاعات، غیر قابل قبول است. برای جبران این کاهش قابلیت اطمینان، RAID از داده‌های افزونه (اطلاعات کترلی) 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.

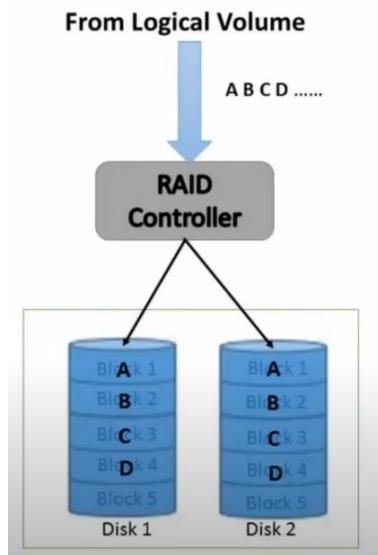
چهار تکنیک در RAID مورد استفاده قرار می‌گیرد:

تکنیک Striping: جهت افزایش کارایی خواندن و نوشتan (**read & write performance**) یعنی افزایش سرعت خواندن و نوشتan یا همان افزایش نرخ انتقال خواندن و نوشتan، مورد استفاده قرار می‌گیرد. (Striping improves performance) در این تکنیک یعنی data striping اصلی به بخش‌های مختلف 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 دقیقاً 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 نوشتan متشكل 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 نوشتan افزایش پیدا کند و به تبع آن زمان انتقال خواندن و نوشتan کاهش یابد. در این حالت یک درخواست خواندن و نوشتan، موجب انتقال موازی داده‌ها از دیسک‌های چندگانه شده است، که نرخ انتقال آن، در مقایسه با انتقال از یک دیسک، افزایش یافته است.

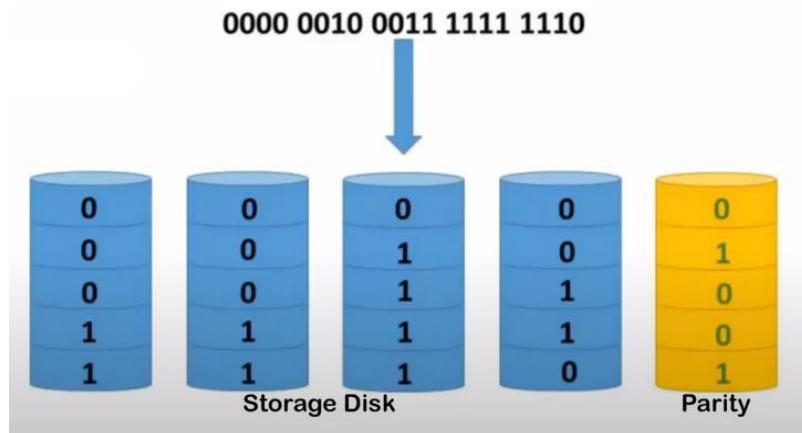


توجه: تکنیک Striping منجر به افزایش کارایی خواندن و نوشتمن (read & write performance) یعنی افزایش سرعت خواندن و نوشتمن یا همان افزایش نرخ انتقال خواندن و نوشتمن می‌شود. در چنین ساختاری، همه دیسک‌ها، در هر درخواست خواندن و نوشتمن مشارکت می‌کنند، بنابراین، تعداد درخواست‌هایی که می‌توانند در هر ثانیه پردازش 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 data striping به طور همزمان و موازی خوانده یا نوشته می‌شود. این تقسیم در data striping می‌تواند در سطوح byte, bit, block از یک فایل انجام شود. برای مثال در سطح bit اگر از یک آرایش 8 دیسکی برای فقط داده‌های اصلی استفاده شود، بیت ۰ از هر بایت به دیسک ۰ انتقال می‌شود.

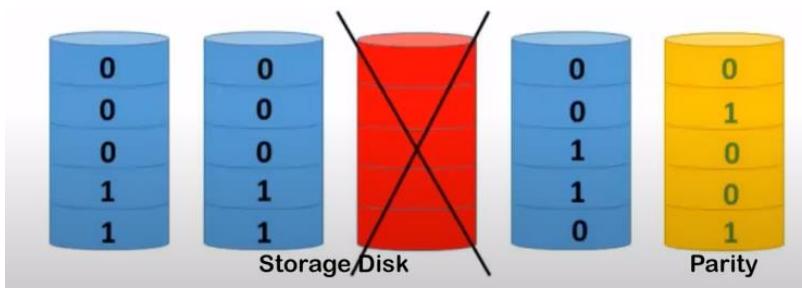
تکنیک Mirroring: جهت افزایش قابلیت اطمینان (reliability) مورد استفاده قرار می‌گیرد. در این تکنیک داده‌های اصلی روی دو دیسک، به طور یکسان نوشته می‌شوند. (mirrored set). در این تکنیک داده‌های افزونه از تکرار کاملاً یکسان از داده‌های اصلی ایجاد می‌شود. (Mirroring provides data redundancy) در نتیجه هر دیسک آرایه، دارای یک دیسک آینه و حاوی همان داده‌هاست.



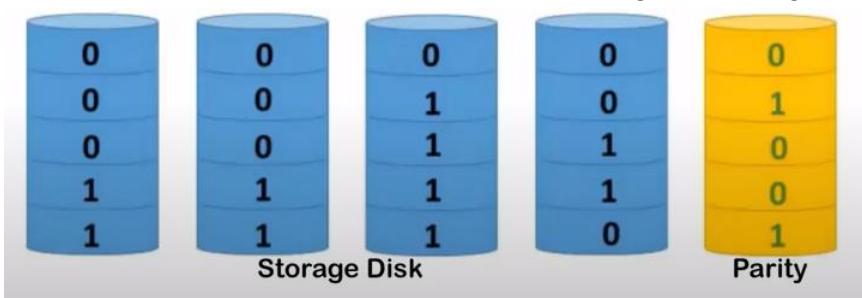
تکنیک Parity: جهت افزایش قابلیت اطمینان (reliability) مورد استفاده قرار می‌گیرد. در این تکنیک داده‌های افروزه از محاسبه توازن از داده‌های اصلی ایجاد می‌شود. در صورت استفاده از توازن زوج، بیت parity طوری مقداردهی می‌شود که تعداد 1 ها در کد، زوج بماند و در صورت استفاده از توازن فرد، بیت parity طوری مقداردهی 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 یک بیت 0 به 1 تبدیل شود، آنگاه توازن بیت‌های هم ردیف تعییر 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 ذخیره شده سازگار باشد، بیت از دست رفته و خراب شده، مقدار 0 داشته و در غیر اینصورت مقدار 1 داشته است.



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.

کد همینگ (Hamming code)

برای اینکه بتوان خطأ در هر داده اطلاعاتی را تشخیص داد و یک بیت خطأ 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 خطأ استفاده شود. برای اینکه بتوان در داده‌های اصلی خطأ را تشخیص داد باید برای هر m بیت داد

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	?	*	*	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 بیت‌هایی 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 برایر می‌شود با:

r_1	r_2	m_3	r_4	m_5	m_6	m_7
*	*	1	1	*	*	1

کنترل خطای کد همینگ

برای عمل کنترل خطای کد همینگ باید 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، خطای خود را باشد محل وقوع خطای می‌توان با معادل مقدار باینری $s_4s_2s_1$ بدست آورد. به عنوان مثال فرض کنید بیت ۵ خراب شود و از ۰ به ۱ دچار خطای شود، به صورت زیر:

r_1	r_2	m_3	r_4	m_5	m_6	m_7
۰	۰	۱	۱	۱	۰	۱

$$s_1 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$s_2 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

$$s_4 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$$

$$s_4s_2s_1 = (101)_2 = (5)_{10}$$

در نتیجه بیت شماره ۵ دچار خطای شده است و قابل تصحیح می‌باشد. و باید از ۱ به ۰ تصحیح شود، به صورت زیر:

r_1	r_2	m_3	r_4	m_5	m_6	m_7
۰	۰	۱	۱	۰	۰	۱

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 نوشتمن 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 نوشتمن در چند دیسک مختلف، جایگزین یک صف خواندن و نوشتمن در یک دیسک واحد می‌شود که به تبع آن افزایش سرعت دسترسی و افزایش نرخ انتقال را شاهد خواهیم بود.

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 عیناً 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 خواهد 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 خطای 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: معمولاً دیسک‌هایی که در RAID مورد استفاده قرار می‌گیرند، قابلیت تعویض فوری دارند، یعنی بدون قطع سیستم می‌توان آنها را تعویض کرد.

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 نوشتمن گفته می‌شود.

توجه: به میزان داده منتقل شده برای فقط یک درخواست جاری در واحد زمان، نرخ انتقال خواندن و نوشتمن گفته می‌شود.

نرخ درخواست خواندن و نوشتمن

توجه: هرچه قدر اندازه Striping و تقسیم داده‌های اصلی بزرگتر (block-level Striping) باشد، نرخ (تعداد) درخواست‌های خواندن و نوشتمن 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 نوشتمن اطلاعات بعدی با تاخیر سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های جداگانه و مستقل دیگر خواندن و نوشتمن به طور موازی از سایر دیسک‌ها امکان‌پذیر خواهد بود، که منجر به افزایش نرخ درخواست خواندن و نوشتمن در دیسک‌های مختلف می‌شود.

توجه: هرچه قدر اندازه Striping و تقسیم داده‌های اصلی کوچکتر (bit , byte-level Striping) باشد، نرخ (تعداد) درخواست‌های خواندن و نوشتمن مختلف جداگانه و مستقل موازی کمتر می‌شود، چون اندازه یک bit 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 نوشتمن اطلاعات بعدی فوراً سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های جداگانه و مستقل دیگر خواندن و نوشتمن به طور موازی از سایر دیسک‌ها امکان‌پذیر خواهد بود، چون همه دیسک‌ها به طور ترتیبی جهت خواندن و نوشتمن، مشغول پاسخ به درخواست جاری هستند و امکان پاسخ به درخواست جدید تا پایان درخواست جاری وجود ندارد، که منجر به کاهش نرخ درخواست خواندن و نوشتمن در دیسک‌های مختلف می‌شود.

توجه: البته دقت نمایید که نرخ درخواست نوشتمن 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 نوشتمن

توجه: هرچه قدر اندازه Striping و تقسیم داده‌های اصلی بزرگتر (block-level Striping) باشد، نرخ انتقال خواندن و نوشتمن 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 نوشتمن اطلاعات بعدی با تاخیر سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های دیگر خواندن و نوشتمن به طور موازی از

سایر دیسک‌ها امکان‌پذیر خواهد بود، که منجر به کاهش نرخ انتقال خواندن و نوشتمن 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 نوشتمن برای فقط یک درخواست جاری بیشتر می‌شود، چون اندازه یک bit 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 نوشتمن اطلاعات بعدی فررا سراغ دیسک بعدی می‌رود، بنابراین پاسخ به درخواست‌های دیگر خواندن و نوشتمن به طور موازی از سایر دیسک‌ها امکان‌پذیر خواهد بود، چون همه دیسک‌ها به طور ترتیبی جهت خواندن و نوشتمن، مشغول پاسخ به درخواست جاری هستند و امکان پاسخ به درخواست جدید تا پایان درخواست جاری وجود ندارد، که منجر به افزایش نرخ انتقال خواندن و نوشتمن 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 نوشتمن 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 نوشتمن بالا را به نرخ انتقال خواندن و نوشتمن بالا ترجیح می‌دهد. کارآمدی افزایش نرخ درخواست خواندن و نوشتمن تحت تاثیر اندازه Striping یا باریکه‌سازی و قطعه‌سازی است. اگر اندازه باریکه بزرگ باشد، به طوری که یک درخواست خواندن و نوشتمن فقط شامل یک دسترسی به دیسک باشد، درخواست‌های خواندن و نوشتمن بعدی می‌توانند به صورت موازی و همزمان انجام گیرند و به این ترتیب زمان صفحه‌بندی هر درخواست را کاهش دهد.

0 سطح RAID

در RAID سطح 0 ، تکنیک Block-level striping without parity or mirroring مورد استفاده قرار می‌گیرد، در نتیجه افزایش کارایی خواندن (n: read performance) و افزایش کارایی نوشتمن (n: write performance) برای «داده‌های اصلی» محقق می‌شود. یعنی افزایش سرعت خواندن و نوشتمن بالایی دارد. البته در صورتی که اندازه بلوك 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 نوشتمن در تعداد دیسک‌ها ضرب می‌شود، زیرا خواندن و نوشتمن به طور همزمان و موازی انجام می‌شود.

همچنین هیچکدام از تکنیک‌های افزونگی (redundancy) مثل Hamming ، Mirroring و Parity مورد استفاده قرار نمی‌گیرد، در نتیجه افزایش قابلیت اطمینان محقق نمی‌شود.

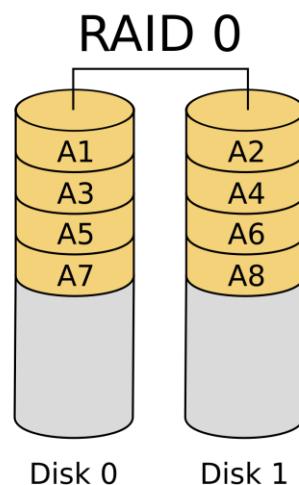
توجه: در این سطح، داده‌های افزونه (redundancy) روی دیسک‌های مختلف توزیع (distributed) نمی‌شود. چون اصلاً برای این سطح داده افزونه تعریف نشده و وجود ندارد.

توجه: اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد، چون اصلاً برای این سطح داده افزونه تعریف نشده و وجود ندارد. به عبارت دیگر تحمل‌پذیری در برابر خطای را ندارد (fault tolerance: None). از آنجا که striping محتوای داده‌های اصلی را بین همه دیسک‌های موجود توزیع می‌کند، 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.

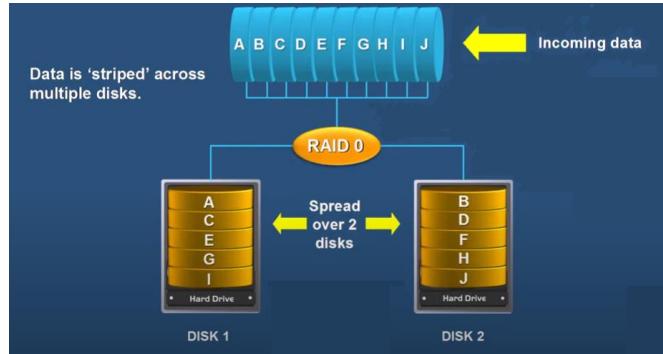
توجه: این سطح، کمترین میزان افزونگی را ایجاد می‌کند، در نتیجه کمترین میزان قابلیت اطمینان را نیز ایجاد می‌کند. 100 درصد کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی و 0 درصد دیگر فضای ذخیره‌سازی برای نگهداری داده‌های افزونه (پشتیبان اطلاعات) مورد استفاده قرار می‌گیرد. یعنی اگر فضای ذخیره‌سازی دیسک اول برابر 50 GB 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)

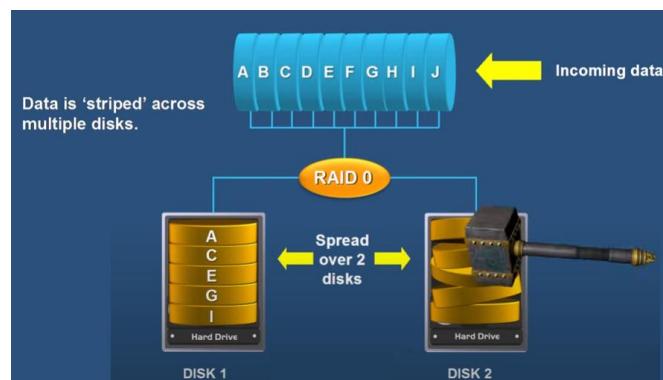
توجه: این سطح، در حداقل 2 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 2)



شکل زیر گویای عملکرد RAID سطح 0 است:



اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد، چون اصلاً برای این سطح داده افزونه تعریف نشده و وجود ندارد. شکل زیر گویای مطلب است:



پُر واضح 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 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 نوشتمن 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 صفحه‌بندی کاهش یابد.

سطح RAID 1

در RAID سطح 1، تکنیک Mirroring without parity or striping مورد استفاده قرار می‌گیرد، در نتیجه افزایش کارایی خواندن (1) و افزایش کارایی نوشتمن (write performance) 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 حداقل 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 1 RAID استفاده شود، عملکرد کلی نوشتمن برابر با سرعت کندترین دیسک است، برای درخواست نوشتمن هر دو دیسک باید بهنگام شوند. در 1 RAID هیچگونه «جريمه نوشتمن» به معنی محاسبات داده‌های افزونه وجود ندارد. اما سطوح 2 تا 6 شامل استفاده از بیت‌های توازن است و به تبع «جريمه نوشتمن» به معنی محاسبات داده‌های افزونه را دارند. در نتیجه، در بهنگام کردن و نوشتمن 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.

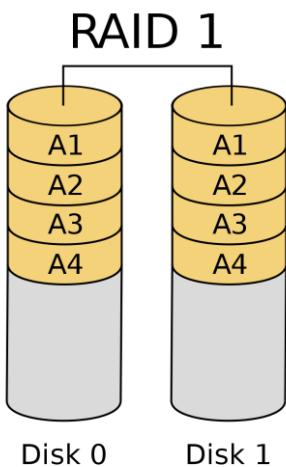
توجه: اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط فقط داده‌های افزونه (fault tolerance: mirrored set) وجود دارد.

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 فقط 50 GB فضای ذخیره‌سازی یکی از دیسک‌ها برای ذخیره داده‌های اصلی مورد استفاده قرار

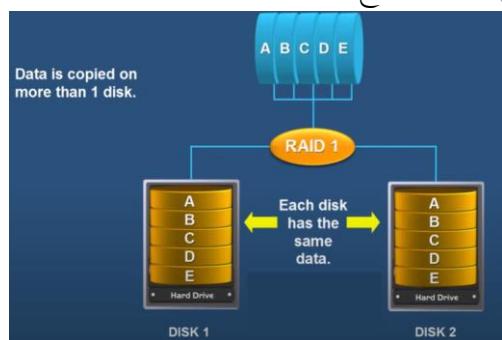
$$\text{می‌گیرد. (space efficiency: } \frac{1}{n} = 50\% \text{)}$$

توجه: این سطح، خرابی (n-1) دیسک را پشتیبانی می‌کند.

توجه: این سطح، در حداقل 2 دیسک و بیشتر کاربرد دارد.

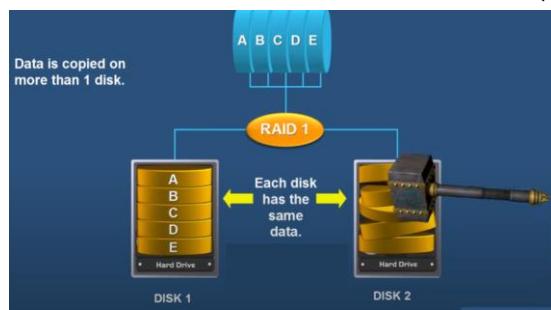


شكل زیر گویای عملکرد RAID سطح 1 است:



اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود دارد،

شكل زیر گویای مطلب است:



پُر واضح 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.

2 سطح RAID

در RAID 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 دو بیتی را فقط کشف نماید.

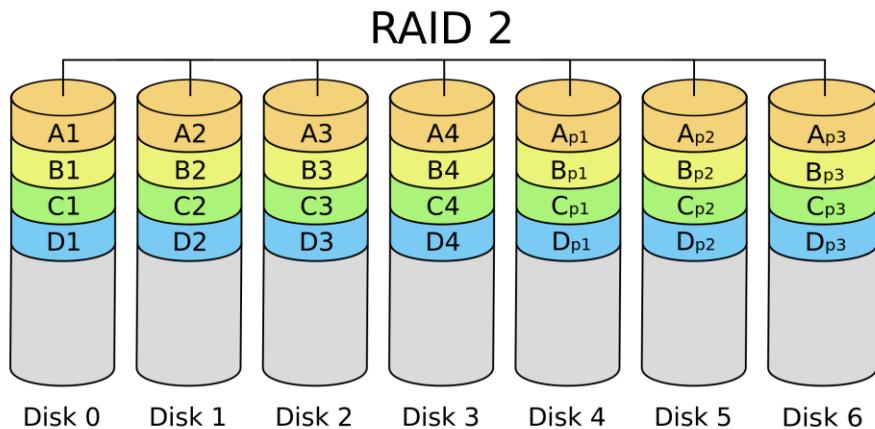
توجه: در کد Hamming یک دیسک خراب معادل یک بیت خراب است، در نتیجه اگر یکی از دیسک‌ها خراب شود و نیاز به تعویض با یک دیسک جدید داشته باشد، امکان بازیابی اطلاعات اصلی توسط داده‌های اصلی و افزونه (کد Hamming) وجود دارد. به عبارت دیگر با خراب شدن یک دیسک، دیسک‌های کد Hamming دسترسی شده و داده‌های از دست رفته 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 در صدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و در صدی دیگر از فضای ذخیره‌سازی برای نگهداری داده‌های افزونه (کد Hamming) یا پشتیبان اطلاعات، مورد استفاده قرار می‌گیرد. (Hamming 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)



توجه: در افزونگی RAID 2 و استفاده از کد Hamming زیاده روی شده است. در 2 RAID تعداد دیسک های افزونه متناسب با الگاریتم تعداد دیسک های داده های اصلی است.

توجه: سطح 2 RAID به طور عملی پیاده سازی نشده است و مورد استفاده قرار نمی گیرد.

توجه: در 2 RAID، تکنیک bit-level Striping مورد استفاده قرار می گیرد و Strping و تقسیم داده های اصلی کوچک و در سطح بیت است، در نتیجه نرخ (تعداد) درخواست های 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 2 RAID در یک خواندن واحد، همزمان تمام دیسک ها مورد دسترسی قرار می گیرند. داده های درخواست شده و کد تصحیح خطای مربوطه به کنترل کننده آرایه فرستاده می شود. اگر یک خطای تک بیتی موجود باشد، کنترل کننده می تواند بی درنگ آنرا تشخیص داده و تصحیح نماید، در نتیجه زمان دسترسی برای خواندن کند نمی شود. برای یک نوشتن واحد نیز، باید تمام دیسک های داده اصلی و دیسک های داده افزونه مورد دسترسی قرار گیرند.

3 سطح RAID

در RAID سطح 3، تکنیک Byte-level striping with dedicated parity مورد استفاده قرار می گیرد، در نتیجه افزایش کارایی خواندن (n-1) read performance: و افزایش کارایی نوشتن (write performance: n-1) برای «داده های اصلی» تا حدی محقق می شود. البته در بخش کارایی نوشتن، کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشتن) وجود دارد، بنابراین سخت افزاری را فرض کنید که می تواند محاسبات مرتبط با بیت توازن را با سرعت کافی انجام دهد.

RAID 3 نیز شبیه RAID 2 عمل می کند، ولی از افزونگی کمتری استفاده می کند و فقط از یک تک بیت افزونه (parity) به عبارت دیگر یک تک بیت توازن منفرد برای بازیابی اطلاعات متناظر استفاده می کند. یعنی فقط از یک دیسک اضافی برای ذخیره سازی تک بیت افزونه بهره می گیرد

که نسبت به ۲ RAID ارزان‌تر است. در حالی که در ۲ RAID تعداد دیسک‌های افزونه به دلیل استفاده از کد Hamming، متناسب با لگاریتم تعداد دیسک‌های داده‌های اصلی بود. بنابراین در کاربرد و عمل ۲ RAID مورد استفاده قرار نمی‌گیرد.

توجه: البته دقیق‌تر 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 آم 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.

توجه: در این سطح، داده افزونه (Parity) روی دیسک‌های مختلف توزیع (distributed) 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 ۱ آم 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 3 RAID تا 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.

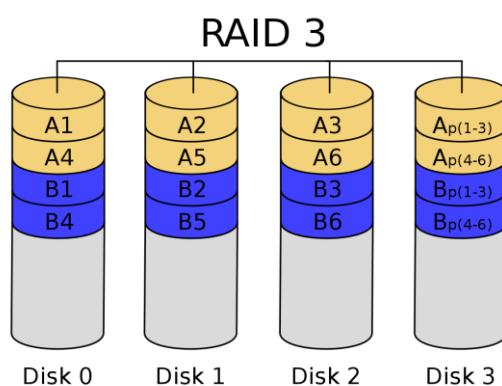
توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Parity در صدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های

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: $1 - \frac{1}{n}$)

(fault tolerance: one drive failure) توجه: این سطح، خرابی یک دیسک را پشتیبانی می‌کند.

(minimum number of drives: 3) توجه: این سطح، در حداقل 3 دیسک و بیشتر کاربرد دارد.



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 Striping 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.

4 سطح RAID

در RAID 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 ام است.

توجه: در این سطح داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود. همچنین تکنیک Dedicated Parity به معنی 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.

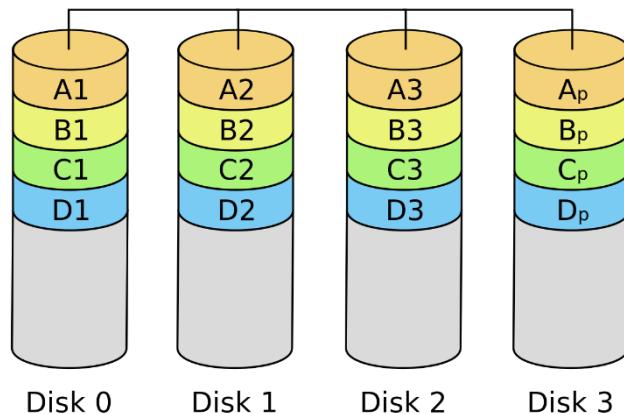
توجه: بازسازی داده‌ها ساده است. و همانند 3 RAID با استفاده از عملگر XOR انجام می‌گردد. توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد در صدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و در صدی دیگر از فضای ذخیره‌سازی معادل یک تک دیسک برای نگهداری داده‌های افزونه (parity) یا پشتیبان اطلاعات شامل مجموع تک بیت‌های افزونه (معادل یک بلوک)،

مورد استفاده قرار می‌گیرد. ($\text{space efficiency: } 1 - \frac{1}{n}$).

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)

RAID 4



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.

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 Striping و تقسیم داده‌های اصلی بزرگ و در سطح بلوک است، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتمن مختلف جداگانه و مستقل موازی زیاد است. اما نرخ انتقال خواندن و نوشتمن برای فقط یک درخواست جاری متوسط است.

توجه: البته دقت نمایید که نرخ انتقال نوشتمن 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 نوشتمن 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 نوشتمن، فقط به یک دیسک دستیابی دارد و به درخواست‌های دیگر امکان می‌دهد تا به وسیله دیسک‌های دیگر پردازش شوند. نرخ انتقال خواندن و نوشتمن در هر دستیابی پایین است چون از قدرت نرخ انتقال همه دیسک‌ها به طور موازی استفاده نمی‌شود، اما چندین دستیابی خواندن و نوشتمن می‌تواند به صورت موازی صورت بگیرد که منجر به افزایش نرخ درخواست خواندن و نوشتمن می‌شود. در آرایه‌ای با دسترسی مستقل، هر دیسک عضو به طور مستقل عمل می‌کند. پس می‌توان درخواست‌های خواندن و نوشتمن مجزا را به صورت موازی برآورده کرد. به همین دلیل، آرایه‌ای با

دسترسی مستقل، بیشتر برای کاربردهای با نرخ بالای درخواست خواندن و نوشتן (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 بلوك 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 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.

5 سطح RAID

در RAID 5، تکنیک Block-level striping with distributed parity مورد استفاده قرار می‌گیرد، در نتیجه افزایش کارایی خواندن (read performance: n) و افزایش کارایی نوشتan (write performance: n-1) در یک full stripe یعنی نوار کامل برای «داده‌های اصلی» تا حدی محقق می‌شود. البته در بخش کارایی نوشتan، 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.

5 RAID 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 بلوك افزونه، نتیجه تناظر و توازن بیت به بیت در بلوك‌های داده‌های اصلی است. برای مثال یک بلوك داده افزونه، نتیجه تناظر و توازن بیت به بیت در بلوك‌های داده‌های اصلی یعنی بلوك اول دیسک اول، بلوك اول دیسک دوم و بلوك اول دیسک n ام 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.

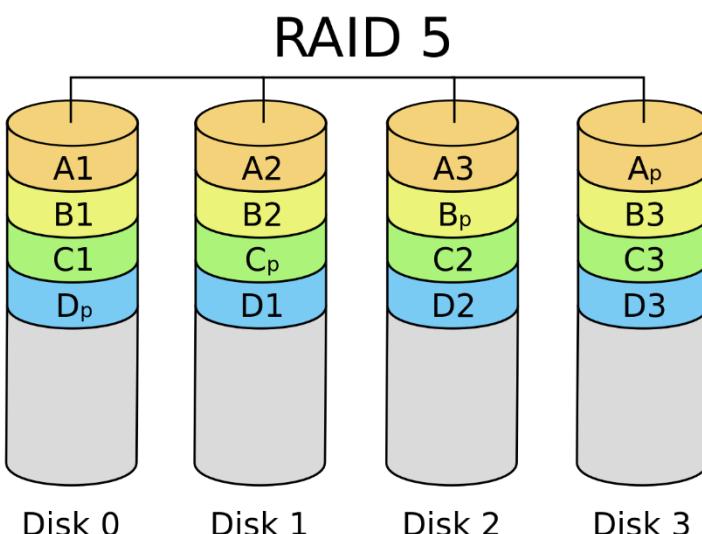
توجه: بازسازی داده‌ها ساده است. و همانند RAID 4 با استفاده از عملگر XOR انجام می‌گردد.

توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Parity در صدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و در صدی 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. (space efficiency: $1 - \frac{1}{n}$)

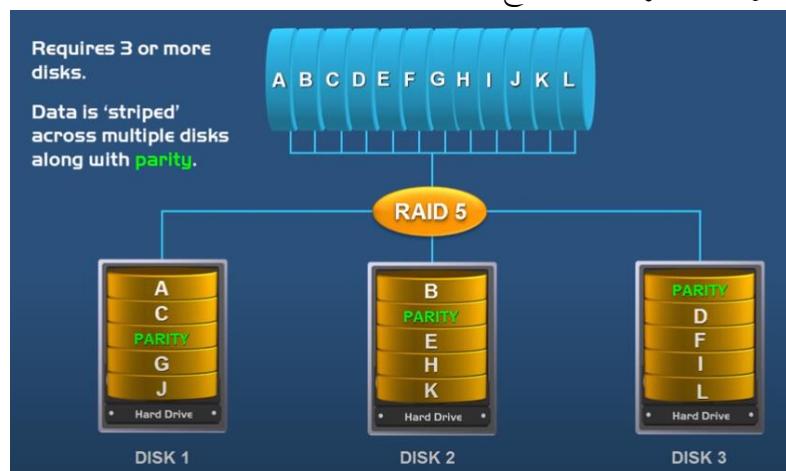
توجه: این سطح خرابی یک دیسک را پشتیانی می‌کند. (fault tolerance: one drive failure)

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 ۱ دیسک برای ذخیره‌سازی تک بیت داده افزونه (بلوک افزونه) به عنوان پشتیبان اطلاعات مورد استفاده قرار می‌گیرد.

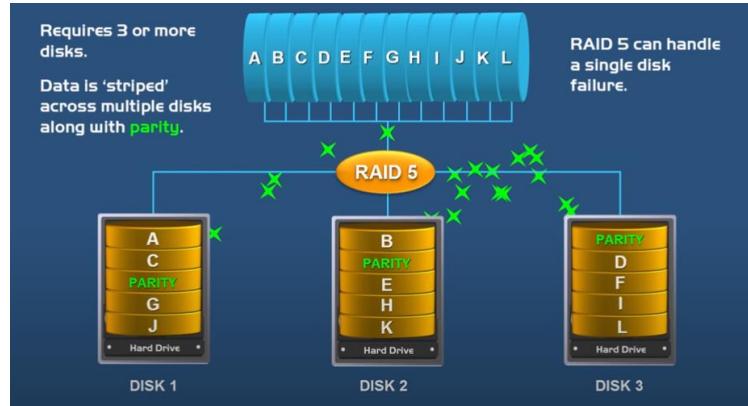
شکل زیر گویای عملکرد RAID سطح ۵ است:



اگر یکی از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود دارد:



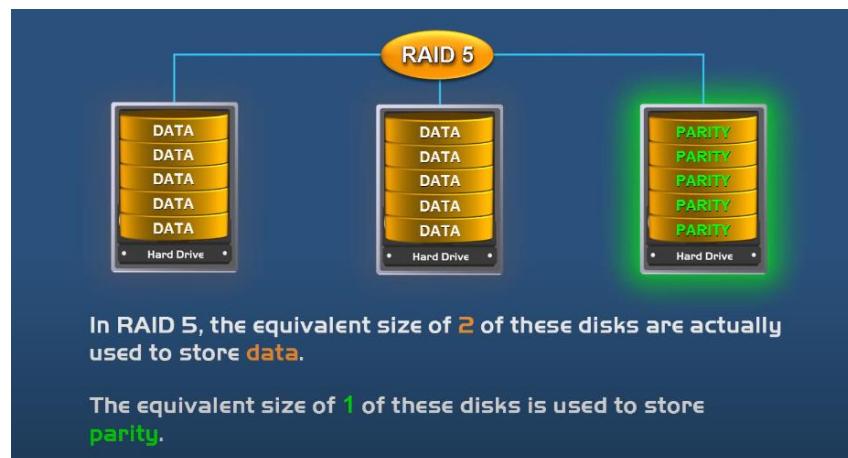
شکل زیر گویای مطلب است:



اگر دو عدد از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد، شکل زیر گویای مطلب است:



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 نوشتمن مختلف جداگانه و مستقل موازی زیاد است. اما نرخ انتقال خواندن و نوشتمن برای فقط یک درخواست جاری متوسط است.

توجه: البته دقت نمایید که نرخ انتقال نوشتمن 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 نوشتمن 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 نوشتمن در یک درخواست واحد بیشتر از یک بلوک باشد، آنگاه نرخ انتقال خواندن و نوشتمن 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 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 نوشتمن) وجود دارد، از آنجاکه هر عمل نوشتمن باید با دیسک توازن درگیر باشد، پس سرعت سخت‌افزاری که محاسبات مرتبط با

بیت توازن را انجام می‌دهد می‌تواند «گلوگاه» ایجاد کند. در 5 RAID 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.

6 سطح RAID

در RAID 6، تکنیک Block-level striping with double distributed parity مورد استفاده قرار می‌گیرد، در نتیجه افزایش کارایی خواندن (read performance: n) و افزایش کارایی نوشت (write performance: n-2) در یک full stripe (write performance: n-2) یعنی نوار کامل برای «داده‌های اصلی» تا حدی محقق می‌شود. البته در بخش کارایی نوشت، کمی کندی حاصل از محاسبات بیت توازن (جریمه نوشت) 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 متمايز 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 6 RAID که کاربر آن نیازمند N دیسک داده اصلی باشد از $N+2$ دیسک تشکیل شده است. وقت کنید که RAID 6 جریمه قابل توجهی برای نوشت دارد، زیرا هر نوشت، بر روی 2 بلوک توازن تاثیر می‌گذارد. به عبارت دیگر محاسبات دو برابری بیت توازن، به تبع جریمه نوشت دو برابری هم در پی خواهد داشت.

توجه: در این سطح داده‌های اصلی روی دیسک‌های مختلف توزیع (distributed) می‌شود.

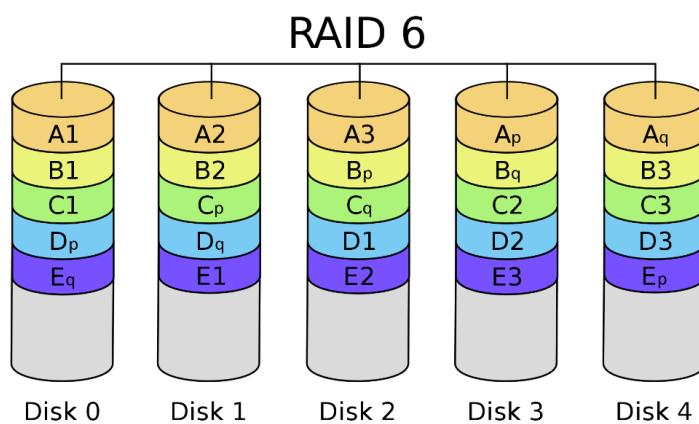
همچنین تکنیک Double Distributed Parity به معنی **Parity** توزیع شده دوباره مورد استفاده قرار می‌گیرد، در نتیجه افزایش قابلیت اطمینان محقق می‌شود.

توجه: بازسازی داده‌ها ساده است. توازن اول همانند RAID 5 با استفاده از عملگر XOR و توازن دوم توسط الگوریتم متمايز دیگری انجام می‌گردد.

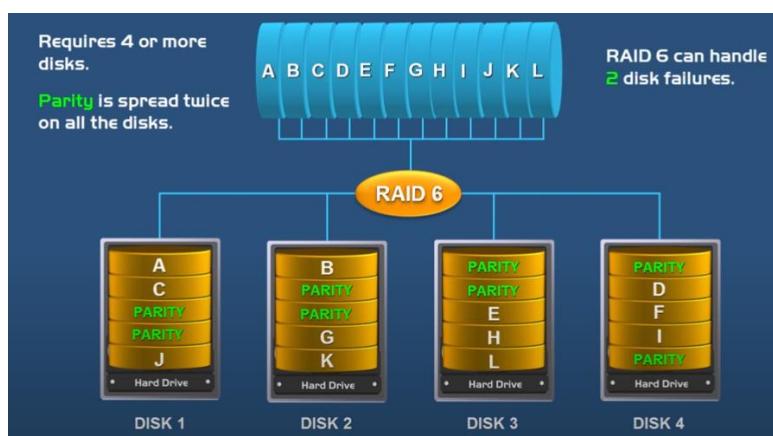
توجه: این سطح، افزونگی ایجاد می‌کند، در نتیجه تا حدی قابلیت اطمینان را نیز ایجاد می‌کند. بسته به تعداد بیت‌های کد Parity درصدی از کل فضای ذخیره‌سازی برای نگهداری داده‌های اصلی (data) و درصدی دیگر از فضای ذخیره‌سازی معادل دو تک دیسک برای نگهداری داده‌های افزونه (parity) یا پشتیبان اطلاعات شامل مجموع تک بیت‌های افزونه (معادل دو بلوک مستقل)،

$$\text{موردن استفاده قرار می‌گیرد. (space efficiency: } 1 - \frac{2}{n} \text{)}$$

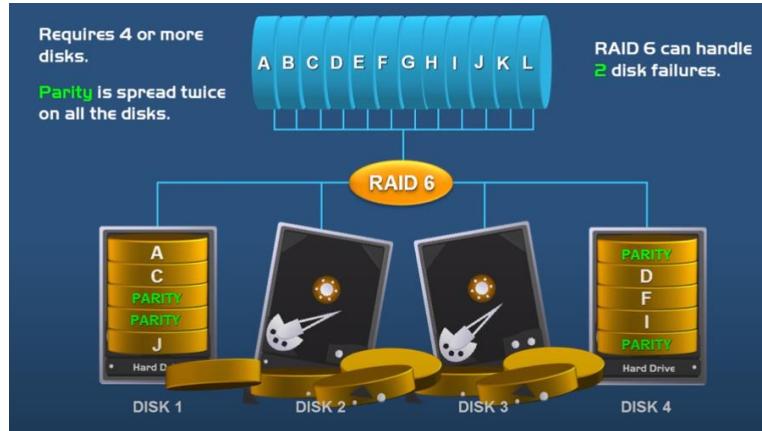
توجه: این سطح، خرابی دو دیسک را پشتیبانی می‌کند.
(fault tolerance: two drive failure)
minimum number of drives: 4



همانطور که در تصویر فوق واضح است، $n-2$ دیسک برای ذخیره‌سازی داده‌های اصلی و 2 دیسک برای ذخیره‌سازی دو تک بیت داده افزونه (دو بلوک افزونه) به عنوان پشتیبان اطلاعات مورد استفاده قرار می‌گیرد.
شکل زیر گویای عملکرد RAID 6 سطح 6 است:



اگر دو عدد از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود دارد:

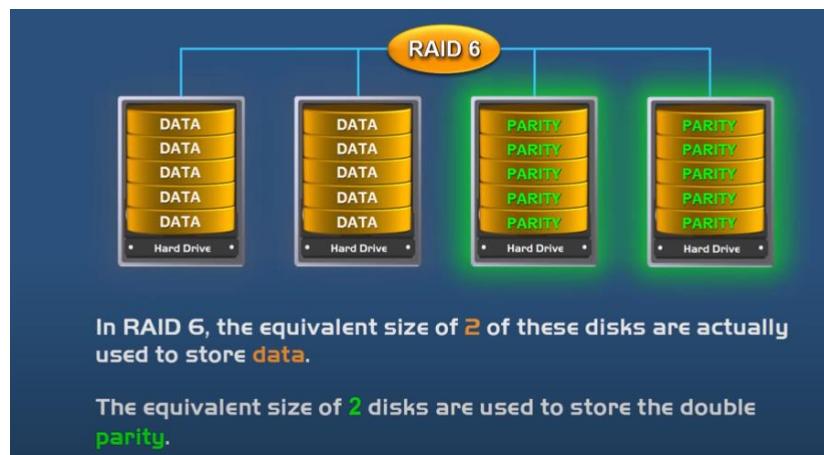


شکل زیر گویای مطلب است:

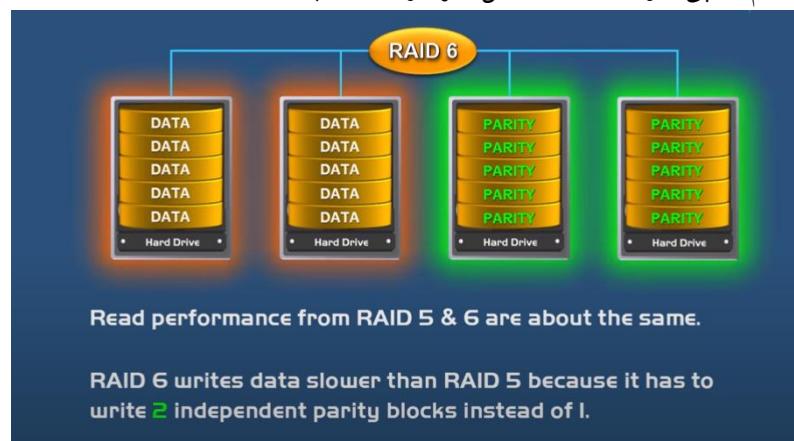


اگر سه عدد و بیشتر از دیسک‌ها خراب شود، امکان بازیابی اطلاعات اصلی توسط داده‌های افزونه وجود ندارد.

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:



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:



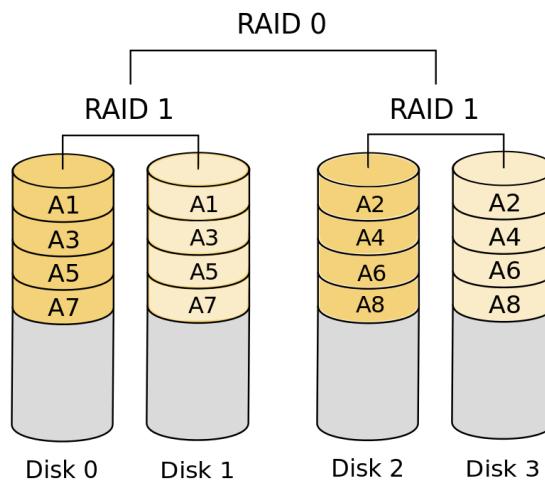
توجه: در RAID 6، تکنیک block-level Striping مورد استفاده قرار می‌گیرد و تقسیم داده‌های اصلی بزرگ و در سطح بلوک است، در نتیجه نرخ (تعداد) درخواست‌های خواندن و نوشتن مختلف جداگانه و مستقل موازی زیاد است. اما نرخ انتقال خواندن و نوشتن برای فقط یک درخواست جاری متوسط است.

توجه: سایر مشخصات RAID 6 همانند RAID 5 است.

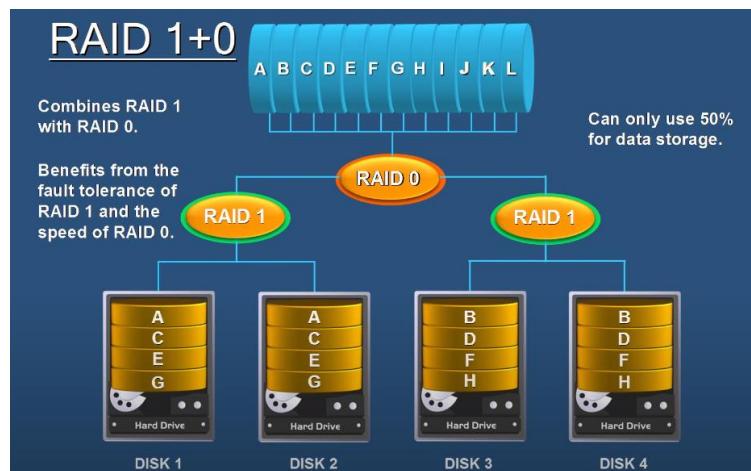
10 سطح RAID

در سطح RAID 10 ، تکنیک Mirroring without parity، and block-level striping مورد استفاده قرار می‌گیرد.

RAID 1+0



شکل زیر گویای مطلب است:

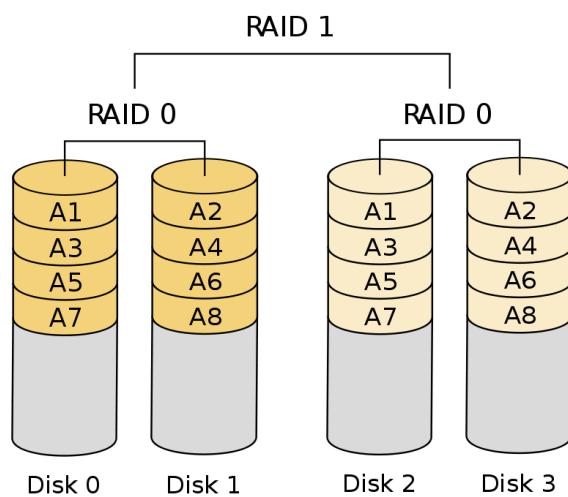


توجه: این سطح در حداقل 4 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 4)

01 سطح RAID

در سطح RAID 01 ، تکنیک Block-level striping، and mirroring without parity مورد استفاده قرار می‌گیرد.

RAID 0+1



توجه: این سطح در حداقل 4 دیسک و بیشتر کاربرد دارد. (minimum number of drives: 4)

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های همروند

۱۰۲ - کدام مورد، سیستم عامل را مجبور می‌کند دستورات S_1 ، S_2 ، S_3 و S_4 که به ترتیب در پردازهای همروند P_1 ، P_2 ، P_3 و P_4 قرار دارند به همان ترتیب S_1 ، S_2 ، S_3 و S_4 اجرا کند؟

(مقدار اولیه سمافورها ۰)

(مهندسی کامپیوٹر - 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)	wait (c) (۱ S_4
S_1 signal (a) signal (b)	wait (b) S_2 signal (a)	wait (a) S_3 signal (b)	wait (a) (۲ 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)	wait (a) (۳ 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)	wait (a) (۴ signal (b) signal (c) S_4

پاسخ تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های همروند

۱۰۲- 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 صفت تشکیل شده است.

ساختار کلی سمافور عمومی به صورت زیر است:

```
Struct semaphore
{
    Int count;
    Queue Type Queue;
} s;
```

(s.count) شمارنده سمافور

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 مسئله‌ی انتظار مشغول و حل مسئله‌ی اولویت معکوس از این صفت استفاده می‌گردد. فرآیندهای منتظر ورود به ناحیه‌ی بحرانی در این صفت نگهداری می‌شوند. اگر آزاد شدن یا خروج از این صفت به ترتیب ورود باشد، اصطلاحاً به آن سمافور قوی می‌گویند و در صورتی که ترتیب خروج مشخص نشده باشد، به آن سمافور ضعیف گفته می‌شود. سمافورهای قوی عدم گرسنگی را تضمین می‌کنند، اما در سمافورهای ضعیف این گونه نیست. در این کتاب کلیه سمافورها، از نوع قوی فرض می‌شوند، مگر اینکه نوع سمافور ضعیف بیان شود. سیستم عامل‌ها نیز معمولاً از سمافور قوی استفاده می‌کنند.

بر روی سمافور عمومی s دو تابع (s) signal و (s) wait اعمال ورود و خروج از ناحیه‌ی بحرانی را کنترل می‌کنند.

تابع (s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.
ساختر 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: راه حل سمافور و تابع اتمیک wait باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می‌توان این راه حل را توسط سرویس‌های سیستم عامل شبیه‌سازی کرد.

شرح تابع: پس از فراخوانی تابع (s) wait توسط یک فرآیند علاقمند به ورود به ناحیه‌ی بحرانی، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود ($s.count = s.count - 1$)، سپس اگر شرط مربوط به دستور (s.count < 0) if برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صفت 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 بلوک شده در صفت سمافور است.

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

ساختار این تابع به صورت زیر است:

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

توجه: راه حل سمافور و تابع اتمیک signal باید توسط سیستم عامل پشتیبانی گردد، در غیر اینصورت می توان این راه حل را توسط سرویس های سیستم عامل شبیه سازی کرد.

شرح تابع: پس از فرآخوانی تابع (signal(s)) توسط یک فرآیند علاقه مند به خروج از ناحیه بحرانی، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می شود ($s.count = s.count + 1$)، سپس اگر شرط مربوط به دستور ($s.count <= 0$) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه مند ورود به ناحیه بحرانی که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فرآخوانی تابع (signal(s)) توسط تابع wake up () بیدار شده، یعنی تغییر وضعیت داده و از وضعیت متظر به صف آمده منتقل می گردد. بنابراین این فرآیند پس از حضور در صف آمده هی پردازندۀ، این شانس را دارد تا توسط زمانبند کوتاه مدت، انتخاب شود و پردازندۀ را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند که از ناحیه بحرانی خارج شود، با اجرای تابع (signal(s))، فرآیند سر صف سمافور را بیدار می کند و اگر صف سمافور خالی باشد و هیچ فرآیند خواهدی داشت در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می شود و تابع خاتمه می یابد.

توجه: با توجه به مقادیر شمارنده های سمافور مطرح شده در سوال، $a = 0$ ، $b = 0$ و $c = 0$ ، فقط و فقط فرآیند S_1 می تواند در ابتدا اجرا شود.

توجه: فرآیندهای P_1 ، P_2 و P_4 قادر حلقه هستند، و توسط شرایطی که برای هم فراهم می کنند، در صورت امکان فقط و فقط می توانند 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 می شود.

فرآیند	دستور	a	b	c	خروجی
P ₄ و P ₃ ، P ₂ ، P ₁	-	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 P₂ تحت هر شرایطی، قبل از اتمام کامل فرآیند P₁ و انجام signal(a) اجرا شود چون اول wait(a) با مقدار 0 را اجرا می‌کند، پس می‌خوابد، همچنین اگر فرآیند P₃ تحت هر شرایطی، قبل از اتمام کامل فرآیند P₂ و انجام signal(b) اجرا شود چون اول wait(b) با مقدار 0 را اجرا می‌کند، پس می‌خوابد. همچنین اگر فرآیند P₄ تحت هر شرایطی، قبل از اتمام کامل فرآیند P₃ 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 P₁ ، P₂ ، P₃ و P₄ در نهایت فقط و فقط منجر به اجرای دستورات S₁ ، S₂ ، S₃ و S₄ 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_4 و P_3 ، P_2 ، P_1	-	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 سenarioی دیگری هم وجود دارد که اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 می‌توانند 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_4 و S_2 ، S_3 ، S_1 شود.

فرآیند	دستور	a	b	c	خروجی
P_4 و P_3 ، P_2 ، P_1	-	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.

فرآیند	دستور	a	b	c	خروچی
P_4 و P_3 ، P_2 ، P_1	-	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 سـناریوی دیگری هم وجود دارد که اجرای فرآیندهای P_1 ، P_2 ، P_3 و P_4 مـی‌تواند منجر به 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_4 ، P_3 ، P_2 ، P_1	-	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	مـی‌خوابد.
P_2	wait (a) S_2 signal (a) signal (a)	-2	0	0	مـی‌خوابد.
P_4	wait (a) wait (a) wait (a) S_4 signal (a) signal (a) signal (a) signal (a)	-3	0	0	مـی‌خوابد.

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_4 و P_3 ، P_2 ، P_1	-	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.

تست‌های فصل چهارم: مدیریت حافظه اصلی

۱۰۳- فرض کنید که طول آدرس مجازی 47 بیت و اندازه صفحه KB 16 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؟

(مهندسى کامپیوترا - دولتی ۱۴۰۰)

۵ (۴)

۴ (۳)

۳ (۲)

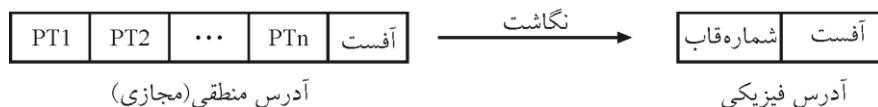
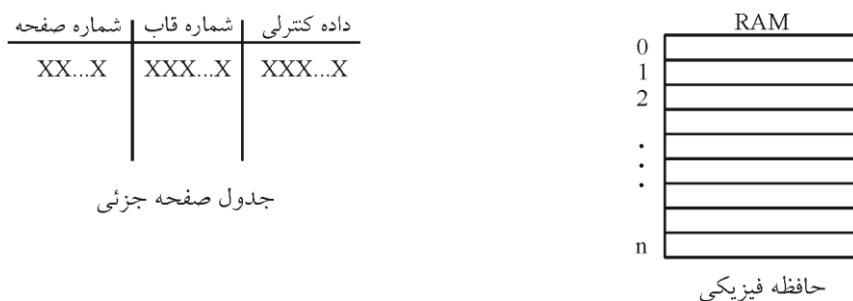
۲ (۱)

عنوان کتاب: سیستم عامل
مولف: ارسسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل چهارم: مدیریت حافظه اصلی

۱۰۳- 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:



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

$= 8B$

توجه: مطابق فرض سؤال، هر مدخل جدول صفحه (عرض جدول صفحه جزئی) 8 بایت در نظر گرفته شده است.

$$\frac{\text{اندازه قاب}}{\text{عرض جدول صفحه}} = \frac{2^4 \times 2^{10} B}{2^3 B} = \frac{2^4 \times 2^{10}}{2^3} = 2^{11} = 2048$$

$2^{47} B$ = اندازه فرآیند (فضای آدرس مجازی)

$16 KB = 16384 B = 2^4 \times 2^{10} B = 2^{14} B$ = اندازه قاب

$$f = \frac{\text{اندازه فرآیند}}{\text{تعداد صفحات فرآیند}} = \frac{2^{47}}{2^{14}} = 2^{33}$$

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

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

روش تجزیه

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

$$2^{33} = 2^{11} \times 2^{11} \times 2^{11}$$

↑ ↑ ↑
PT1 PT2 PT3

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

روش لگاریتم

$$d = \lceil \log_r f \rceil = \lceil \log_{2^{11}} 2^{33} \rceil = 3$$

روش تقسیم متوالی

$$\frac{\text{تعداد صفحات فرآیند}}{r} = \frac{f}{r} = \frac{2^{33}}{2^{11}} = 2^{22}$$

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

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

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.

$$\log_2^{\text{اندازه صفحه}} = \log_2^{2^{14}} = 14$$

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

PT1	PT2	PT3	آفس
بیت 11	بیت 11	بیت 11	14
بیت 33			

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۱۰۴- 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  $i = 0 \text{ OR } i = 1$  به صورت زیر باشد، کدام گزینه صحیح است؟
    (مهندسى کامپیووتر - دولتى 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.

پاسخ تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های همروند

۱۰۴- گزینه (۴) صحیح است.

یک ریاضی دان هلندی به نام Decker اولین کسی بود که یک راه حل نرم‌افزاری «دو فرآیندی» درست برای حل مسئله شرایط رقابتی ارائه داد. و سر انجام این راه حل در سال ۱۹۶۵ 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:

P_0 :

```
{  
  (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 */  
}
```

P_1 :

```
{  
  (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

توجه: اگر فرآیند P_0 علاقه‌مند ورود به ناحیه بحرانی باشد، ابتدا flag مربوط به خود را به TRUE مقداردهی می‌کند (پرچم خود را بالا می‌برد) و سپس flag مربوط به فرآیند P_1 را بررسی می‌کند. همچنین اگر فرآیند P_1 علاقه‌مند ورود به ناحیه بحرانی باشد، ابتدا flag مربوط به خود را به TRUE مقداردهی می‌کند (پرچم خود را بالا می‌برد) و سپس flag مربوط به فرآیند P_0 را بررسی 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 مقدار turn برابر 0 باشد، ورود به ناحیه بحرانی حق مسلم فرآیند P_0 است و به آن باید اصرار و پافشاری کند. و اگر 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:

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

توجه: مانند این آزمون‌ها رابه عنوان مبدع آن «قوانين ارسسطو» نام‌گذاری کردیم، این قوانین به «قوانين چهارگانه ارسسطو» نیز موسوم است.

قانون اول ارسسطو (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) 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*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدھیم، 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 بدھید.

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

P_1 :

(۱) flag [1] = TRUE;

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

(۲) while (flag [0]) {

توجه: شرط حلقه TRUE است، پس بدنه حلقه یعنی دستور {...} if (turn == 0) اجرا می‌شود.

(۳) if (turn == 0) {

(۴) flag [1] = FALSE;

توجه: هم اکنون مقدار turn برابر ۰ است، پس شرط دستور if (turn == 0) TRUE و به تبع مقدار [1] برابر flag FALSE می‌شود.

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

(۵) while (turn == 0);

توجه: هم اکنون مقدار turn برابر ۰ است، پس شرط حلقه while (turn == 0); TRUE برابر 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 مدامی 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 گرفته است. 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.

قانون دوم ارسسطو(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:

P₀:

(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:

P₁:

(1) flag [1] = TRUE;

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

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه بحرانی را دارند. بنابراین آرایه فرآیندها هر دو $flag [1] = TRUE$ و $flag [0] = TRUE$ می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه بحرانی هستند.

در ادامه پردازنده را از فرآیند P_1 بگیرید و به فرآیند P_0 بدهید.

توجه: هم اکنون $flag [1] = TRUE$ و $flag [0] = TRUE$ 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 کتترل برنامه از کل بدنه دستور if خارج شده و کتترل برنامه مجدداً به ابتدای حلقه جاری یعنی (flag [1]) باز می‌گردد.

توجه: هم اکنون مقدار [1] flag برابر TRUE است، پس شرط حلقه (flag [1]) برابر TRUE است، پس کتترل 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.

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

توجه: هم اکنون flag [0] = TRUE و flag [1] = FALSE 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] = FALSE و flag [1] = TRUE 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 کتترل 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.

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

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

(2) while (flag [1]) {

شرط حلقه FALSE است، پس کتترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P₀ قرار نمی‌گیرد، 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 بدھیم 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 ارسسطو (آزمون اول شرط انحصار متقابل) و قانون دوم ارسسطو (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشند. بنابراین شرط انحصار متقابل در سوال مطرح شده برقرار است.

قانون سوم ارسسطو (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 مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجدداً 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 :

(۱) flag [0] = TRUE;

توجه: هم اکنون $turn = 0$ و $flag [0] = TRUE$ و $flag [1] = FALSE$.

(۲) 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*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدھیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازندۀ 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 بدھیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازندۀ داخل ناحیه باقیمانده فرآیند P_0 مشغول حرکت است.

(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی: در ادامه پردازندۀ را از فرآیند P_0 بگیرید و به فرآیند P_1 بدھید. 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 بدھیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۴) می‌شویم. هم اکنون پردازندۀ 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*/
```

```
(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 بدھیم، خب قراردادیم. 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 برگرد و مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجدداً 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 مجدداً 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 برگرد و مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجدداً وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، خب شد، فرآیند دوم مجدداً وارد ناحیه بحرانی خودش شد. بنابراین شرط پیشرفت برقرار است.

قانون چهارم ارسسطو (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 برگرد و مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجدداً 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:

$turn = 0$, $flag[0] = FALSE$, $flag[1] = FALSE$

(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 :

(۱) $\text{flag}[0] = \text{TRUE}$;

توجه: هم اکنون $\text{turn} = 0$ و $\text{flag}[0] = \text{TRUE}$ و $\text{flag}[1] = \text{FALSE}$ است.

(۲) $\text{while}(\text{flag}[1]) \{$

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 :

(۱) $\text{flag}[1] = \text{TRUE}$;

توجه: هم اکنون $\text{turn} = 0$ و $\text{flag}[0] = \text{TRUE}$ و $\text{flag}[1] = \text{TRUE}$ است.

(۲) $\text{while}(\text{flag}[0]) \{$

توجه: شرط حلقه TRUE است، پس بدنه حلقه یعنی دستور $\{\dots\}$ اجرا می‌شود.

(۳) $\text{if}(\text{turn} == 0) \{$

(۴) $\text{flag}[1] = \text{FALSE}$;

توجه: هم اکنون مقدار turn برابر ۰ است، پس شرط دستور $\text{if}(\text{turn} == 0)$ if برابر TRUE و به تبع

مقدار [۱] flag برابر FALSE می‌شود. یعنی اگر مقدار کنونی turn برابر ۰ 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 $\text{turn} = 0$ و $\text{flag}[0] = \text{TRUE}$ و $\text{flag}[1] = \text{FALSE}$ است.

(۵) $\text{while}(\text{turn} == 0);$

توجه: هم اکنون مقدار turn برابر ۰ است، پس شرط حلقه $\text{while}(\text{turn} == 0)$; while 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.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدھیم، 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 بدھید.

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 $turn = 1$ 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 بدھیم، 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، در حال حاضر مقدار کنونی turn برابر ۱ 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 کرده است و پایین کشیده است یعنی $flag[0] = FALSE$ است.

توجه بسیار مهم: اما اگر فرآیند P_0 با وجود اینکه شمشیر و پرچم خود را غلاف کرده است و پایین کشیده است، بخواهد بدعهدی، بداخلاتی 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 حق دار 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 یافته باشد و به همین سبب مغروف 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!

(گام ۴): فرآیند اول به ابتدای برنامه برگرد و مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجدداً 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 ۱ 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 تابع دچار «گرسنگی موقت» شده است، اما این گرسنگی ابدی و همیشگی نیست، یعنی:

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

P_0 :

(1) flag [0] = TRUE;

توجه: هم اکنون $\text{flag}[0] = \text{TRUE}$ و $\text{flag}[1] = \text{FALSE}$ و $\text{turn} = 1$ 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 ۱ است و نوبتی هم که باشد نوبت فرآیند P_1 است که از critical section خودش استفاده کند، با بدنهای 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 آم» به طور پی در پی بتواند وارد critical section برای 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 متسب 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 حق خواهی دقیقاً از همانجا شروع می‌شود که وقت و کوانتم و برره 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 کرده باشد، یعنی فرآیند P_0 در چه موقعیت زمانی و مکانی از قطعه کد خود به بازی

ناجونمردانه خود پس از «بار دوم» به بعد پایان می‌دهد، آنگاه شرط حلقه while ($turn == 0$) 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 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 قبل در فرآیند P_0 مقداردهی 1 شده بود) و سپس مقدار [1] خود را برابر TRUE قرار می‌دهد، یعنی پرچم خود را بالا می‌برد و شمشیر خود را بیرون می‌کشد. وقت 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]) بسته شده است. بنابراین بعد انجام دستور flag [1] = TRUE در فرآیند P_1 کنترل برنامه به ابتداء و بررسی شرط حلقه (flag [0]) می‌رود. از آنجا که مقدار flag [0] قبل و قبل از ورود به critical section در بار دوم در فرآیند P_0 برابر TRUE شده است خط (1) و همچنین متغیر turn قلا در فرآیند P_0 مقداردهی 1 شده است، در نتیجه بررسی شرط حلقه (flag [0]) 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 {...} if (turn == 0) {...} اجرا می‌شود.

(3) if (turn == 0) {

(4) flag [1] = FALSE;

توجه: هم اکنون مقدار turn برابر 1 است، پس شرط دستور if (turn == 0) 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 while (turn == 0) 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]) 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 P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

حلت دوم: مهلت زمانی پردازنه فرآیند P0 در نقطه **remainder section** برای بار دوم» به بعد تمام شود:

در این حالت پردازنه از فرآیند P0 به فرآیند P1 تعویض متن می‌کند و در ادامه شرط حلقه جاری یعنی $(turn == 0)$; در فرآیند P1 مورد بررسی قرار می‌گیرد. در این حلت فرآیند P1 از حلقه; $(turn == 0)$; while خارج می‌شود، چون در حال حاضر مقدار متغیر turn برابر 1 است (متغیر turn قبل در فرآیند P0 مقداردهی 1 شده بود) و سپس مقدار [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 $[1] = TRUE$; در فرآیند P1 کترل برنامه به ناحیه بحرانی (critical section) نمی‌رود چون قبل critical section نماد «}» مربوط به کترل حلقه جاری یعنی $([0]$ while (flag بسته شده است. بنابراین بعد انجام دستور $[1] = TRUE$; در فرآیند P1 کترل برنامه به ابتداء و بررسی شرط حلقه $([0]$ flag می‌رود. از آنجا که مقدار $[0]$ قبل و قبل از ورود به remainder section در بار دوم در فرآیند P0 برابر FALSE شده است خط (8)، پس شرط حلقه FALSE است، پس کترول برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P1 قرار می‌گیرد، به صورت زیر:

P1:

/*critical section*/

که در نهایت حق به حق دار می‌رسد و سرانجام فرآیند P1 نیز می‌تولند وارد ناحیه بحرانی خودش شود.

همانطور که در (گام ۴) گفته شد که فرآیند اول به ابتدای برنامه برگرد و مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که به طور مکرر مجدداً وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب نشد، فرآیند اول نتوانست به طور مکرر مجدداً وارد ناحیه بحرانی خودش بشود. بنابراین شرط انتظار محدود به دلیل نبود گرسنگی برقرار است.

قانون دوم ارسسطو(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 متظر 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 بدھیم 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 ارسسطو (آزمون بن‌بست) و قانون چهارم ارسسطو (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.

تست‌های فصل هفتم: مدیریت بن‌بست

۱۰۵- یک کامپیوتر دارای m چاپگر از یک نوع است. این چاپگرها به وسیله ۳ پردازه A و B و C استفاده می‌شوند که در زمان بیشترین نیاز (حداکثر تقاضا) به ترتیب به ۳ 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)

13 (۴)

12 (۳)

11 (۲)

10 (۱)

عنوان کتاب: سیستم عامل

مولف: ارسسطو خلیلی‌فر

ناشر: انتشارات راهیان ارشاد

آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل هفتم: مدیریت بن‌بست

۱۰۵- 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:

$$\sum_{i=1}^n \text{Request}[i] < m + n : \text{مجموع درخواست‌های فرآیندها برای منابع } m + n$$

$$\rightarrow 3+4+6 < m+n \rightarrow 13 < m+3 \rightarrow m > 10 \rightarrow m_{\min} = 11$$

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 فرآیند 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 بدین ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(1-r)$ 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 $(1-r)$ 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 $2+3+5=10$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

$$m > 10 \rightarrow m_{\min} = 11$$

تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۱۰۶- دو پردازه متناوب با مشخصات زیر مفروض است. کدام گزینه بزرگترین مقدار x را برای پردازه ۲ نشان می‌دهد به نحوی که زمان‌بندی قبضه‌ای (نرخ یکنواخت) (مهندسى کامپیوترا - دولتی ۱۴۰۰)

امکان‌پذیر باشد؟

	Period	CPU Time		
P ₁	50	25	35 (۴)	25 (۲)
P ₂	80	x	30 (۳)	20 (۱)

عنوان کتاب: سیستم عامل

مؤلف: ارسسطو خلیلی‌فر

ناشر: انتشارات راهیان ارشد

آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازندۀ

۱۰۶- گزینه (۳) صحیح است.

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 الزاماً 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، ضرب العجل‌ها 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 وسائل 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

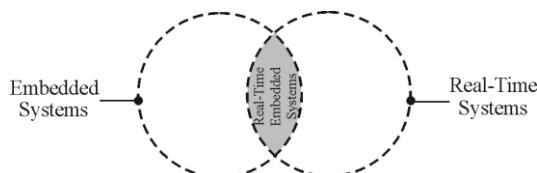
عامل باید با شرایط سیستم‌های بی‌درنگ سخت سازگار باشد یعنی سیستم عامل هم بی‌درنگ باشد. سیستم عامل بی‌درنگ نوعی سیستم عامل است که در آن، زمان پارامتر کلیدی است. سیستم بی‌درنگ به سیستمی گفته می‌شود که درستی اجرای یک عملیات در آن تنها به درست بودن عملیات از نظر منطقی بستگی نداشته باشد بلکه اجرای آن عملیات 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 شهراب 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 رزو بلیط هوایپیما در صورت خالی شدن لحظه‌ای یک صندلی، خوب است اولویت و حق رعایت شود و فوراً رزو انجام شود، ولی اگر هم ضرب‌الاجل‌ها گاه‌ها 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 فوراً اعلام شود، ولی اگر هم ضرب‌الاجل‌ها گاه‌ها 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. توجه 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، باز هم 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 شدن این ارجحیت را حفظ می‌کند. در واقع در سیستم‌های 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 بی‌درنگ نرم می‌بایست پاسخی سریع داشته باشند ولی مساله پاسخ‌دهی به حادی سیستم‌های بی‌درنگ سخت نمی‌باشد.

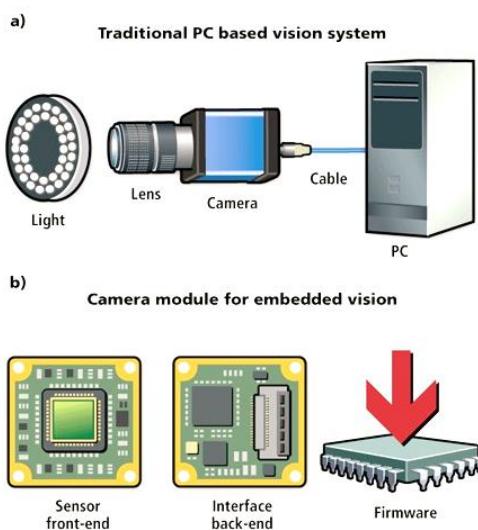
انسان در وادی زندگی نیازهای گوناگونی دارد، یکی از نیازهای اساسی انسان، نیاز به امنیت است. اما گاه‌آ، ممکن است در معرض عوامل محیطی و بیرونی و یا حتی درونی امنیت انسان در شرایط هشیاری یا ناھشیاری به مخاطره بیفتند. بنابراین نیاز است تا مکانیزمی همواره هوشیار و همیشه بیدار و با اشراف لحظه به لحظه، مخاطرات 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 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 دامنه وسیعی از وسائل 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 تعییه 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 نظری 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 نظری 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 مقررین به صرفه‌بودن هزینه‌های تولید، به شکل ابیوه تولید شده‌اند. امروزه درون اکثر وسائل و دستگاه‌های پیامون مـا (خودپرداز، تلفن همراه، اتومبیل و

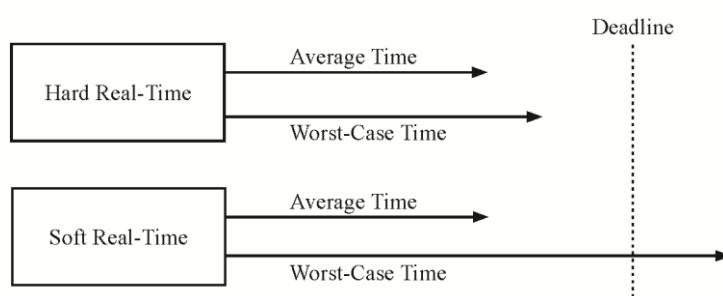
ماشین لباسشویی) سیستم نهفته قرار دارد. شکل زیر یک سیستم امنیتی را در دو شکل نهفته و (مبتنی بر PC) نشان می‌دهد.



در علم کامپیوتر، محاسبات بی‌درنگ (RTC: Real-Time Computing) و یا محاسبات واکنشی (Reactive Computing)، 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 حتماً 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 Deadline 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:

Hard Vs. Soft Real-Time Applications



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 معمولاً 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».

نوجه: 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.

زمان‌بندی بی‌درنگ

در سیستم‌های بی‌درنگ، زمان پاسخگویی به فرآیندها نباید از یک حد مشخصی (مهلت زمانی) بیش‌تر شود، در غیر اینصورت از بین خواهند رفت. سیستم‌های بی‌درنگ معمولاً به شیوه‌ای پیاده‌سازی می‌شوند که باید به اتفاقات و حوادث خارجی سریعاً پاسخ دهند. این وقایع به دو طبقه متناوب و غیرمتناوب تقسیم می‌شوند:

وقایع متناوب (مساوی): آن دسته از وقایعی که در فواصل زمانی منظم و مساوی رخ می‌دهند، 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 غیرمساوی رخ می‌دهند، مانند فشردن پدال ترمز اتومبیل در فواصل زمانی نامنظم و غیرمساوی.

شرط زمان‌بندی

زمان‌بندی تمام وقایع متناوب باهم در یک سیستم بی‌درنگ بر اساس رابطه زیر ممکن است 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، بلکه به این معنی است که زمان‌بندی تمام وقایع متناوب باهم در یک سیستم بی‌درنگ بر اساس برقراری معیار فوق ممکن است شدنی باشد یا نباشد.

توجه: یک سیستم بی‌درنگ که منطبق با معیار فوق نباشد، به طور قطع غیرقابل زمان‌بندی است، اما اگر این معیار را داشته باشد یک الگوریتم خوب ممکن است بتواند آنرا زمان‌بندی کرده و همه مهلت‌ها را محقق کند.

سوال: در یک سیستم بی‌درنگ نرم، ۳ واقعه متناوب با دوره‌های تناوب ۱۰۰ و ۲۰۰ و ۵۰۰ میلی‌ثانیه وجود دارند، اگر زمان پردازش هر واقعه به ترتیب ۵۰ و ۳۰ و ۱۰۰ میلی‌ثانیه باشد، آیا این سیستم قابل زمان‌بندی است؟

پاسخ:

$$\sum_{i=1}^3 \frac{t_i}{p_i} = \frac{50}{100} + \frac{30}{200} + \frac{100}{500} = 0.5 + 0.15 + 0.2 = 0.85 \leq 1$$

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 (پیش‌آجرا)، اما در الگوریتم‌های پویا، اولویت زمان‌بندی حین اجرای فرآیندها تعیین می‌شود (حین‌آجرا). 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 دوره تناوب کمتر، دارای اولویت بیشتر خواهند بود.

در زمان اجرای فرآیندها، زمان‌بند پردازندۀ همیشه فرآیندی که بیشترین اولویت را دارد، انتخاب می‌کند و چنانچه فرآیندی با اولویت بیشتر در صفت آماده حضور یابد، بلاfaciale فرآیند جاری را قبضه خواهد کرد.

مثال اول: دو فرآیند متناوب P_1 و P_2 با دوره تناوب به ترتیب برابر 50 و $p_1=100$ و $p_2=35$ میلی‌ثانیه و زمان پردازش (CPU Burst) به ترتیب برابر 20 و $t_1=20$ و $t_2=35$ را در نظر بگیرید، همچنین فرض کنید مهلت زمانی انجام هر فرآیند برابر همان دوره تناوب باشد، آیا این سیستم قابل زمان‌بندی است؟

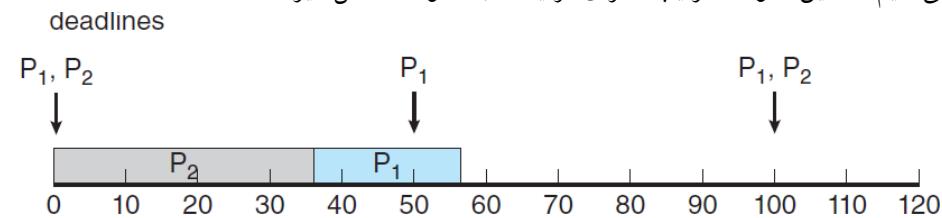
	Period	CPU Time
P_1	50	20
P_2	100	35

پاسخ:

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{20}{50} + \frac{35}{100} = 0.4 + 0.35 = 0.75 \leq 1$$

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 مثل 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:

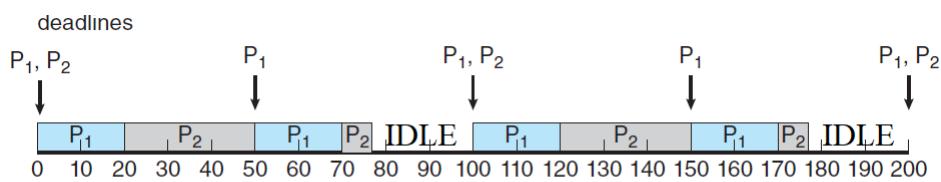


Scheduling of tasks when P_2 has a higher priority than P_1 .

واضح است که ابتدا P_2 ، اجرای خود را آغاز نموده و در زمان 35 خاتمه می‌یابد. در این نقطه، 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 داشته است. بنابراین مهلت زمانی برای P_1 برآورده نشده است پس الگوریتم مذکور نتوانسته است

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 خواهد 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.

واضح است که ابتدا P_1 ، اجرای خود را آغاز نموده و در زمان 20 خاتمه می‌یابد. و مهلت زمانی اول فرآیند 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 باقی‌مانده است 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 رسیده است. 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. وقتی 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 $p_1=50$ و $p_2=80$ میلی‌ثانیه و زمان پردازش (CPU Burst) به ترتیب برابر $t_1=25$ و $t_2=35$ را در نظر بگیرید، همچنین فرض کنید مهلت زمانی انجام هر فرآیند برابر همان دوره تناوب باشد، آیا این سیستم قابل زمان‌بندی است؟

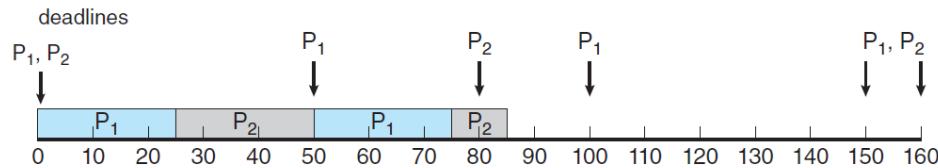
	Period	CPU Time
P ₁	50	25
P ₂	80	35

پاسخ:

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{35}{80} = 0.5 + 0.44 = 0.94 \leq 1$$

واضح است که در مجموع، بهره‌وری فرآیندهای متناوب P₁ و P₂ برابر 0.94 است. بنابراین از نظر تکویری 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 P₁ اولویت بالاتری را نسبت به P₂ دریافت می‌کند. زیرا فرآیند P₁ دوره تناوب کمتر و به تبع فرکانس و اولویت بیشتری دارد. در این صورت ترتیب اجرای فرآیندها به صورت شکل زیر است:



Missing deadlines with rate-monotonic scheduling.

واضح است که ابتدا P₁، اجرای خود را آغاز نموده و در زمان 25 خاتمه می‌یابد. و مهلت زمانی اول فرآیند P₁ هم برآورده می‌شود. در این نقطه، P₂ اجرای خود را آغاز کرده و تا لحظه 50 اجرا می‌شود. در لحظه 50 با اینکه 10 میلی‌ثانیه از زمان اجرای فرآیند P₂ باقی‌مانده است اما مطابق الگوریتم نرخ یکنواخت، پردازنده از P₂ (اولویت کمتر) گرفته می‌شود و به P₁ (اولویت بیشتر) داده می‌شود چون در لحظه 50 رویداد راهانداز بار دوم برای فرآیند P₁ از راه رسیده است. مطابق تعریف الگوریتم نرخ یکنواخت همواره اولویت با اولویت بیشتر و به صورت «غیرانحصاری» (non-preemptive) یا قبضه‌ای است. در ادامه فرآیند P₁ پردازش خود را در نقطه 75 کامل می‌کند که در این لحظه P₂ از پایان اجرای قبلی، کار خود را ادامه می‌دهد. P₂ پردازش خود را در لحظه 85 تکمیل می‌کند غافل از اینکه در لحظه 80 رویداد راه انداز P₂ مجدد رخ داده است و بنابراین مهلت P₂ 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 باید از یک حد مشخصی (مهلت زمانی) بیشتر شود، در غیر اینصورت از بین خواهند رفت.

دقت کنید که فرآیند P_1 هر ۵۰ میلی ثانیه یکبار و در لحظات ۰ و ۵۰ و ۱۰۰ و ۱۵۰ و ... و فرآیند P_2 هر ۸۰ میلی ثانیه یکبار و در لحظات ۰ و ۸۰ و ۱۶۰ و ... و این چرخه ادامه دارد و در نهایت مهلت زمانی هر دو فرآیند محقق نمی شود.

توجه: ممکن است در مسئله‌ای معیار قابلیت زمان‌بندی در تئوری برقرار باشد، یعنی بهره‌وری پردازنده کمتر یا مساوی ۱ 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 سیستم شامل N فرآیند از رابطه زیر محاسبه می شود:

$$N(2^{\frac{1}{N}} - 1)$$

مقادیر زیر برای رابطه فوق برقرار است:

$N(2^{\frac{1}{N}} - 1)$	N	Utilization
	1	1
	2	0.83
	∞	0.69

توجه: مطابق جدول فوق، واضح است که با در نظر گرفتن فقط و فقط یک فرآیند در سیستم بی‌درنگ، بهره‌وری پردازنده (CPU) برابر ۱۰۰ درصد خواهد بود.

توجه: در مثال اول، بهره‌وری پردازنده (CPU) در دو حالت بدون الگوریتم نرخ یکنواخت و با الگوریتم نرخ یکنواخت برابر مقدار ۰.۷۵ 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 ۰.۷۵ از مقدار ۰.۸۳ به عنوان حداقل مقدار بهره‌وری پردازنده (CPU) در حالت ۲ فرآیندی کمتر هم بود، مطابق رابطه‌ی زیر:

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{20}{50} + \frac{35}{100} = 0.4 + 0.35 = 0.75 \leq N(2^{\frac{1}{N}} - 1) = 0.83 \leq 1$$

توجه: بنابراین در حالت کمتر یا مساوی حداقل مقدار بهره‌وری پردازنده (CPU)، الگوریتم زمان‌بندی نرخ یکنواخت، «تصمیم» 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 ۰.۹۴ 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 ۰.۹۴ از مقدار ۰.۸۳ به عنوان حداقل مقدار بهره‌وری پردازنده (CPU) در حالت ۲ فرآیندی بیشتر هم بود، مطابق رابطه‌ی زیر:

$$N(2^{\frac{1}{N}} - 1) = 0.83 \leq \sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{35}{80} = 0.5 + 0.44 = 0.94 \leq 1$$

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 حتماً 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؟

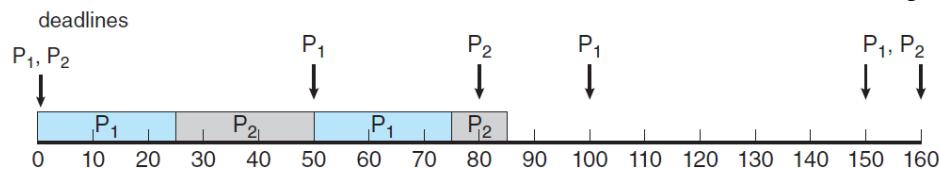
	Period	CPU Time
P ₁	50	25
P ₂	80	x

پاسخ: برای کشف بزرگترین مقدار x برای P₂ به نحوی که زمانبندی قبضه‌ای (نرخ یکنواخت) Rate Monotonic امکان‌پذیر باشد باید گزینه‌هایی که شرط زمانبندی را برقرار می‌کنند، در نمودار گانت نیز بررسی شوند. به عبارت دیگر هم از نظر تغوری 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

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{35}{80} = 0.5 + 0.44 = 0.94 \leq 1$$

همانطور که در مثال دوم بررسی کردیم، دیدیم که مقادیر فوق در نمودار گانت قابل زمانبندی کامل و دوره‌ای نبود.



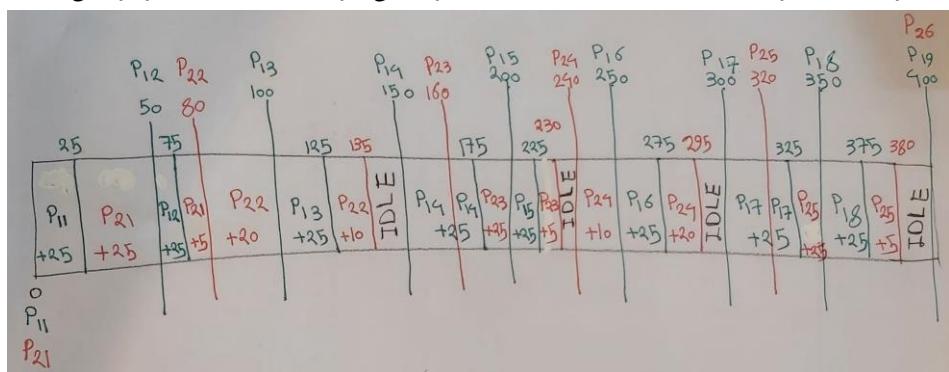
Missing deadlines with rate-monotonic scheduling.

گزینه سوم پاسخ سوال است.

	Period	CPU Time
P ₁	50	25
P ₂	80	x=30

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{30}{80} = 0.5 + 0.375 = 0.875 \leq 1$$

مقادیر فوق از نظر تئوری درست است، اما باید از نظر عملی نیز نمودار گانت آن نیز بررسی شود:



واضح است که مهلت‌های زمانی فرآیندها در نمودار گانت برآورده شده‌است، چون در لحظه 400 به تناوب و ملاقات مجدد فرآیندها باهم رسیده‌اند.

در گزینه سوم مهلت‌های زمانی فرآیندها از نظر تئوری و عملی قابل زمان‌بندی است. و پاسخ سوال هم هست چون صورت سوال، بزرگترین مقدار ممکن را خواسته است.

گزینه دوم پاسخ سوال نیست.

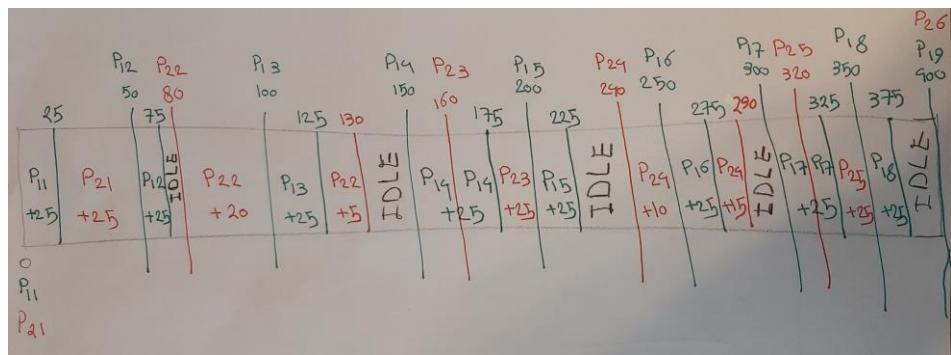
	Period	CPU Time
P ₁	50	25
P ₂	80	x=25

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{25}{80} = 0.5 + 0.3125 = 0.8125 \leq 1$$

بر اساس الگوریتم نرخ یکنواخت، مقدار 0.8125 از مقدار 0.83 به عنوان حداقل مقدار بهره‌وری پردازنده (CPU) در حالت 2 فرآیندی کمتر است، مطابق رابطه‌ی زیر:

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{25}{80} = 0.5 + 0.3125 = 0.8125 \leq N(2^{\frac{1}{N}} - 1) = 0.83 \leq 1$$

توجه: همانطور که پیش‌تر گفتیم، در حالت کمتر یا مساوی حداقل مقدار بهره‌وری پردازنده (CPU)، الگوریتم زمان‌بندی نرخ یکنواخت، «تضمين» 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 دوم مهلت‌های زمانی فرآیندها از نظر تئوری و عملی قابل زمانبندی است. اما پاسخ سوال نیست چون صورت سوال، بزرگترین مقدار ممکن را خواسته است.

گزینه اول پاسخ سوال نیست.

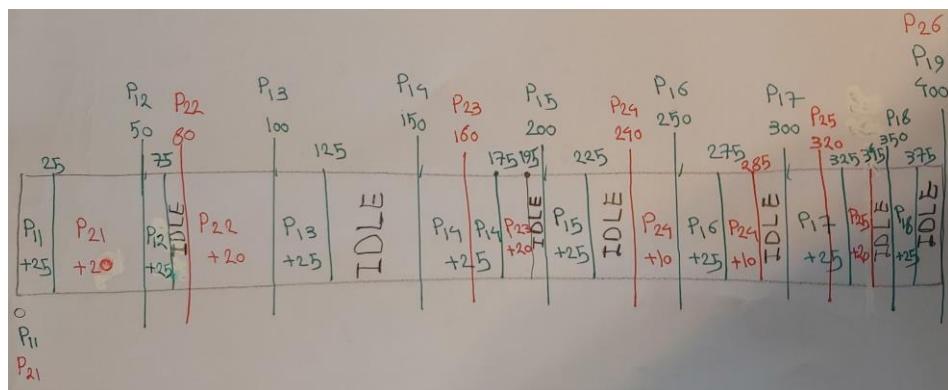
	Period	CPU Time
P ₁	50	25
P ₂	80	x=20

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{20}{80} = 0.5 + 0.25 = 0.75 \leq 1$$

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:

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{20}{80} = 0.5 + 0.25 = 0.75 \leq N(2^{\frac{1}{N}} - 1) = 0.83 \leq 1$$

توجه: همانطور که پیش‌تر گفتیم، در حالت کمتر یا مساوی حداقل مقدار بهره‌وری پردازنده (CPU)، الگوریتم زمانبندی نرخ یکنواخت، «تضمين» 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 خواسته است.
مقدار X در حداکثر (سقف) بهره‌وری از رابطه زیر محاسبه می‌شود:

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{x}{80} \leq 1$$

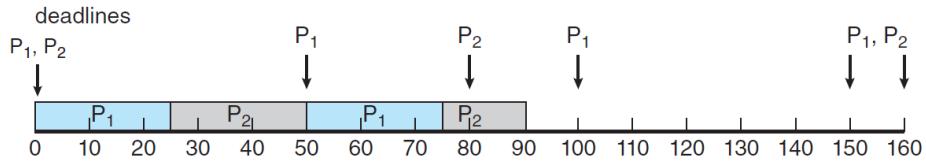
$$\frac{x}{80} \leq \frac{1}{2}$$

$$x \leq 40 \rightarrow x = 40$$

	Period	CPU Time
P ₁	50	25
P ₂	80	x=40

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{40}{80} = 0.5 + 0.5 = 1 \leq 1$$

واضح است که در مجموع، بهره‌وری فرآیندهای متناوب P₁ و P₂ 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 فلسفی 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 باقی‌مانده است 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 رسیده است. مطابق تعریف الگوریتم نرخ یکنواخت همواره اولویت با اولویت بیشتر و به صورت «غیرانحصاری» (non-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:

$$\sum_{i=1}^2 \frac{t_i}{p_i} = \frac{25}{50} + \frac{x}{80} \leq N(2^{\frac{1}{N}} - 1)$$

$$\frac{1}{2} + \frac{x}{80} \leq 2(2^{\frac{1}{2}} - 1)$$

$$0.5 + \frac{x}{80} \leq 0.83$$

$$\frac{x}{80} \leq 0.83 - 0.5$$

$$\frac{x}{80} \leq 0.33 \rightarrow x \leq 80 \times 0.33 \rightarrow x \leq 26.4$$

توجه: همانطور که دیدید مقادیر x در گزینه اول $x = 20$ و گزینه دوم $x = 25$ کوچکتر از مقدار حداقل (کف) بهره‌وری یعنی $26.4 < 25 < 20$ است. پس همانطور که پیش‌تر گفتم، در حالت کمتر یا مساوی حداقل مقدار بهره‌وری پردازنده (CPU)، الگوریتم زمان‌بندی نرخ یکنواخت، «تضمين» می‌کند که فرآيندها بگونه‌ای زمان‌بندی شوند که مهلت‌های زمانی فرآيندها در نمودار گانت برآورده شود، که دیدید چنین هم شد.

تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۱۰۷- در یک الگوریتم برنامه‌ریزی اولویت‌دار که پنج پردازه و اولویت‌های آن‌ها به صورت زیر است، وجود دارد. میانگین زمان انتظار چند میلی ثانیه است؟

فرض کنید که هر چه مقدار اولویت کمتر باشد، اولویت پردازه بیشتر است.

یعنی پردازه P_4 دارای کمترین اولویت و پردازه P_2 دارای بیشترین اولویت است.

(مهندسان کامپیوترا - دولتی ۱۴۰۰)

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

7.75 ms (۴)

8.2 ms (۳)

8 ms (۲)

7 ms (۱)

عنوان کتاب: سیستم عامل

مولف: ارسسطو خلیلی فر

ناشر: انتشارات راهیان ارشد

آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل دوم: مدیریت فرآیندها و زمانبندی پردازندۀ

۱۰۷- گزینه (۳) صحیح است.

الگوریتم‌های Priority (زمانبندی با اولویت)

در این روش زمانبندی، هریک از فرآیندها اولویت مخصوص به خود را دارد. 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 الگوریتم‌های Priority، 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.

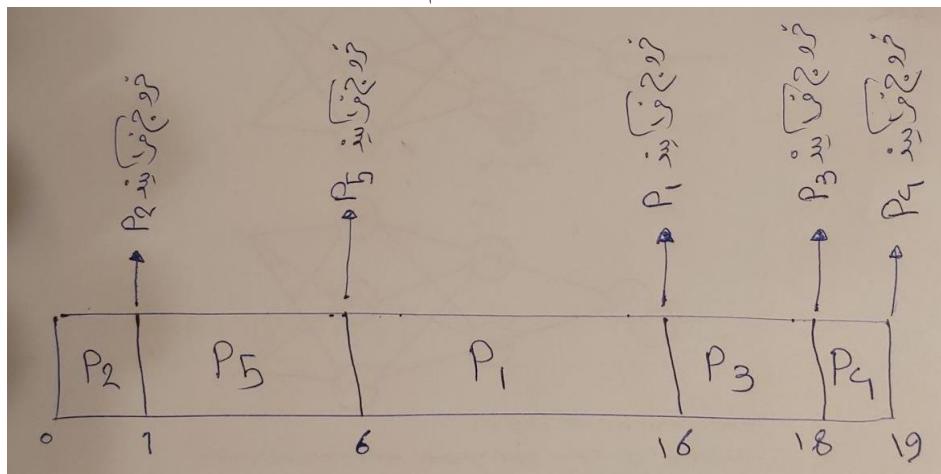
- یک اولویت دینامیک برای تغییراتی که در محیط اتفاق می‌افتد تغییر می‌کند.

توجه: در الگوریتم‌های اولویت، جهت مقابله با مشکل قحطی‌زدگی برحی از فرآیندها، می‌توان از تکنیکی موسوم به سالخوردگی (Aging) استفاده کرد. در این تکنیک به تدریج اولویت

پردازش‌هایی که مدت مديدة 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

$$P_1 = 16 - 0 = 16$$

$$P_2 = 1 - 0 = 1$$

$$P_3 = 18 - 0 = 18$$

$$P_4 = 19 - 0 = 19$$

$$P_5 = 6 - 0 = 6$$

$$\text{ATT} = \frac{16+1+18+19+6}{5} = \frac{60}{5} = 12$$

زمان اجرای فرآیند - زمان بازگشت فرآیند = زمان انتظار فرآیند

$$P_1 = 16 - 10 = 6$$

$$P_2 = \text{زمان انتظار} = 1 - 1 = 0$$

$$P_3 = \text{زمان انتظار} = 18 - 2 = 16$$

$$P_4 = \text{زمان انتظار} = 19 - 1 = 18$$

$$P_5 = \text{زمان انتظار} = 6 - 5 = 1$$

$$\text{میانگین زمان انتظار} = AWT = \frac{6+0+16+18+1}{5} = \frac{41}{5} = 8.2$$

$$\text{میانگین زمان اجرا} = AST = \frac{10+1+2+1+5}{5} = \frac{19}{5} = 3.8$$

$$\text{AVG Turnaround Time} = \text{AVG Service Time} + \text{AVG Waiting Time}$$

$$\text{میانگین زمان انتظار} + \text{میانگین زمان اجرا} = \text{میانگین زمان بازگشت}$$

$$12 = 3.8 + 8.2$$

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

$$\text{میانگین زمان بازگشت} = \text{میانگین زمان انتظار} + \text{میانگین زمان اجرا}$$

$$12 = 8.2 + 3.8$$

تست‌های فصل پنجم: مدیریت حافظه مجازی

۹۴- کدام الگوریتم جایگزینی صفحه از ناهنجاری Belady رنج می‌برد؟ (مهندسی IT - دولتی ۱۴۰۰)

LIFO (۴) MRU (۳)

FIFO (۲)

LRU (۱)

عنوان کتاب: سیستم عامل
مولف: ارسسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل پنجم: مدیریت حافظه مجازی

۹۴- 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 شناس مورد استفاده قرار گرفتن را به اندازه کافی در اختیار داشته و اکنون باید این شناس 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 بلافصله باید دوباره به حافظه آورده شود.

مثال: فرض کنید در سیستمی 3 قاب حافظه وجود دارد، اگر درخواست‌های زیر از چپ به راست، به این سیستم وارد شود، چند وقfe نقص صفحه رخ می‌دهد؟

4 3 2 1 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
1 قاب	4	4	4	1	1	1	5	5	5	5	5	5
2 قاب		3	3	3	4	4	4	4	4	2	2	2
3 قاب			2	2	2	3	3	3	3	3	1	1
وقfe خطای صفحه	x	x	x	x	x	x	x			x	x	

جمعاً 9 وقfe 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 می‌باید، اما در الگوریتم FIFO و در بعضی از الگوهای خاص ارجاع به صفحه‌ها، با افزایش تعداد قاب‌ها، تعداد وقfe‌های نقص صفحه نیز افزایش می‌یابد. این پدیده را ناهنجاری FIFO (FIFO Anomaly) یا ناهنجاری بی‌لیدی (Belady Anomaly) گویند.

برای روشن شدن قضیه، همان مثال قبل را این‌بار با 4 قاب حافظه بررسی می‌کنیم:

وروودی	4	3	2	1	4	3	5	4	3	2	1	5
قابل 1	4	4	4	4	4	4	5	5	5	5	1	1
قابل 2		3	3	3	3	3	3	4	4	4	4	5
قابل 3			2	2	2	2	2	2	3	3	3	3
قابل 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.

تست‌های فصل هفتم: مدیریت بن بست

۹۵ - یک کامپیوتر دارای ۶ چاپگر است و n پردازه در کامپیوتر برای به دست آوردن این چاپگرها رقابت می‌کنند. هر کدام از پردازه‌ها به ۳ چاپگر نیاز دارند. بیشترین مقدار n که تضمین نماید سیستم بدون بن بست است، چند است؟
(مهندسی IT - دولتی ۱۴۰۰)

- 1 (۱)
- 2 (۲)
- 3 (۳)
- 4 (۴)

عنوان کتاب: سیستم عامل
مولف: ارسسطو خلیلی فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل هفتم: مدیریت بن‌بست

۹۵ - 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:

$\sum_{i=1}^n \text{Request}[i] < m + n$ برای منابع n :

$$\rightarrow 3n < m + n \rightarrow 2n < m$$

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 حداقل r منبع نیازمند است. اگر فرآیند 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 بدین ترتیب 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:

$$n \times (r-1) < m$$

$$n \times r - n < m$$

$$n \times r < m + n$$

$$\sum_{i=1}^n \text{Request}[i] < m + n$$

$$\longrightarrow n \times 3 < 6 + n \rightarrow 2n < 6 \rightarrow n < 3 \rightarrow n = 2$$

تست‌های فصل اول: مفاهیم اولیه

۹۶- 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

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل اول: مفاهیم اولیه

۹۶- گزینه (۳) 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 واسطه 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)، محدودیت‌هایی وضع نشده است و از آنچاکه 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 تعريف 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 واسطه یا بلاک تابع است. استفاده‌کنندگان پیمانه‌ها (توابع) نیازی به دانستن جزئیات داخلی پیمانه‌ها (توابع) ندارند.

توجه: تعريف 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 مُدد کاربر به مُدد 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

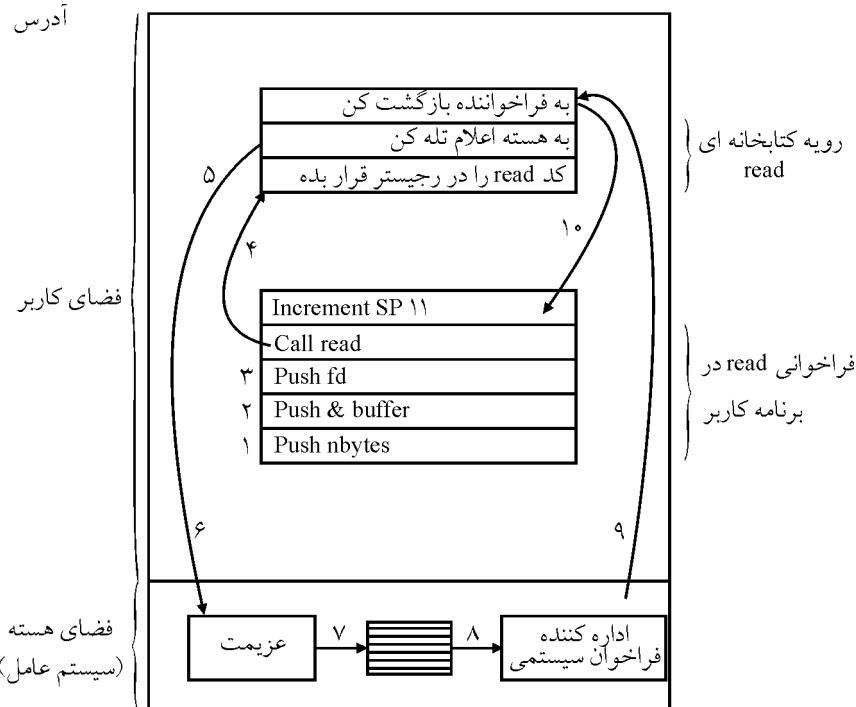
توسط برنامه اصلی (برنامه کاربر) انجام شود:

Count = Read (fd, buffer, nbytes);

تابع Read دارای سه پارامتر است، پارامتر اول مشخص کننده 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 برنامه‌نویسی C و C++ از پارامتر آخر به او 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 دستور العمل 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 کترول 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 کترول 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 باشد، مانند اینکه فراخوان سیستمی تلاش کند از صفحه کلید بخواند ولی هنوز چیزی از طرف کاربر تایپ نشده باشد، از آنجا فراخوان سیستمی متظاهر و مسدود شده داخل فرآیند جاری است بنابراین فرآیند جاری از طرف سیستم عامل به صفت مسدود و متظاهر منتقل می‌شود. در این شرایط سیستم عامل به صفت 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 صفت آماده سیستم عامل منتقل می‌شود و ممکن است در دفعه بعد توسط زمان‌بند کوتاه مدت سیستم عامل جهت

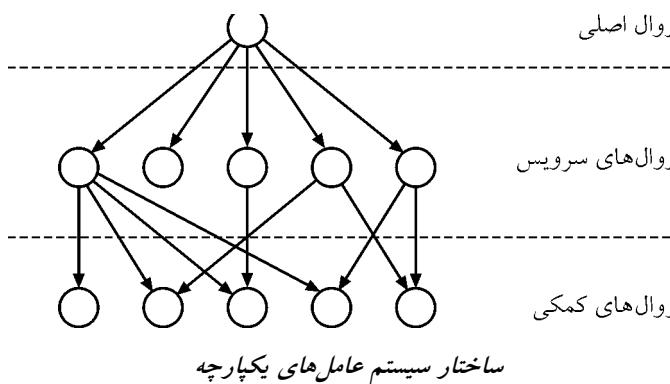
اجرا انتخاب شود و مراحل 9 تا 11 انجام می‌شود. (مرحله 9) سپس این رویه به روش بازگشت به برنامه کاربر بازمی‌گردد. (مرحله 10) برای خاتمه کار، برنامه کاربر پشته را پاک‌سازی می‌کند، همان‌گونه که پس از بازگشت از همه رویه‌ها این عمل صورت می‌گیرد. (مرحله 11)

آدرس



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 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:



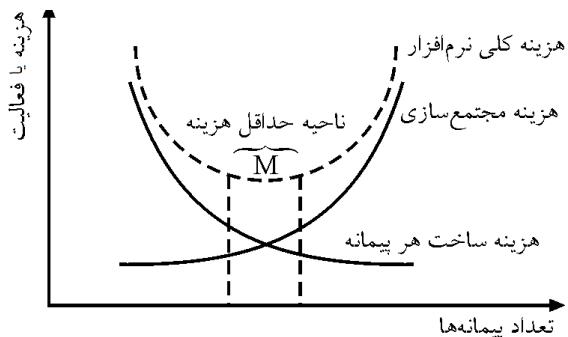
پیمانه‌ای کردن یعنی تقسیم نرم‌افزار (در اینجا خود برنامه سیستمی سیستم عامل) به چند مولفه جداگانه و متمایز که این مولفه‌ها به عنوان پیمانه‌هایی هستند که از اجتماع آن‌ها، خواسته‌های مساله برآورده می‌شود. با این عمل مدیریت مفهومی یک برنامه حاصل می‌شود و قابلیت خوانایی نرم‌افزار چندین برابر می‌گردد. درک 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.

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 نرم افزاری با افزایش تعداد پیمانه‌ها الزاماً 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.

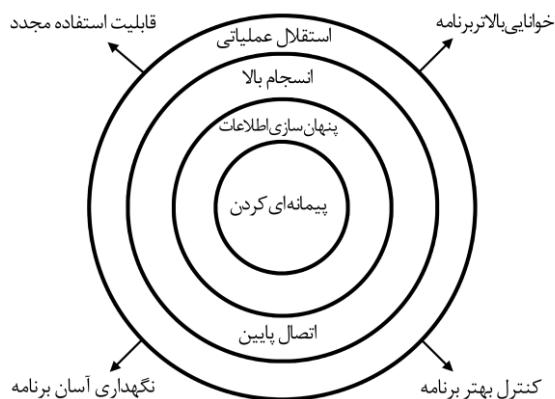
استقلال عملیاتی، نتیجه پیمانه‌ای کردن و پنهان‌سازی اطلاعات است. به بیان دیگر شرط لازم برای برقراری استقلال عملیاتی، پیمانه‌ای کردن و پنهان‌سازی اطلاعات است و شرط کافی برای برقراری استقلال عملیاتی انسجام بالا و اتصال پایین است. در صورتی که پیمانه‌هایی را با عملکرد تک منظوره (انسجام بالا) و عدم ارتباط 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 نرم افزار 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: پیمانه‌ای کردن، 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.

شكل 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 دکسترا و دانشجویانش در سال 1968 ساخته شد. این سیستم دارای شش لایه بود که در شکل زیر نمایش داده شده است:

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 وقهه 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، جایه‌جا 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 بافر کردن 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 پنجم (اپراتور): فرآیند اپراتور سیستم در این لایه قرار دارد.

مزیت‌های سیستم‌های لایه‌ای:

۱- طراحی پیمانه‌ای (Modularity) مناسب.

توجه: مزیت اصلی روش لایه‌ای، 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.

- 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 فرآخوانی‌ها 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 فرآخوانی‌ها می‌افزاید، در نتیجه یک فرآخوانی سیستمی در این ساختار، زمان بیشتری نسبت به ساختار غیر لایه‌ای صرف می‌کند. برای رفع این نقض 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، مسئله 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 کمی 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 3	هسته سیستم عامل 2	هسته سیستم عامل 1
ماشین مجازی		
سخت‌افزار		

مدل ماشین غیر مجازی

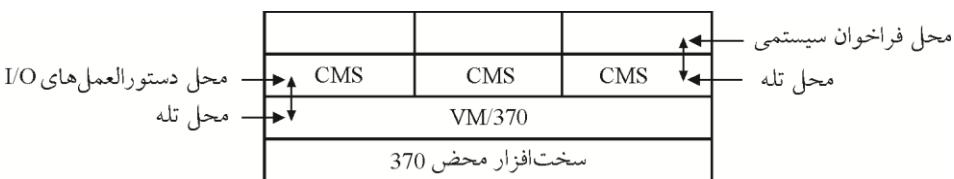
مدل ماشین مجازی

یک سیستم اشتراک زمانی دو وظیفه عمدی و مجزا دارد:

۱- مدیریت منابع (چندبرنامگی و مالتی پلکس کردن پردازنده و سایر منابع)

۲- مدیریت واسط کاربر (یک ماشین توسعه یافته با استفاده از یک واسط که بر روی سخت افزار خام قرار گیرد).

اما ایده ساختار ماشین مجازی این است که دو وظیفه فوق از یکدیگر جدا شوند، سیستم عامل VM/370 بر روی سیستم های IBM بهترین مثال از مفهوم ماشین مجازی است. قلب سیستم که به نام مانیتور ماشین مجازی (Virtual machine monitor) یا VMM یا Hypervisor معروف است بر روی سخت افزار عریان اجرا شده و فقط وظیفه اول یعنی مدیریت منابع (چندبرنامگی و مالتی پلکس کردن پردازنده و سایر منابع) را انجام می دهد و اکنون می توان 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 ماشین های مجازی برخلاف سیستم های عامل یک ماشین توسعه یافته نیستند بلکه هر کدام یک نسخه دقیق از سخت افزار خام هستند که شامل تمام خصوصیات سخت افزار مانند مد کاربر، مد هسته، I/O و وقفه می باشند. 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 می توان بر روی هر کدام از این ماشین های مجازی یک سیستم عامل معمولی نصب نمود، این عمل همان عمل نصب سیستم عامل بر روی یک سخت افزار خام حقیقی است. شکل زیر ساختار VM/370 را نشان می دهد.



توجه: هر کاربر یک برنامه (CMS: Conversational satalinal monitor system) مخصوص به خود را دارد که یک سیستم عامل تک کاربره محاوره ای است.

توجه: سیستم عامل هایی که بر روی ماشین های مجازی اجرا می شوند، هیچ گونه ارتباطی با یکدیگر ندارند، در واقع هر کدام خود را در یک سیستم مجزا تصور می کنند بنابراین می توان 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 اینظر تصور کند. حال سوال این است که این استقلال چگونه بر روی یک سخت افزار مشترک پیاده سازی می شود؟ جواب تسهیم (Multiplexing) و Slooping می باشد.

در واقع کافی است منابع سیستم با توجه به خصوصیاتش به دو صورت زمانی یا فضایی بین ماشین های مجازی تسهیم و یا Spool شوند. به عنوان مثال:

- ایجاد پردازنده مجازی با تسهیم زمانی پردازنده حقیقی که این عمل به کمک PCB های مجازی

انجام می شود.

- ایجاد حافظه های مجازی با تسهیم فضای حافظه اصلی بین ماشین های مجازی.
- کردن دستگاه های ورودی و خروجی Spool.
- تشکیل دیسک مجازی با تسهیم فضای دیسک حقیقی.

همانطور که مشخص است دستیابی به سخت افزار و فرآخوان های سیستمی نمی تواند بر روی CMS اجرا شود و در واقع CMS قادر است این کار را ندارد، پس برنامه اجرا شده در CMS چگونه فرآخوانی سیستمی انجام می دهد؟ فرض کنید برنامه ای که روی CMS اجرا می شود قصد انجام عمل I/O دارد و یک فرآخوانی سیستمی صادر می کند. این فرآخوانی سیستمی باعث تغییر مدد کاربر به مد هسته و می شود و سیستم عامل روی CMS (از آنجا که نمی داند CMS سخت افزار حقیقی نیست) سعی می کند که این عمل را روی CMS اجرا کند و در واقع یک تله مجازی در CMS رخ می دهد، اکنون CMS که در حقیقت یک نرم افزار است، این درخواست را به VM/370 ارجاع می دهد و اکنون یک تله واقعی در VM/370 رخ می دهد و درخواست (با توجه به تسهیم با Spool) به اجرا رسیده و پاسخ به لایه های بالاتر ارجاع می شود. از آنجا که وظیفه چندبرنامگی و ارائه ماشین توسعه یافته کاملاً مجزا از یکدیگر هستند، هر یک از این تکه برنامه ها بسیار ساده تر شده و از انعطاف پذیری (Flexibility) بیشتر و قابلیت نگهداری (Maintainability) آسان تری برخوردار هستند.

توجه: در ساختار ماشین مجازی دو وظیفه اصلی چندبرنامگی و ایجاد واسط راحت (مستقل از سخت افزار) از یکدیگر مجزا شده اند. 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 بخش ها ساده تر شده و از قابلیت انعطاف بیشتری برخوردارند.

توجه: از آنجا که هر ماشین مجازی کاملاً مشابه سخت افزار واقعی است، هر یک از آنها می توانند هر سیستم عاملی را مستقل اجرا کنند لذا می توان همزمان سیستم عامل های مختلفی را روی این ماشین اجرا کرد.

یکی دیگر از کاربردهای ماشین های مجازی به مجازی سازی محیط برنامه نویسی Programming-environment Virtualization (Virtualization) موسوم است، در این روش VMM ها سخت افزار واقعی را جهت اجرای سیستم عامل های متعدد نسخه برداری و مجازی سازی نمی کنند، بلکه هدف ایجاد قابلیت حمل (Portability) برنامه ها بر روی سخت افزار و سیستم عامل های مختلف است. کامپایلر زبان java که توسط شرکت Sun Microsystems طراحی شده است، یک خروجی بایت کد (byte code) تولید می کند. این بایت کدها دستوراتی هستند که بر روی ماشین مجازی جاوا (JVM) اجرا می شوند. جهت اجرای برنامه های java در یک ماشین، آن کامپیوتر می بایست دارای یک JVM باشد. بدین ترتیب برنامه هایی که به زبان java نوشته شده اند به راحتی بر روی انواع کامپیوترها اجرا می شوند. 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 عامل‌ها 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 (Probability) بالای 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 فرایند‌ها و زمانبندی پایه‌ای را به هسته واگذار می‌کنند و دیگر خدمات سیستم عامل توسط تعدادی فرایند 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 ایده همراه بوده است که تا جایی که ممکن است کدها به لایه‌های بالاتر منتقل شوند تا نهایتاً یک هسته کمینه پدید آید. برای این منظور اکثر وظایف سیستم عامل را در سطح کاربر و مشابه فرایند‌های 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

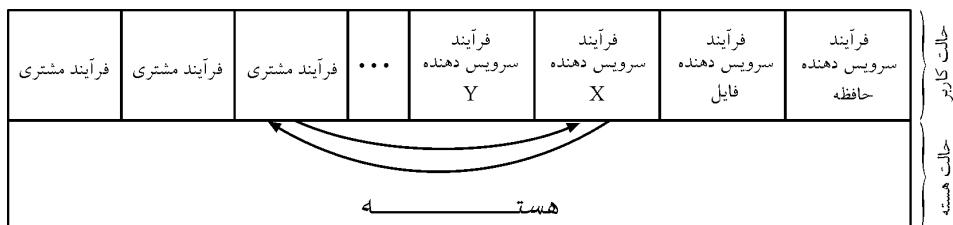
۳- بخش سطح پایین مدیریت حافظه مانند برنامه‌ریزی ثبات‌های سخت‌افزار مدیریت حافظه

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.

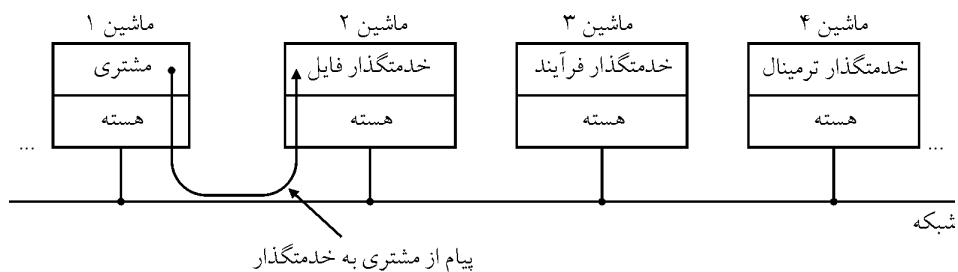
توجه: سرویس‌دهنده‌ها در مُد کاربر اجرا شده و ریز هسته با آنها همانند فرآیندهای کاربر رفتار می‌کند. بنابراین هیچ یک از سرویس‌دهنده‌ها دسترسی مستقیم به سخت‌افزار را ندارد. همچنین اگر کارکرد یکی از سرویس‌دهنده‌ها مختلط شود، فقط همان بخش مختلط می‌شود و این اختلال به دلیل استقلال عملیاتی هر سرویس‌دهنده به سایر سرویس‌دهنده‌ها و به تبع کل سیستم منتقل نمی‌شود.

توجه: نحوه‌ی 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 مجرماً از سرویس‌دهنده اجرا شده است، در این حالت همانطور که در مدل مشتری و سرویس‌دهنده ماشین محلی، یک پیام درخواست از مشتری به سرویس‌دهنده از طریق هسته ارسال می‌شود 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:

مهمنترین 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

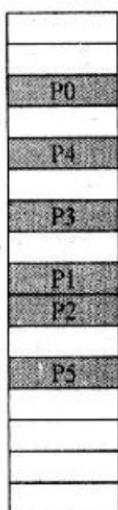
معایب سیستم‌های مشتری و سرویس‌دهنده:

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

سرویس دهنده باعث وقوع 2 تله می شود و پاسخ آن نیز به همین صورت، بنابراین برای هر درخواست و پاسخ 4 تله رخ می دهد، 2 فراخوان سیستمی برای ارسال و دریافت «پیام درخواست» و همچنین 2 فراخوان سیستمی برای ارسال و دریافت «پیام پاسخ». همچنین در موازات آن فراخوانی های سیستمی برای مدیریت حافظه و I/O نیز رخ می دهد که کارایی را تا حد زیادی کاهش می دهد.

تست‌های فصل چهارم: مدیریت حافظه اصلی

۹۷- 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 زیر، 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 کامپیووتر - دولتی ۱۴۰۰)

17240 (۴)

13144 (۳)

9048 (۲)

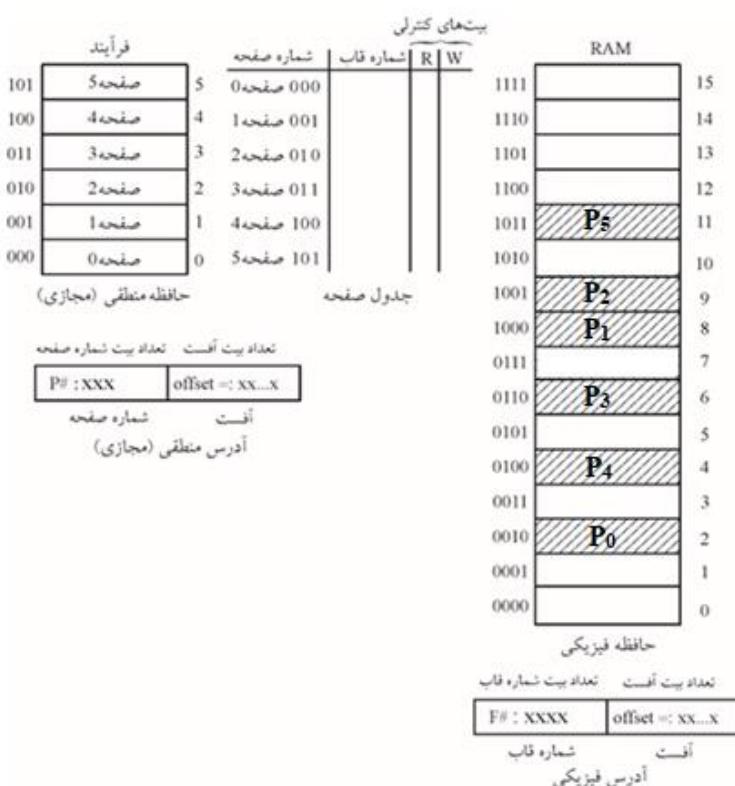
7000 (۱)

عنوان کتاب: سیستم عامل
مولف: ارسسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل چهارم: مدیریت حافظه اصلی

۹۷- گزینه (۳) صحیح است.

توجه: مطابق اطلاعات مساله و شکل حافظه اصلی صورت سوال، شکل زیر گویای مطلب است:



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_5 تا P_0) 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:

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

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

$$\text{تعداد صفحات فرآیند} = \log_2 b = \lceil \log_2^6 \rceil = 3\text{bit}$$

$$\text{تعداد قاب‌های حافظه فیزیکی} = \log_2 b = \log_2^{16} = 4\text{bit}$$

$$\text{اندازه صفحه یا اندازه قاب} = \log_2 b = \log_2^{2048} = 11\text{bit}$$

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

$$\text{تعداد بیت آفس} + 11\text{bit} = 14\text{bit}$$

$$\text{تعداد بیت آفس} + \text{تعداد بیت شماره صفحه} = \text{طول آدرس منطقی}$$

$$4\text{bit} + 11\text{bit} = 15\text{bit}$$

همچنین داریم:

$$\frac{\text{تعداد بیت آفس}}{\text{تعداد بیت شماره صفحه}} = 2 \quad \frac{\text{تعداد بیت آفس}}{\text{اندازه حافظه منطقی (فرآیند)}} = 2 \times 2$$

$$2^{14} = 2^3 \times 2^{11} = 16\text{KB}$$

$$\frac{\text{تعداد بیت آفس}}{\text{تعداد بیت شماره قاب}} = 2 \quad \frac{\text{تعداد بیت آفس}}{\text{اندازه حافظه فیزیکی (RAM)}} = 2 \times 2$$

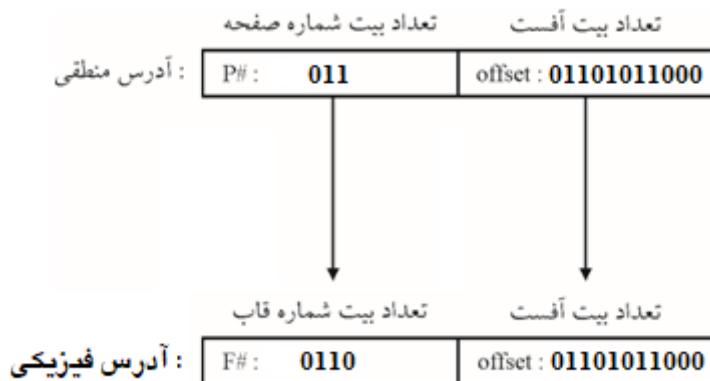
$$2^{15} = 2^4 \times 2^{11} = 32\text{KB}$$

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 کرده است، با این‌ری 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

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 011,01101011000 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 شده است.

توجه: باینری آدرس 7000 برابر 14 بیت به عنوان طول کل آدرس منطقی به صورت 011,01101011000 است. از آنجا که فرآیند موجود در این سیستم شامل 6 صفحه است، بنابراین 3 بیت سمت چپ (b) 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 ($b = \log_2^{16} = 4\text{bit}$) 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 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

باينري آدرس فیزیکی به صورت زير 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:

$$(01101011000)_2 = 512 + 256 + 64 + 16 + 8 = (856)_{10}$$

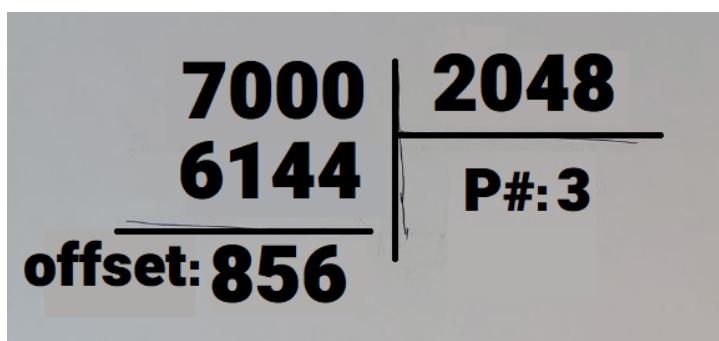
حاصل جمع دسیمال شماره قاب و آفست برابر دسیمال آدرس فیزیکی است، به صورت زیر:

$$12288 + 856 = 13144$$

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 امین قاب

حافظه اصلی، بنابراین 6 قاب قبل از قاب شماره 6 (7 امین 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 پُر واضح است که گزینه سوم پاسخ سوال است.

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۹۸ - برنامه زیر از ۳ پردازه هم‌روند تشکیل شده است. و این ۳ پردازه از ۳ سماور بازنی که به صورت زیر مقداردهی شده‌اند، استفاده می‌کنند. دستور print ('HELLO') چند بار اجرا می‌شود؟

$S_0=1, S_1=0, S_2=0$

(مهندسی IT – دولتی ۱۴۰۰)

Proccess P₀:

```
while (true) {  
    wait(S0);  
    print ('HELLO');  
    signal (S1);  
    signal (S2);  
}
```

Proccess P₁:

```
wait(S1);  
signal (S0);
```

Proccess P₂:

```
wait(S2);  
signal (S0);
```

۱) دقیقاً ۲ بار

۲) حداقل ۲ بار

۳) دقیقاً ۳ بار

عنوان کتاب: سیستم عامل
مولف: ارسسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۹۸ - گزینه (۴) صحیح است.

راه حل سمافور در سال ۱۹۶۵ توسط Dijkstra پیشنهاد شده است.
ساختار کلی این راه حل به صورت زیر می‌باشد:

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

توجه: از عبارت mutex گرفته شده است.

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

توجه: در مقاله Dijkstra، از نام‌های P و V (حرف اول کلمات آلمانی تست "prober" 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 گاه‌آبجای 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 نحوه 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 (s) **signal**: عملیات آن به ترتیب شامل تست خالی بودن صف، افزایش مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

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

توجه: در سمافور دودویی، شمارنده سمافور، هیچگاه منفی نمی‌شود و در هر شرایطی فقط می‌تواند دو مقدار بایزی 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 (s) **signal** 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، $S_0 = 1$ و $S_1 = 0$ و $S_2 = 0$ ، فرآیند P_0 ابتدا اجرا می‌شود.

توجه: فرآیندهای P_1 و P_2 قادر 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 $\text{wait}(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $\text{release}(S_1)$ و $\text{release}(S_2)$ 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 $\text{release}(S_0)$ 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 $\text{wait}(S_0)$ ، با سرنوشتی که دارد، برای همیشه می‌خوابد.

دقت کنید، اجرای دو دستور $\text{release}(S_1)$ و $\text{release}(S_2)$ 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 $\text{wait}(S_0)$ را انجام می‌دهد و در ادامه اولین '0' را چاپ می‌کند. و با دو عمل $\text{release}(S_1)$ و $\text{release}(S_2)$ 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 $\text{release}(S_0)$ 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 $\text{release}(S_1)$ و $\text{release}(S_2)$ 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 حداقل 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 شанс 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.

تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۹۹- فرض کنید زمان رسیدن و مدت زمان اجرای سه پردازه به صورت زیر باشد:

بیشینه زمان چرخش (Turnaround Time) برای الگوریتم SJF (Shortest Job First) بدون قبضه (non preemptive) کدام است؟
(مهندسی IT - دولتی ۱۴۰۰)

Process	Arrival Time	Burst Time
P1	1	14
P2	3	7
P3	2	7

31 ms (۴)

28 ms (۳)

21 ms (۲)

14 ms (۱)

عنوان کتاب: سیستم عامل

مؤلف: ارسسطو خلیلی‌فر

ناشر: انتشارات راهیان ارشد

آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنه

۹۹- 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 «غیرقابل پس گرفتن» یا «غیرقابل تخلیه پیش هنگام» نیز گفته می‌شود.

توجه: یک نقص عمدۀ الگوریتم SJF این است که ممکن است باعث قحطی زدگی 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.

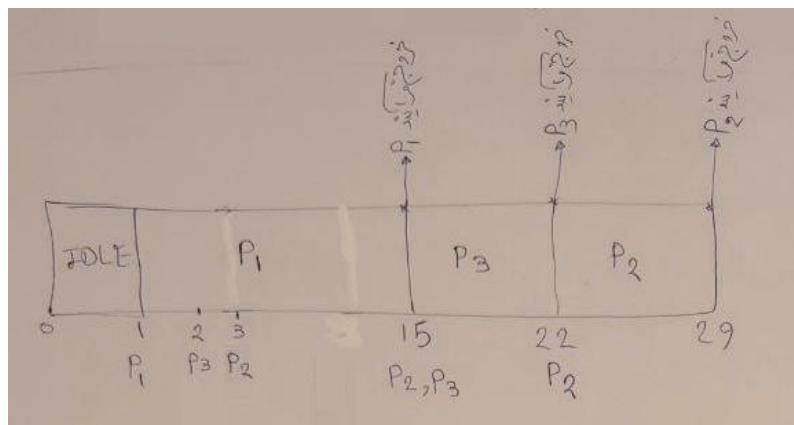
توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسخ و میانگین زمان گردش کار (بازگشت) فرآیندهاست.

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.

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 ۱۵، زمان باقی‌مانده فرآیند P₂ و P₃ هر دو برابر ۷ واحد زمانی است که به دلیل ورود زودتر فرآیند P₃ پردازنده به همان فرآیند P₃ اختصاص می‌یابد. که البته بهتر بود طراح محترم فرض این شرایط را بیان می‌کرد.

زمان ورود فرآیند - زمان خروج کامل فرآیند = زمان بازگشت فرآیند

$$P_1 = 15 - 1 = 14$$

$$P_2 = 29 - 3 = 26$$

$$P_3 = 22 - 2 = 20$$

$$\text{ATT} = \frac{14+26+20}{3} = \frac{60}{3} = 20 \quad \text{میانگین زمان بازگشت}$$

زمان اجرای فرآیند - زمان بازگشت فرآیند = زمان انتظار فرآیند

$$P_1 = 14 - 14 = 0$$

$$P_2 = 26 - 7 = 19$$

$$P_3 = 20 - 7 = 13$$

$$M\text{یانگین زمان انتظار} = AWT = \frac{0+19+13}{3} = \frac{32}{3} = 10.67$$

$$M\text{یانگین زمان اجرا} = AST = \frac{14+7+7}{3} = \frac{28}{3} = 9.33$$

$$AVG\text{ Turnaround Time} = AVG\text{ Service Time} + AVG\text{ Waiting Time}$$

$$M\text{یانگین زمان انتظار} + M\text{یانگین زمان اجرا} = M\text{یانگین زمان بازگشت}$$

$$20 = 9.33 + 10.67$$

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

$$M\text{یانگین زمان بازگشت} = M\text{یانگین زمان انتظار} + M\text{یانگین زمان اجرا}$$

$$20 = 9.33 + 10.67$$

توجه: بیشینه زمان بازگشت یا چرخش (Turnaround Time) برابر 26 و مربوط به فرآیند P₂ 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 کلیدنهایی این سوال حذف گردید، که کار درستی بوده است.

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۱۰۰- الگوریتم زیر که تغییر یافته الگوریتم پرسون برای حل ناحیه بحرانی است برای دو فرآیند i و j ارائه شده است. (مشابه آن برای i وجود دارد که اندیس‌های i و j تعویض می‌گردد). کدام گزینه درست است؟ (فرض کنید که الگوریتم پرسون 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) شرط لازم ناحیه بحرانی را دارد و راه حل مناسب است.

پاسخ تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۱۰۰- 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:

$P_0:$ { (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 */ }
--	--

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: هنانام این آزمون‌ها رابه عنوان مبدع آن «قوانين ارسسطو» نام‌گذاری کردیم، این قوانین به «قوانين چهارگانه ارسسطو» نیز موسوم است.

قانون اول ارسسطو (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 P₀ 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 بدیم، خب قرار دادیم. حال در ادامه آزمون اول وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P₀ 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 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P₁:

(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 P₁ قرار می‌گیرد، به صورت زیر:

P₁:

/*critical section*/

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

قانون دوم ارسسطو(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) 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 */

}

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

P_1 :

(1) turn = 1;

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه بحرانی را دارند، متغیر turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآیند، شماره‌ی فرآیند خود را در متغیر turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیره کند، فرآیندی است که شماره‌اش در متغیر turn باقی می‌ماند و دیگری اثراش در متغیر turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغیر turn گره خورده است، بنابراین در این مسئله 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 مدامی که کوانتم 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 بدھیم 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 ارسسطو (آزمون اول شرط انحصار متقابل) و قانون دوم ارسسطو (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشند. بنابراین شرط انحصار متقابل در سوال مطرح شده برقرار نیست.

قانون سوم ارسسطو (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 مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجدداً 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:

P_0 :

```
{
(1) turn = 0;
(2) while (turn == 1);
/* critical section */
(3) turn = 1;
/* remainder section */
}
```

P_1 :

```
{
(1) turn = 1;
(2) while (turn == 0);
/* critical section */
(3) turn = 0;
/* remainder 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.

(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 بدھیم، 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*/

(3) turn = 1;

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 بدھیم، 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 بدھید.

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 بدھیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۴) می‌شویم. هم اکنون پردازندۀ 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 بدھیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۵) می‌شویم. هم اکنون پردازندۀ داخل ناحیه باقی‌مانده فرآیند P_1 مشغول حرکت است.

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

قانون چهارم ارسسطو (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 برگرد و مجدداً تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجدداً وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.

```
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 */  
}
```

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:
فرض کنید فرآیند P₀ 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 P₀ قرار می‌گیرد، به صورت زیر:

```
P0:  
/*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:

```
P1:  
(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 P₁ قرار می‌گیرد، به صورت زیر:

```
P1:
```

/*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 ارسسطو(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 بن‌بست 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 متظر 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 بدھیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن‌بست رخ داده است و شرط انتظار محدود نقض شده است. خب همانطور که در آزمون دوم دیدیم 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 ارسسطو (آزمون بن‌بست) و قانون چهارم ارسسطو(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.