موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

سیستم عامل

(مدیریت حافظه مجازی)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

آبراهام سیلبرشاتز، ویلیام استالینگز و اندرو اس تنن‌بام

ارسطو خلیلی‌فر

۵ حافظه مجازی

حافظه مجازی (Virtual Memory)

در همه روش‌های پیشینِ مدیریت حافظه، یک فرآیند تنها زمانی می‌توانست اجرا گردد که تمام فرآیند بتواند در آن واحد در حافظه حاضر باشد. در این حالت اگر فرآیند قدری بزرگ باشد، پیدا کردن فضای خالی مناسب برای آن مشکل خواهد بود.

ایده اصلی حافظه مجازی این است که حتی اگر به اندازه کافی فضای خالی بر روی حافظه در اختیار نداشته باشیم، به یک فرآیند اجازه اجرا بدهیم. نکته اصلی این است که یک فرآیند در آنِ واحد به همه داده‌ها و کد خود نیاز ندارد بلکه در هر لحظه فقط به بخشی از داده و قسمتی از کد نیازمند است. در تکنیک حافظه مجازی فقط قسمتی از فرآیند به حافظه آورده می‌شود که در حال حاضر به آن نیاز است و مابقی فرآیند می‌تواند کماکان بر روی دیسک قرار گیرد و در طول اجرای یک فرآیند این جابه‌جایی‌ها بین حافظه و دیسک مرتباً صورت گیرد. با این کار می‌توانیم فرآیندهای بیشتری را در داخل حافظه‌ی اصلی نگهداری کنیم.

اگر این تکنیک از دید فرآیند مخفی بماند، فرآیند گمان می‌کند تمام فضای درخواستی وی به او تخصیص داده شده است، در صورتی که چنین نیست و در واقع سیستم عامل با تلاشی مضاعف و با انجام جابه‌جایی‌های پی در پی این دید را برای فرآیند ایجاد کرده است.

نکته: با توجه به مسائل مطرح شده، مجموع کلیه فضای آدرس فرآیندهایی که در حال اجرا هستند، می‌تواند از اندازه حافظه فیزیکی بیشتر شود و **این یعنی حافظه مجازی**. در واقع همه فرآیندها گمان

می‌کنند به طور کامل در حافظه قرار دارند، غافل از اینکه قسمت اعظم هر یک از آنها بر روی دیسک است.

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

۴ ـ ۱ ـ ۱ ـ صفحه‌بندی برحسب نیاز (Demand Paging)

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

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

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

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

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

الگوریتم‌های جایگزینی صفحه (Page Replacement)

هنگامی که یک وقفه نقص صفحه رخ می‌دهد سیستم عامل باید یکی از صفحات را از حافظه بیرون ببرد تا جا برای صفحه جدید باز شود. اگر صفحه قدیمی در مدت زمانی که در حافظه بوده، تغییر کرده باشد، باید محتویات آن در دیسک نوشته شود تا تغییرات از دست نرود، اما اگر تغییر نکرده باشد، کپی موجود بر روی دیسک همچنان معتبر است و نیازی به نوشتن محتویات صفحه بر روی دیسک نیست و صفحه جدید به سادگی بر روی صفحه قدیمی نوشته می‌شود.

هنگامی که نقص صفحه رخ می‌دهد، می‌توان هر صفحه‌ای را برای خروج از حافظه انتخاب کرد، اما برای مثال اگر صفحه‌ای انتخاب شود که زیاد مورد استفاده قرار می‌گیرد به احتمال زیاد کمی بعد باید دوباره آن را به حافظه برگردانیم و این یعنی تصمیم اشتباه.

برای جایگزینی صفحه الگوریتم‌های زیادی وجود دارند که ایده‌ها و کارآیی متفاوتی دارند. تعدادی از آنها را بررسی می‌کنیم:

الگوریتم FIFO (First In First Out)

این الگوریتم ساده‌ترین الگوریتم از نظر پیاده‌سازی است. در این روش سیستم عامل لیستی از صفحات را به ترتیب ورود به حافظه نگه می‌دارد. وقتی یک خطای نقص صفحه رخ می‌دهد، سیستم عامل **قدیمی‌ترین** صفحه را برای بیرون رفتن انتخاب می‌کند. ایده این روش این است که قدیمی‌ترین صفحه شانس مورد استفاده قرار گرفتن را به اندازه کافی در اختیار داشته و اکنون باید این شناس به صفحه دیگری داده شود.

نکته: نقص الگوریتم FIFO این است که حتی اگر صفحه‌ای بارها و به طور مکرر استفاده شود، سرانجام به قدیمی‌ترین صفحه تبدیل و حذف می‌شود، در صورتی که احتمالاً بلافاصله باید دوباره به حافظه آورده شود.

مثال: فرض کنید در سیستمی ۳ قاب حافظه وجود دارد، اگر درخواست‌های زیر از چپ به راست، به این سیستم وارد شود، چند وقفه نقص صفحه رخ می‌دهد؟ ۵، ۱، ۲، ۳، ۴، ۵، ۳، ۴، ۱، ۲، ۳، ۴

حل: فرض می‌کنیم در ابتدا هر ۳ قاب خالی هستند، با جدول زیر تعداد نقص صفحه را به دست می‌آوریم:

ورودی	۴	۳	۲	۱	۴	۳	۵	۴	۳	۲	۱	۵
قاب ۱	۴	۴	۴	۱	۱	۱	۵	۵	۵	۵	۵	۵
قاب ۲		۳	۳	۳	۴	۴	۴	۴	۴	۲	۲	۲
قاب ۳			۲	۲	۲	۳	۳	۳	۳	۳	۱	۱
وقفه خطای صفحه	×	×	×	×	×	×	×			×	×	

جمعاً ۹ وقفه خطای صفحه رخ می‌دهد.

نکته: در نگاه اول به نظر می‌رسد با افزایش قاب‌هایی از حافظه که در اختیار یک فرآیند است، تعداد نقص صفحه‌ها همواره کاهش می‌یابد، اما در الگوریتم FIFO و در بعضی الگوهای خاصِ ارجاع به صفحه‌ها، با افزایش تعداد قاب‌ها، تعداد وقفه‌های نقص صفحه نیز افزایش می‌یابد. این پدیده را **ناهنجاری FIFO** (FIFO Anomaly) و یا **ناهنجاری بی‌لیدی** (Belady Anomaly) گویند.

برای روشن شدن قضیه، همان مثال قبل را این بار با ۴ قاب حافظه بررسی می‌کنیم:

ورودی	۴	۳	۲	۱	۴	۳	۵	۴	۳	۲	۱	۵
قاب ۱	۴	۴	۴	۴	۴	۴	۵	۵	۵	۵	۱	۱
قاب ۲		۳	۳	۳	۳	۳	۳	۴	۴	۴	۴	۵
قاب ۳			۲	۲	۲	۲	۲	۲	۳	۳	۳	۳
قاب ۴				۱	۱	۱	۱	۱	۱	۲	۲	۲
وقفه خطای صفحه	×	×	×	×			×	×	×	×	×	×

مشاهده می‌کنیم با همان دنباله ارجاع و با ۴ قاب، تعداد نقص صفحه‌ای که رخ می‌دهد به ۱۰ می‌رسد. البته اگر تعداد قاب‌ها را به ۵ افزایش دهیم تعداد نقص صفحه در این مثال یکباره به ۵ نقص صفحه کاهش می‌یابد.

الگوریتم بهینه (Optimal)

اساس کار این الگوریتم کاملاً منطقی است. در این الگوریتم صفحه‌ای باید برای ترک حافظه انتخاب شود که در آینده، دیرتر از همه به آن نیاز پیدا می‌کنیم. به عنوان مثال از بین دو صفحه A و B، صفحه A را تا ۲ میلیون دستور دیگر و صفحه B را تا ۳ میلیون دستور دیگر نیاز نداریم. بنابراین کاملاً منطقی است که صفحه B را از حافظه خارج کنیم تا خطاهای نقص صفحه را تا حد امکان به تأخیر بیندازیم.

نکته: الگوریتم **بهینه** ناهنجاری بی‌لیدی **ندارد**.

نکته: با اجرای این الگوریتم کمترین تعداد خطای نقص صفحه رخ می‌دهد.

مثال: حافظه‌ای را با ۳ قاب در نظر بگیرید. با استفاده از روش بهینه و برای دنباله ارجاعات زیر چند خطای نقص صفحه رخ می‌دهد؟ ۱، ۴، ۳، ۵، ۲، ۱، ۳، ۴، ۱، ۲، ۳

ورودی	۳	۲	۱	۴	۳	۱	۲	۵	۳	۴	۱
قاب ۱	۳	۳	۳	۳	۳	۳	۳	۳	۳	۳	۱
قاب ۲		۲	۲	۴	۴	۴	۴	۴	۴	۴	۴
قاب ۳			۱	۱	۱	۱	۲	۵	۵	۵	۵
نقص صفحه	×	×	×	×			×	×			×

در این مثال جمعاً ۷ خطای نقص صفحه رخ می‌دهد.

نکته: همان‌گونه که ذکر شد الگوریتم بهینه کمترین تعداد نقص صفحه ممکن را باعث می‌شود، اما نقص عمده آن این است که در عمل قابل پیاده‌سازی نیست. زیرا سیستم عامل باید بداند در آینده چه صفحه‌هایی و به چه ترتیبی موردنیاز هستند که این مسئله در عمل غیرممکن است.

نکته: از الگوریتم بهینه فقط می‌توان برای ارزیابی دیگر الگوریتم‌ها استفاده کرد. از آنجا که این الگوریتم کمترین خطای نقص صفحه را باعث می‌شود، الگوریتم‌های قابل پیاده‌سازی را با این الگوریتم مقایسه می‌کنند.

الگوریتم LRU (Last Recently Used)

ایده اصلی این الگوریتم این است که اگر صفحه‌ای در چند دستور اخیر مراجعات زیادی داشته است، به احتمال قوی در دستورات بعدی هم ارجاعات زیادی خواهد داشت، همچنین اگر یک صفحه، اخیراً هیچ مراجعه‌ای نداشته، احتمالاً در آینده نزدیک هم ارجاعی نخواهد داشت.

در واقع این الگوریتم بیان می‌کند هنگام وقوع خطای نقص صفحه، صفحه‌ای را حذف کنید که طولانی‌ترین زمان عدم استفاده را دارد.

می‌توان گفت LRU تقریبی از الگوریتم بهینه می‌باشد که در آن به جای توجه به آینده، به گذشته توجه می‌شود.

نکته: الگوریتم **LRU** ناهنجاری بی‌لیدی **ندارد**.

مثال: حافظه‌ای را با ۳ قاب صفحه در نظر بگیرید، با استفاده از روش LRU **و برای دنباله ارجاعات زیر، چند خطای نقص صفحه رخ می‌دهد؟** ۲، ۳، ۱، ۵، ۳، ۱، ۴، ۲، ۴، ۳، ۴، ۲، ۱، ۳

ورودی	۳	۱	۲	۴	۳	۴	۲	۴	۱	۳	۵	۱	۳	۲
قاب ۱	۳	۳	۳	۴	۴	۴	۴	۴	۴	۴	۵	۵	۵	۲
قاب ۲		۱	۱	۱	۳	۳	۳	۳	۱	۱	۱	۱	۱	۱
قاب ۳			۲	۲	۲	۲	۲	۲	۲	۳	۳	۳	۳	۳
نقص صفحه	×	×	×	×	×				×	×	×			×

جمعاً ۹ خطای صفحه رخ می‌دهد.

نکته: الگوریتم LRU در عمل قابل پیاده‌سازی می‌باشد اما هزینه پیاده‌سازی آن قدری بالاست. در این روش به یک لیست از تمامی صفحات حافظه نیاز داریم که در آن صفحات به ترتیبِ ارجاعات اخیر آنها مرتب شده باشند، در واقع این لیست باید در هر ارجاع به حافظه به روز شود. در عمل الگوریتم LRU را جهت افزایش سرعت، به صورت سخت‌افزاری پیاده‌سازی می‌کنند. برای پیاده‌سازی LRU

روش‌های مختلفی پیشنهاد شده است. مانند پشته، استفاده از شمارنده و نوعی ماتریس دوبعدی. برای بررسی این روش‌ها می‌توان به کتاب پروفسور تنن‌باوم رجوع کرد.

الگوریتم دومین شانس (Second Chance)

این الگوریتم از خانواده الگوریتم FIFO می‌باشد. این ایده موجب می‌شود در الگوریتم FIFO، صفحاتی که زیاد استفاده شده‌اند از حافظه خارج نشوند. در این الگوریتم به ازای هر صفحه در جدول صفحه، یک بیت با عنوان R در نظر می‌گیریم (حرف R از ابتدای کلمه Referenced گرفته شده است). اگر به یک صفحه موجود در حافظه ارجاع شود، بیت R آن صفحه یک می‌شود.

در این روش برای حذف، باز هم به سراغ قدیمی‌ترین صفحه می‌رویم (همانند FIFO)، اما قبل از حذف قدیمی‌ترین صفحه، بیت R آن را چک می‌کنیم، اگر این بیت صفر باشد به این معناست که این صفحه هم قدیمی است و هم ارجاعی به آن صورت نگرفته است، بنابراین با خیالی آسوده آن را حذف می‌کنیم. اما اگر بیت R آن یک باشد، سیستم عامل بیت R آن را صفر کرده و این صفحه را به انتهای صف می‌فرستد. در واقع با این کار به این صفحه یک شانس دیگر داده می‌شود تا در حافظه باقی بماند.

نکته: الگوریتم **دومین شانس**، ناهنجاری بی‌لیدی **دارد**.

نکته: اگر هنگام یک نقص صفحه، بیت R مربوط به همه صفحات یک باشد، این الگوریتم همانند FIFO عمل می‌کند (البته با قدری اتلاف وقت) زیرا در حالتی که بیت R همه صفحه‌ها یک باشد، این الگوریتم تمام صفحه‌ها را به انتهای لیست برده و بیت R آنها را صفر می‌کند و دوباره قدیمی‌ترین صفحه در ابتدای لیست قرار می‌گیرد اما این بار با بیت R برابر صفر.

نکته: دقت کنید در الگوریتم دومین شانس، اگر صفحه‌ای زیاد مورد ارجاع واقع شود، این شانس را دارد که در حافظه باقی بماند. به عنوان مثال فرض کنید یک صفحه با بیت R=۱ به ابتدای صف برسد، در این حالت سیستم عامل بیت R آن را صفر کرده و این صفحه را به ابتدای صف منتقل می‌کند، حال اگر این صفحه طی مدت زمانی که دوباره به ابتدای صف نزدیک می‌شود باز هم ارجاع شود، بیت R آن مجدداً یک می‌شود و باز هم در حافظه باقی می‌ماند.

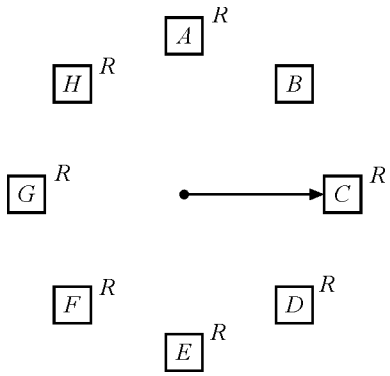
الگوریتم ساعت (Clock)

الگوریتم دومین شانس الگوریتم خوبی است اما پیوسته در حال جابه‌جا کردن صفحات در لیست است، این کار قدری کارآیی آن را کاهش می‌دهد.

یک ایده برای بهبود پیاده‌سازی الگوریتم دومین شانس، این است که ابتدا و انتهای لیست صفحات را به هم وصل کنیم. در واقع از یک لیست پیوندی حلقوی استفاده می‌کنیم که مانند یک ساعت عقربه‌ای عمل می‌کند. در این ساعت، عقربه همواره قدیمی‌ترین صفحه را نشان می‌دهد. هنگامی که یک خطای نقص صفحه رخ می‌دهد، سیستم عامل صفحه‌ای را که عقربه ساعت بر روی آن است بررسی می‌کند،

اگر بیت R آن صفر باشد، این صفحه حذف و عقربه روی صفحه بعدی برده می‌شود. اما اگر بیت R آن یک باشد، سیستم عامل فقط بیت R را صفر کرده و عقربه را یک صفحه جلو می‌برد.

نکته: دقت کنید الگوریتم ساعت همان الگوریتم دومین شانس است. در واقع می‌توان گفت یکی از راه‌های پیاده‌سازی الگوریتم دومین شانس، ایده ساعت می‌باشد.



مفهوم الگوریتم ساعت

مثال: حافظه‌ای با ۳ قاب آزاد را در نظر بگیرید، با استفاده از الگوریتم دومین شانس (ساعت) و با دنباله ارجاعات زیر چند نقص صفحه رخ می‌دهد؟

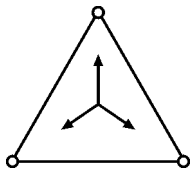
دنباله ارجاعات از چپ به راست: ۲، ۵، ۳، ۴، ۳، ۵، ۲، ۶، ۵، ۳، ۴، ۳، ۲، ۱

حل: فرض کنید اندیس R مشخص کننده مقدار ۱ برای بیت R می‌باشد.

ورودی	۱	۲	۳	۴	۳	۵	۶	۲	۵	۳	۴	۳	۵	۲
حافظه	۱ ↑	۱ ↑۲	۱ ↑۲ ۳	۴ ↘۲ ۳_R	۴ ↘۲ ۳_R	۴ ↙۵ ۳_R	۶ ↘۵ ۳	۶ ↙۲ ۳	۶ ↑۲ ۵	۳ ↘۲ ۵	۳ ↙۴ ۵	۳ ۳_R ↙۴ ۵	۳_R ↙۴ ۵_R	۳ ↙۴ ۵
نقص صفحه	*	*	*	*		*	*	*	*	*	*			*

جمعاً ۱۱ وقفه نقص صفحه رخ می‌دهد.

در این مثال در واقع ساعت و حالت‌های ممکن عقربه آن به صورت زیر است:



الگوریتم سالمندی (Aging)

در الگوریتم Aging به ازای هر صفحه یک شمارنده n بیتی (معمولاً ۸ بیتی) تعبیه می‌کنیم. در

پریودهای زمانی منظم، شمارنده همه صفحات یک بیت به سمت راست شیفت داده می‌شود و در واقع سمت راست‌ترین بیت از بین می‌رود و جای خالی بیت سمت چپ با بیت R هر صفحه پر می‌شود.

به عنوان مثال اگر شمارنه یک صفحه 00101101 و بیت R آن نیز یک باشد، بعد از وقوع وقفه تایمر، محتوای شمارنده این صفحه، 10010110 خواهد شد.

می‌توان گفت این شمارنده‌های ۸ بیتی، تاریخچه استفاده از هر صفحه را برای هشت پریود آخر نگهداری می‌کنند. سیستم عامل این شمارنده‌ها را به صورت اعداد بدون علامت در نظر می‌گیرد و هر چه شمارنده مربوط به یک صفحه، عدد بزرگ‌تری را نشان دهد، به این معناست که آن صفحه اخیراً بیشتر استفاده شده است و هر چه شمارنده عدد کوچکتری را نشان دهد، به این معناست که آن صفحه اخیراً کمتر استفاده شده است.

با این مقدمه، هنگام رخ دادن وقفه نقص صفحه، سیستم عامل صفحه‌ای را خارج می‌کند که شمارنده آن از همه کمتر باشد.

نکته: به عنوان مثال اگر شمارنده یک صفحه 0000 0000 باشد، به این معناست که این صفحه در ۸ پریود زمانی اخیر، اصلاً استفاده نشده است و اگر شمارنده یک صفحه 1111 1111 باشد، به این معناست که این صفحه در هر پریود حداقل یکبار مورد استفاده قرار گرفته است.

نکته: در این الگوریتم صفحه‌ای که شمارنده آن 10000001 می‌باشد، نسبت به صفحه‌ای با شمارنده 01111111 اولویت بیشتری جهت ماندن در حافظه دارد. در واقع صفحه دوم از حافظه خارج می‌شود زیرا از دید این الگوریتم صفحه اول در گذشته نزدیک‌تری به کار گرفته شده است.

نکته: هنگام وقوع وقفه نقص صفحه، ممکن است شمارنده چندین صفحه با هم برابر بوده و از همه کمتر باشند، در این حالت می‌توان بین آنها براساس الگوریتم FIFO یک صفحه را برای خروج انتخاب کرد.

نکته: الگوریتم Aging در واقع یک روش برای پیاده‌سازی ایده LRU می‌باشد.

الگوریتم NRU (Not Recently Used)

برای پیاده‌سازی این روش باید به ازای هر صفحه از دو بیت وضعیت استفاده کرد. این دو بیت عبارتند از بیت‌های M و R که به ترتیب از کلمات Modified و Referenced اقتباس شده‌اند. بیت R هنگامی برای یک صفحه Set می‌شود که به آن صفحه ارجاع شده باشد (چه این ارجاع برای خواندن باشد و چه برای نوشتن). اما بیت M هنگامی برای یک صفحه Set می‌شود که محتویات آن صفحه تغییر کرده باشد. در واقع در هر درایه جدول صفحه (به ازای هر صفحه) این دو بیت وجود دارند.

نحوه عملکرد الگوریتم NRU و شیوه استفاده از این دو بیت به صورت زیر است:

وقتی فرآیندی شروع به کار می‌کند، سیستم عامل بیت‌های R و M را به ازای همه صفحات آن با صفر پر می‌کند. هنگامی که به یک صفحه ارجاع می‌شود (فقط خواندن) بیت R آن صفحه یک می‌شود و هنگامی که یک صفحه تغییر کند (ارجاع جهت نوشتن) بیت M آن صفحه یک می‌شود. از طرفی برای اینکه بین صفحه‌ای که اخیراً به آن ارجاع شده و صفحه‌ای که قبلاً به آن ارجاع شده، تفاوت وجود داشته باشد، سیستم عامل در پریودهای زمانیِ منظم **همه** بیت‌های R را صفر می‌کند.

با اعمال این روش، هنگامی که یک نقص صفحه رخ می‌دهد، ابتدا سعی می‌شود صفحه‌ای برای خارج کردن انتخاب شود که به آن ارجاع نشده باشد، یعنی بیت R آن صفر باشد، اما اگر چنین صفحه‌ای پیدا نشد، ناچاریم از بین صفحاتی که بیت R آنها یک است، صفحه‌ای را انتخاب کنیم که از میان آنها، صفحه‌ای که تغییر نکرده باشد (بیت M آن صفر باشد) اولویت دارد زیرا نیازی به نوشتن دوباره آن در دیسک نداریم. در نهایت اگر چنین صفحه‌ای هم پیدا نشد، باید یک صفحه که بیت M آن یک است را انتخاب کنیم.

با دقت در الگوریتم NRU متوجه می‌شویم این الگوریتم صفحات را به ۴ گروه تقسیم می‌کند:

M	R	رده
۰	۰	گروه ۰
۰	۱	گروه ۱
۱	۰	گروه ۲
۱	۱	گروه ۳

با این تفسیر، گروه صفر صفحاتی هستند که اصلاً به آنها ارجاعی نشده است. گروه یک صفحاتی هستند که به آنها رجوع شده اما ارجاع‌ها فقط برای خواندن بوده و چیزی در آنها تغییر نکرده است. گروه ۲ در واقع صفحاتی هستند که کمی قبل به آنها ارجاع شده و این ارجاع آنها را تغییر هم داده است اما سیستم عامل با منقضی شده تایمر، بیت R آنها را صفر کرده است. گروه ۳ صفحاتی هستند که اخیراً به آنها ارجاع شده و تغییر هم یافته‌اند.

به این ترتیب مشاهده می‌شود بهترین گزینه برای ترک حافظه، گروه ۰، پس از آن گروه ۱، سپس گروه ۲ و در نهایت گروه ۳ می‌باشد.

نکته: الگوریتم **NRU** امکان ناهنجاری بیِ‌لیدی را **دارد**.

الگوریتم LFU **یا** NFU (Least Frequently Used **یا** Not Frequently Used)

این الگوریتم در برخی کتاب‌ها با عنوان LFU و در برخی دیگر با عنوان NFU ذکر شده است. این روش در واقع یکی از روش‌های پیاده‌سازی LRU می‌باشد. در این ایده، یک شمارنده به هر صفحه تعلق می‌گیرد که در شروع کار برابر صفر است. در هر وقفه‌ی ساعت، سیستم عامل بیت R هر صفحه را

(که می‌تواند صفر یا یک باشد) به شمارنده هر صفحه می‌افزاید. در واقع می‌توان گفت این شمارنده تعداد ارجاعات هر صفحه را شمارش و نگهداری می‌کند.
با اعمال این روش، هنگامی که یک وقفه‌ی نقص صفحه رخ می‌دهد، صفحه‌ای که شمارنده آن **کمترین مقدار** را دارد، جهت خروج از حافظه انتخاب می‌شود.
نکته: یک ایراد الگوریتم LFU این است که این الگوریتم هرگز چیزی را فراموش نمی‌کند. برای مثال، صفحه‌ای را در نظر بگیرید که در یک بازه زمانی بارها مورد ارجاع واقع شود و در نتیجه شمارنده آن به ناگاه افزایش چشمگیری پیدا کند، اما پس از مدتی ارجاعات به این صفحه قطع شوند. الگوریتم LFU موجب می‌شود این صفحه کماکان در حافظه باقی بماند زیرا مقدار شمارنده آن بسیار بالاست.

الگوریتم MFU (Most Frequently Used)
این الگوریتم همانند LFU از یک شمارنده به ازای هر صفحه استفاده می‌کند. نحوه افزایش این شمارنده نیز همانند LFU می‌باشد اما هنگام وقوع یک نقص صفحه، صفحه‌ای برای خروج انتخاب می‌شود که مقدار شمارنده آن از همه **بزرگ‌تر** باشد!!
ایده این الگوریتم این است که صفحه‌ای که شمارنده آن بزرگ است، به اندازه کافی در حافظه قرار داشته اما صفحه‌ای که شمارنده آن کوچک است احتمالاً تازه به حافظه وارد شده است و باید شانس استفاده شدن به وی داده شود.

تخصیص قاب به فرآیندها

جدا از اینکه الگوریتم جایگزینی صفحه در سیستم عامل چگونه باشد، چگونگی تخصیص قاب‌های فیزیکی به فرآیندها نیز تأثیر مهمی بر روی کارایی دارد.

تخصیص مساوی در برابر تخصیص متناسب

اولین مسئله‌ای که باید هنگام تخصیص قاب‌ها رعایت شود این است که در ابتدای کار قاب‌های موجود چگونه بین فرآیندها تقسیم شوند.
اولین الگوی تخصیص، الگوی **تخصیص مساوی** نام دارد. در این روش قاب‌های حافظه بین فرآیندهای موجود، به طور مساوی تقسیم می‌شوند. به عنوان مثال اگر در یک سیستم m قاب آزاد داشته باشیم و بخواهیم آنها را بین n فرآیند تقسیم کنیم، به هر یک از فرآیندها $\left(\frac{m}{n}\right)$ قاب آزاد اختصاص می‌یابد.
اما در الگوی **تخصیص متناسب** فرآیندها براساس اندازه نهایی خود، حافظه را در اختیار می‌گیرند. بنابراین فرآیندهای بزرگ‌تر تعداد قاب بیشتری اختیار خواهند کرد. فرض کنید تعداد صفحات فرآیند P_i برابر S_i و تعداد کل قاب‌های حافظه فیزیکی برابر m باشد. در این حالت به هر فرآیند، a_i قاب حافظه تعلق می‌گیرد که a_i برابر است با:

$$a_i = \frac{S_i}{\sum S_i} \times m$$

به عنوان مثال حافظه‌ای را در نظر بگیرید که ۷۰ قاب آزاد در اختیار داشته باشد. در این سیستم ۳ فرآیند P_1، P_2 و P_3 وجود دارند که اندازه P_1 برابر ۲۵ صفحه، اندازه P_2 برابر ۵۰ صفحه و اندازه P_3 برابر ۱۰۰ صفحه است. با استفاده از سیاست تخصیص متناسب، ۷۰ قاب آزاد حافظه به صورت زیر بین این ۳ فرآیند تقسیم می‌شوند:

$a_1 = \frac{۲۵}{۱۷۵} \times ۷۰ = ۱۰$ (تعداد قاب در اختیار P_1)

$a_2 = \frac{۵۰}{۱۷۵} \times ۷۰ = ۲۰$ (تعداد قاب در اختیار P_2)

$a_3 = \frac{۱۰۰}{۱۷۵} \times ۷۰ = ۴۰$ (تعداد قاب در اختیار P_3)

نکته: در عمل، سیستم عامل‌ها علاقه دارند تعداد قاب بیشتری در اختیار فرایندهای با اولویت بالاتر قرار دهند تا این فرآیندها با سرعت بیشتری اجرا شوند. به همین جهت اغلب سیستم عامل‌ها علاوه بر اندازه فرآیندها، به اولویت آنها نیز توجه می‌کنند. در واقع هنگام تخصیص قاب به فرآیندها، به فرآیندهای با اولویت بالاتر، حافظه بیشتری اختصاص می‌دهند.

تخصیص ثابت در برابر تخصیص متغیر

در سیاست **تخصیص ثابت**، تعداد ثابتی قاب به یک فرآیند داده می‌شود. در واقع در این روش هنگام بار شدن اولیه فرآیند، تعدادی قاب به آن اختصاص می‌یابد و تا انتها فرآیند تنها می‌تواند از همین تعداد قاب استفاده کند.

اما سیاست **تخصیص متغیر** اجازه می‌دهد تا تعداد قاب‌های تخصیص یافته به یک فرآیند در طول اجرای آن تغییر کند. به عنوان مثال اگر یک فرآیند به طور مداوم دچار وقفه نقص صفحه می‌شود، باید تعداد قاب بیشتری به وی داده شود تا نرخ خطای صفحه آن کاهش یابد.

تخصیص محلی در برابر تخصیص سراسری

تخصیص قاب‌ها به فرآیندها می‌تواند به دو صورت محلی و سراسری انجام شود:

در تخصیص محلی (Local)، هنگامی که برای یک فرآیند وقفه نقص صفحه رخ داد، فقط از بین قاب‌های مربوط به آن فرآیند می‌توان یک قاب را برای جایگزینی انتخاب کرد. اما در تخصیص سراسری (Global) هنگامی که یک فرآیند با نقص صفحه مواجه شد، از بین مجموعه کل قاب‌ها می‌توان یک قاب را برای جایگزینی انتخاب کرد، حتی اگر این قاب در اختیار فرآیندی دیگر باشد.

نکته: در واقع در تخصیص محلی، سهم هر فرآیند از حافظه هرگز تغییر نمی‌کند (تخصیص ثابت)، اما

در تخصیص سراسری، سهم فرآیندها از حافظه به صورت پویا تغییر می‌کند (تخصیص متغیر).

نکته: یکی از مشکلات تخصیص سراسری این است که مجموعه قاب‌های در اختیار یک فرآیند به رفتار صفحه‌بندی فرآیندهای دیگر نیز وابسته است. بنابراین یک فرایند یکسان ممکن است در اجراهای متفاوت، به گونه‌ای متفاوت عمل کند. مثلاً در یک اجرا ۱ ثانیه به طول انجامد و در اجرای دیگر ۵ ثانیه!

نکته: یکی از مشکلات تخصیص محلی این است که ممکن است یک فرآیند با نقص صفحه‌های مکرر مواجه شود و در واقع حافظه کم بیاورد، اما فرآیندهای دیگر از حافظه خود استفاده مؤثر و مفیدی نکنند، در واقع تعدادی قاب بدون استفاده باشند.

نکته: در عمل، سیستم عامل‌ها بیشتر از روش تخصیص سراسری استفاده می‌کنند.

اندازه قاب‌ها و صفحات

تعیین اندازه صفحات را می‌توان بر عهده سیستم عامل گذاشت. باید دقت کرد که انتخاب اندازه مناسب و بهینه برای صفحات، تأثیر مستقیمی بر روی کارایی دارد. برای بهبود کارایی گاهی اندازه صفحه باید کوچک باشد و گاهی بزرگ. برخی از عوامل مؤثر بر تعیین اندازه مناسب صفحه به قرار زیرند:

الف ـ معمولاً اندازه فرآیندها مضرب صحیحی از اندازه صفحه نیست و به طور متوسط نیمی از آخرین صفحه به ازای هر فرآیند خالی می‌ماند (تکه تکه شدن داخلی). بنابراین با کوچک‌تر شدن اندازه صفحات، میزان فضای تلف شده بابت تکه تکه شدن داخلی کاهش می‌یابد.

ب ـ بزرگ بودن اندازه صفحه یعنی بلااستفاده ماندن بخش بزرگی از حافظه زیرا وقتی که یک صفحه بزرگ به حافظه آورده شود، تمام قسمت‌های آن مورد استفاده قرار نمی‌گیرد. بنابراین با کوچک‌تر شدن اندازه صفحه، استفاده مؤثرتری از فضای حافظه می‌شود.

ج ـ هر چه اندازه صفحات بزرگ‌تر باشد، فرآیند به صفحات کمتری تقسیم می‌شود، در نتیجه اندازه جداول صفحه کاهش می‌یابد. در صورتی که با کوچک شدن اندازه صفحات، جداول صفحه بزرگ‌تر می‌شوند و این یعنی هزینه اضافی.

د ـ کوچک بودن صفحات و در نتیجه افزایش تعداد صفحات یک فرآیند به معنای افزایش تعداد دفعات جابجایی صفحات بین دیسک و حافظه است. با علم به اینکه مدت زمان لازم برای جابجایی یک صفحه کوچک بین دیسک و حافظه تقریباً برابر با مدت زمان جابجایی یک صفحه بزرگ می‌باشد، درمی‌یابیم کاهش اندازه صفحات و در نتیجه افزایش تعداد جابجایی‌ها منجر به تأخیر بیشتر می‌شود.

هـ ـ در برخی از سیستم‌ها هنگام تعویض متن، جداول صفحه باید در ثبات‌های سخت‌افزاری بار شوند. هر چه اندازه صفحات کوچک‌تر باشد، تعداد صفحات یک فرآیند بیشتر می‌شوند، در نتیجه اندازه

جدول صفحه بزرگ‌تر خواهند شد و بارکردن این جداول در ثبات‌های سخت‌افزاری بیشتر طول می‌کشد.

نکته: **سربار حافظه به ازای هر فرآیند** به دو قسمت تقسیم می‌شود: یکی میزان حافظه تلف شده به ازای آخرین صفحه فرآیند (تکه تکه شدن داخلی) و دیگری جدول صفحه (که به نوعی سربار محسوب می‌شود)، که باید این حافظه‌های سربار را به حداقل رساند. این مسئله را می‌توان به صورت ریاضی تحلیل کرد: اگر فرض کنیم اندازه متوسط فرآیندها s بایت، اندازه صفحه‌ها p بایت و اندازه هر درایه در جدول صفحه e باید باشد، هر فرآیند به طور متوسط به تعداد $\frac{s}{p}$ صفحه نیاز دارد در نتیجه اندازه جدول صفحه هر فرآیند برابر است با $e \times \frac{s}{p}$ بایت. از طرفی هر فرآیند در آخرین صفحه خود $\frac{p}{۲}$ بایت فضای تلف شده دارد (تکه تکه شدن داخلی). بنابراین هر فرآیند به طور متوسط سرباری معادل $\frac{es}{p} + \frac{p}{۲}$ خواهد داشت که باید این مقدار را به حداقل رساند. البته مشاهده می‌کنید در عبارت $e \times \frac{s}{p}$، اندازه صفحه (p) در مخرج کسر قرار دارد که با کاهش اندازه صفحه، این مقدار افزایش می‌یابد، اما در عبارت $\frac{p}{۲}$ اندازه صفحه در صورت کسر قرار دارد که با کاهش اندازه صفحه این مقدار کاهش می‌یابد. بنابراین اندازه بهینه p باید مقداری بین این دو مقدار باشد. برای به دست آوردن این نقطه، از عبارت $\frac{es}{p} + \frac{p}{۲}$ نسبت به p مشتق گرفته و مقدار آن را برابر صفر قرار می‌دهیم:

$$-\frac{se}{p^۲} + \frac{۱}{۲} = ۰ \quad \Rightarrow \quad p = \sqrt{۲se}$$

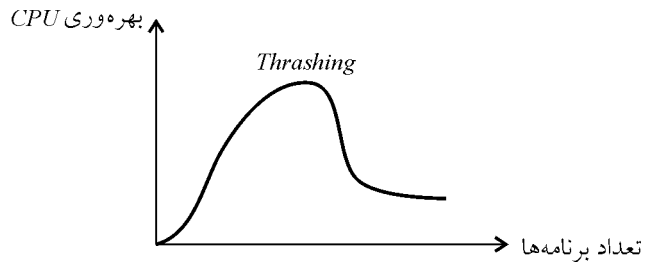
پس **اندازه بهینه صفحه** برای حداقل شدن سربار برابر است با $\sqrt{۲se}$.

۴ـ۱۳ـ کوبیدگی (له شدگی) (Thrashing)

اگر حافظه تخصیص داده شده به یک فرآیند، آن‌قدر کوچک باشد که نتواند صفحاتی که فرآیند، زیاد با آنها سر و کار دارد را در خود جای دهد، سرعت اجرای فرآیند کاهش می‌یابد، زیرا این فرآیند خطاهای نقص صفحه زیادی تولید می‌کند. در این حالت مدت زمان اجرای یک فرآیند به چندین و چند برابر حالت عادی افزایش می‌یابد. اصطلاحاً به برنامه‌ای که در هر ۲ یا ۳ دستور خود یک خطای نقص صفحه تولید کند، **کوبیده شده (لهیده)** گویند.

نکته: فرآیندی که در حالت Thrashing واقع است، به جای آن که زمان CPU را به اجرا اختصاص دهد، زمان زیادی را صرف انجام عملیات صفحه‌بندی می‌کند.

نکته: نسبت میزان بهره مفید CPU با افزایش تعداد فرآیندها به صورت زیر است:



پدیده کوبیدگی

در واقع تا یک نقطه، افزایش تعداد فرآیندها (افزایش سطح چندبرنامگی)، بهره‌وری CPU را افزایش می‌دهد، اما از یک نقطه به بعد، افزایش تعداد برنامه‌ها بهره‌وری پردازنده را کاهش می‌دهد. به عنوان مثال دو فرآیند A و B را در نظر بگیرید. فرض کنید برنامه A در حال اجرا به یک صفحه نیاز دارد، بنابراین بعد از یک خطای نقص صفحه، صفحه مورد تقاضایش به حافظه آورده می‌شود و به جای یک صفحه از فرآیند B در یک قاب حافظه قرار می‌گیرد، در این لحظه نوبت به اجرای برنامه B می‌رسد و فرآیند B به همان صفحه قدیمی خودش نیاز دارد، بنابراین با یک خطای نقص صفحه آن صفحه را به حافظه برمی‌گرداند و این بار صفحه فرآیند A باید حافظه را ترک کند و الی آخر ... در واقع در این حالت صفحاتی که فرآیندها زیاد با آنها سر و کار دارند به طور کامل در حافظه قرار ندارد و کارآیی سیستم به شدت کاهش می‌یابد.

نکته: اگر عملکرد سیستم عامل را در قبال درجه چندبرنامگی سیستم بررسی کنیم، متوجه می‌شویم تحت شرایطی خاص، پدیده Thrashing به شدت تشدید می‌شود.

سناریوی زیر را در نظر بگیرید:

می‌دانیم سیستم عامل بر بهره‌وری CPU نظارت دارد و اگر بهره‌وری CPU بسیار کم بشاد، درجه چندبرنامگی را با افزودن یک فرآیند جدید به سیستم افزایش می‌دهد تا بهره CPU افزایش یابد. فرض کنید در این سیستم از یک الگوریتم جایگزینی سراسری برای صفحات استفاده می‌شود که صفحات را بدون توجه به این که مربوط به کدام فرآیند هستند، جایگزین می‌کند. حال فرض کنید فرآیندی وارد یک مرحله جدید از اجرا شده و به چند صفحه جدید نیاز دارد. بنابراین وقفه‌های نقص صفحه برای این فرآیند آغاز می‌شوند و این فرآیند قاب‌های فرآیندهای دیگر را در اختیار می‌گیرد. از طرفی فرآیندهایی که تعدادی از صفحات آنها از حافظه خارج شده‌اند، به آن صفحات نیاز دارند، بنابراین وقفه‌های نقص صفحه برای این فرآیند آغاز می‌شوند و این فرآیند قاب‌های فرآیندهای دیگر را در اختیار می‌گیرد. از طرفی فرآیندهایی که تعدادی از صفحات آنها از حافظه خارج شده‌اند، به آن صفحات نیاز دارند، بنابراین وقفه‌های نقص صفحه برای آنها نیز شروع می‌شود و به طور مشابه این فرآیندها قاب‌های فرآیندهای دیگر را در اختیار می‌گیرند و این مسئله برای فرآیندهای دیگر تکرار

می‌شود. بنابراین فرآیندها پی در پی خطای نقص صفحه را تجربه می‌کنند و صفحات درگیر، مرتباً به داخل و خارج حافظه مبادله می‌شوند. به این ترتیب بهره‌وری CPU کاهش می‌یابد و زمانبند پردازنده متوجه این کاهش بهره‌وری می‌گردد و جهت افزایش بهره‌وری CPU درجه چندبرنامگی را افزایش می‌دهد.

فرآیندهای جدید نیز برای شروع سعی می‌کنند قاب‌هایی از سایر فرآیندها دریافت کنند که در نتیجه نقص صفحه‌های بیشتری رخ می‌دهد.

نکته: پدیده Thrashing را می‌توان با استفاده از الگوریتم‌های محلی جایگزینی صفحه به جای الگوریتم‌های سراسری، کنترل کرد. در این حالت یک فرآیند که دچار کوبیدگی شده است نمی‌تواند فرآیندهای دیگر را نیز مبتلا کند!

نکته: یک ایده دیگر جهت مقابله با Thrashing این است که فرآیندها اولویت بندی شوند. در این صورت برنامه‌های با اولویت پایین‌تر اجازه ندارند صفحات مربوط به فرآیندهای با اولویت بالاتر را از حافظه خارج کنند.

نکته: یک راه دیگر مقابله با Thrashing کنترل تعداد وقفه‌های نقص صفحه می‌باشد. به این ترتیب که اگر تعداد نقص‌های صفحه برای یک فرآیند افزایش یافت، باید تعدادی قاب حافظه به آن اختصاص یابد و اگر تعداد نقص‌های صفحه برای یک فرآیند از یک حد پایین کمتر شد، باید تعدادی قاب از آن فرآیند پس گرفته شود. نکته مهم اینجاست که اگر تعداد نقص‌های صفحه یک فرآیند بالا رفت، ولی قاب آزاد در حافظه وجود نداشت، باید درجه چندبرنامگی سیستم را کاهش داد و یک یا چند فرآیند را به حالت معلق درآورد تا قاب‌هایی که در اختیار دارد، آزاد شود. به این تکنیک **فرکانس خطای صفحه یا PFF** (Page Fault Frequency) گویند.

مدل مجموعه کاری (Working Set Model)

مجموعه کاری هر فرآیند، عبارت است از صفحاتی از آن فرآیند که اگر در حافظه قرار داشته باشند، فرآیند موردنظر کارآیی و سرعت قابل قبولی دارد. در واقع اگر مجموعه کاری یک فرآیند در حافظه باشد، فرآیند با تعداد معقول و مناسبی وقفه نقص صفحه به کار خود ادامه می‌دهد. مجموعه کاری یک فرآیند در طول اجرای آن ممکن است تغییر کند. به عنوان مثال در یک بازه زمانی خاص، یک فرآیند فقط به ۷ صفحه خود نیاز دارد، بنابراین مجموعه کاری این فرآیند در آن بازه، آن ۷ صفحه هستند، اما همین فرآیند در یک بازه دیگر فقط، به ۵ صفحه دیگر خود نیاز دارد.

ایده مدل مجموعه کاری در واقع یک رهیافت جهت مقابله با پدیده Thrashing است. در این مدل (راهکار) از مفهوم مجموعه کاری به خوبی استفاده می‌شود. به طور خلاصه مدل مجموعه کاری بیان می‌کند که قبل از دادن نوبت اجرا به یک فرآیند، باید مجموعه کاری آن فرآیند به درون حافظه بار شود.

می‌توان گفت با انتقال مجموعه کاری یک فرآیند قبل از اجرای آن به درون حافظه، نرخ خطاهای نقص صفحه کاهش می‌یابد.

نکته: به بار کردن صفحات یک فرآیند قبل از اجرای آن، **پیش صفحه‌بندی (Prepaing)** گویند.

نکته: ایده **مدل مجموعه کاری** از صادق بودن **اصل محلی بودن مراجعات** استفاده می‌کند. اصل محلی بودن مراجعات به بیان ساده چنین است: به طور معمول هنگام اجرای یک فرآیند معقول، ارجاعات به حافظه یک گستره خاص محدود هستند. البته این گستره در طول اجرای فرآیند، از مکانی به مکان دیگر منتقل می‌شود.

ایده حافظه مجازی اغلب با تکنیک صفحه‌بندی پیاده‌سازی می‌شود اما حافظه مجازی را می‌توان با قطعه‌بندی نیز ترکیب کرد. اما چون اندازه قطعات در قطعه‌بندی مساوی نیستند، این روش قدری پیچیده است.

نکته: رعایت اصول ساده برنامه‌نویسی می‌تواند در افزایش سرعت اجرای فرآیندها و کاهش تعداد نقص صفحه تأثیر مستقیم داشته باشد. به عنوان مثال قطعه برنامه زیر را در نظر بگیرید که سعی دارد همه عناصر یک آرایه دو بعدی ۱۰۰×۱۰۰ را با صفر پر کند:

```
for i:=1 to 100 do
    for j:=1 to 100 do
        a[j,i]=0;
```

فرض کنید هر یک از عناصر این آرایه، ۲ بایتی هستند و اندازه هر صفحه در حافظه نیز ۲۰۰ بایت باشد. در این صورت این آرایه ۱۰۰۰۰ عنصری، جمعاً ۱۰۰ صفحه را اشغال می‌کند و از آنجا که عناصر آرایه در زبان‌های برنامه‌نویسی C و پاسکال به صورت سطری در حافظه ذخیره می‌شوند، در واقع هر سطر این آرایه در یک صفحه قرار می‌گیرد. یعنی در صفحه اول، عناصر a[1,1] تا a[1,100]، در صفحه دوم عناصر a[2,1] تا a[2,100]، تا صفحه صدم که عناصر a[100,1] تا a[100,100] قرار می‌گیرند.

با دقت در قطعه برنامه نوشته شده، مشاهده می‌شود که پردازش به صورت ستونی انجام می‌شود زیرا از اندیس حلقه بیرونی به عنوان اندیس ستون در آرایه استفاده کرده است (اندیس i). به این ترتیب اگر فط یک صفحه برای داده‌ها در اختیار این برنامه باشد، برای انجام عملیات خود، دقیقاً ۱۰۰۰۰ خطای نقص صفحه رخ می‌دهد. زیرا هنگامی که یک صفحه به حافظه آورده شد، فقط یکی از عناصر آن پردازش می‌شود و برای عنصر بعدی یک خطای نقص صفحه رخ می‌دهد، زیرا این عنصر در صفحه بعدی قرار دارد.

به این ترتیب به ازای هر عنصر، یک خطای نقص صفحه رخ می‌دهد.

اما اگر برنامه به صورت زیر نوشته شود، شرایط تغییر می‌کند:

```
for i:=1 to 100 do
        for j:=1 to 100 do
            a[i,j]=0;
```

در این حالت هنگامی که یک صفحه به حافظه آورده می‌شود، هر ۱۰۰ عنصر آن به ترتیب پردازش می‌شوند و نیازی به نقص صفحه نیست. در واقع جمعاً ۱۰۰ نقص صفحه رخ می‌دهد.