# سیستم‌های عامل

**نیم‌سال اول ۰۳-۰۲**

دانشکده‌ی مهندسی کامپیوتر

شماره دانشجویی: ۹۹۱۰۵۵۶۱

شایان صالحی

تمرین چهارم

## پاسخ مسئله‌ی ۱.

در جواب به این سوال می‌توان گفت که پیج‌های حافظه به یک اندازه نیستند و به یک اندازه بودن آنها می‌تواند مشکلاتی همانند مشکلات زیر به وجود بیارن:

- بهره‌وری ناکارآمد حافظه (تکه‌تکه شدن داخلی): وقتی فرآیندها به کل اندازه استاندارد صفحه نیاز ندارن، بخش استفاده نشده صفحه هدر می‌ره. به این موضوع تکه‌تکه شدن داخلی می‌گن.
- مشکلات عملکرد برای داده‌های حجیم: برای فرآیندهایی که با مجموعه‌های داده بزرگ سروکار دارن، استفاده از صفحات با اندازه استاندارد می‌تونه منجر به افزایش بار مدیریت صفحات شه و دسترسی به حافظه رو کند کنه.
- انعطاف‌پذیری محدود: صفحات با اندازه ثابت ممکنه برای همه نوع برنامه‌ها، به خصوص اونهایی که نیازهای متغیر حافظه‌ای دارن، بهینه نباشن.

حالا راه‌هایی که می‌تونیم این مشکل رو حل کنیم به این صورته:

- اندازه‌های متفاوت صفحه (ترکیبی از اندازه‌های صفحه): بعضی از سیستم‌های مدرن از بیشتر از یک اندازه صفحه پشتیبانی می‌کنن. این روش به سیستم عامل اجازه می‌ده تا حافظه رو به شکلی تخصیص بده که بیشتر به اندازه مجموعه داده‌های یک فرآیند مناسب باشه. مثلاً، صفحات بزرگ می‌تونن برای برنامه‌هایی که نیاز به حافظه زیادی دارن استفاده شن، در حالی که صفحات کوچکتر برای فرآیندهای کم‌تقاضا به کار می‌رن.
- صفحات بزرگ و صفحات عظیم (HugePages): تو لینوکس و بعضی از سیستم‌های عامل شبه یونیکس، یه ویژگی به نام HugePages وجود داره که اجازه استفاده از اندازه‌های بزرگتر صفحه (مثل ۲ مگابایت یا ۱ گیگابایت) رو برای برنامه‌های خاص می‌ده، که باعث کاهش بار مدیریت تعداد زیادی صفحات می‌شه.
- صفحه‌بندی تقاضامحور و جابجایی (Swapping): اگرچه راه‌حل مستقیمی برای مشکل اندازه صفحه نیست، صفحه‌بندی تقاضامحور (بارگذاری صفحات به حافظه فقط زمانی که نیاز هستند) و جابجایی (انتقال صفحات بین رم و دیسک) می‌تونه تاثیر اندازه‌های ناکارآمد صفحه رو با بهینه‌سازی نحوه استفاده از حافظه کاهش بده.

هر کدوم از این راه‌حل‌ها با معایب خودشون همراه هستن. مثلاً، در حالی که اندازه‌های بزرگتر صفحه می‌تونه بار مدیریت تعداد زیادی صفحه رو کم کنه، می‌تونه منجر به تکه‌تکه شدن داخلی بیشتر شه اگر درست مدیریت نشه. به همین دلیل، سیستم‌های عامل اغلب مکانیزم‌هایی رو فراهم می‌کنن تا این عوامل رو بر اساس نیازهای سیستم و برنامه‌هایش تعادل ببخشن.

## پاسخ مسئله‌ی ۲.

### الف

از آنجایی که هر PTE ۸ بیت برای آدرس فیزیکی داشته می‌دانیم که با این بیت‌ها ۲۵۶ = $۲^۸$ آدرس مختلف را می‌توان پوشش داد. همچنین اندازه هر صفحه براساس بیت‌های آفست تعیین شده که براساس معماری داده شده ۴ است. بنابراین هر صفحه می‌توان ۱۶ = $۲^۴$ بایت را آدرس‌دهی کند.

پس در نهایت بیشترین سایز آدرس فیزیکی برابر خواهد بود با:

$$۲۵۶ \times ۱۶ = ۴۰۹۶$$

### ب

انداره فضای آدرس‌دهی مجازی به تعداد بیت‌های virtual address مربوط می‌شود که در این حالت خاص ۸ بیت بوده پس ماکزیموم فضای آن می‌تواند ۲۵۶ = $۲^۸$ باشد.

## ج

از آنجایی که تعداد بیت‌های آفست ۴ بیت بوده اندازه هر صفحه برابر با ۱۶ = $۲^۴$ بایت خواهد بود.

### د

ابتدا آدرس داده شده را از مبنای ۱۶ به مبنای دو می‌بریم:

$$(۰x۶۳) \implies (۰۱۱۰۰۰۱۱)_۲$$

از آنجایی که با ارزش‌ترین رقم صفر بوده پس این آدرس از طریق MMU آدرس‌دهی نمی‌شود. بنابراین ۷ بیت باقی‌مانده همان آدرس فیزیکی را نشان می‌هند که برابر ۶۳ در مبنای ۱۶ است.

### ه

translations های ولید آنهایی هستند که در page table بیت ولید ۱ داشته باشند. آدرس‌های زیر شامل این حالت می‌شوند:

$$۰xFF, \quad ۰x۰۸, \quad ۰xB۰, \quad ۰xEE, \quad ۰xDD$$

بنابراین رنج آدرس‌های ولید متناسب با آنها به این صورت خواهد بود:

```
0xFF00   to   0xFF0F
0x0800   to   0x080F
0xB000   to   0xB00F
0xEE00   to   0xEE0F
0xDD00   to   0xDD0F
```

## پاسخ مسئله‌ی ۳.

در اینجا می‌دانیم که به صورت کلی در سیستم‌های ۳۲ بیتی PTE size ۴ بایت یا همان ۳۲ بیت است. حال برای پیدا کردن تعداد PTEها خواهیم داشت:

$$Number\,of\,PTEs = \frac{Page\,Size}{PTE\,Size} = \frac{۴۰۹۶}{۴} = ۱۰۲۴$$

در مورد استفاده از حافظه واقعی برای کمتر کردن سایز Page Table، سیستم عامل بخشی از حافظه فیزیکی (واقعی) رو برای ذخیره جدول‌های صفحه استفاده می‌کنه. اما، لازم نیست همه جدول‌های صفحه کاملاً پر شده باشند یا حتی همیشه در حافظه باشند.

همچنین روش Demand Paging سیستم عامل اجازه می‌ده فقط اون صفحاتی (و در نتیجه جدول‌های صفحه) رو به حافظه بارگذاری کنه که در حال حاضر نیازه، و اینکار اثر حافظه‌ای جدول‌های صفحه رو کاهش می‌ده.

تو سیستم جدول صفحه چند سطحی، فقط جدول صفحه سطح بالاتر (سطح_۲ در این مورد) باید کاملاً در حافظه حضور داشته باشه. جدول‌های صفحه سطح پایین‌تر (سطح_۱) به موقع بارگذاری می‌شن. این کار به شدت میزان حافظه واقعی لازم برای جدول‌های صفحه رو کاهش می‌ده.

در مورد آدرس ۳۲ بیتی هم ما سه بخش زیر را داریم:

- آفست ۱۲ بیتی: مکان دقیق درون یک صفحه رو مشخص می‌کنه (چون$۴۰۹۶ = ۲^{۱۲}$، که اندازه صفحه است). آفست ۱۲ بیتی برای پیدا کردن بایت دقیق در صفحه ۴ کیلوبایتی که فرآیند می‌خواد بهش دسترسی داشته باشه، استفاده می‌شه.
- جدول صفحه سطح_۱ ۱۰ بیتی: این قسمت آدرس برای ایندکس کردن در جدول صفحه سطح_۱ استفاده می‌شه. ۱۰ بیت برای جدول صفحه سطح_۱ استفاده می‌شه تا در جدول صفحه سطح_۱ مشخص شده (که توسط PTE سطح_۲ اشاره شده) ایندکس بشه، که دوباره می‌تونه ۱۰۲۴ ورودی داشته باشه.
- جدول صفحه سطح_۲ ۱۰ بیتی: این قسمت آدرس برای ایندکس کردن در جدول صفحه سطح_۲ استفاده می‌شه. ۱۰ بیت برای جدول صفحه سطح_۲، یکی از ۱۰۲۴ ($۲^{۱۰}$) ورودی‌های موجود در جدول صفحه سطح_۲ رو ایندکس می‌کنه. این ورودی به یک جدول صفحه سطح_۱ در حافظه اشاره می‌کنه.

این ساختار اجازه می‌ده تا مدیریت یک فضای آدرس مجازی بزرگ (۴ گیگابایت در یک سیستم ۳۲ بیتی) انجام بشه، در حالی که اثر حافظه‌ای جدول‌های صفحه رو با استفاده از تکنیک‌های صفحه‌بندی سلسله مراتبی و صفحه‌بندی تقاضامحور، قابل مدیریت نگه می‌داره.

## پاسخ مسئله‌ی ۴.

### الف

در اینجا از آنجایی که میانگین زمان دسترسی به حافظه $۵۰ ns$ بوده و دسترسی TLB هم $۱۰ ns$ است. حال دو حالت داریم:

- در حالت TLB hit زمان دسترسی به این صورت است:

$$time\,to\,access\,memory = ۱۰\ ns(TLB) + ۵۰\ ns(Memory) = ۶۰\ ns$$

- در حالت TLB miss داریم:

$$time\,to\,access\,memory = ۱۰\ ns(TLB) + ۵۰\ ns(PageTable) + ۵۰\ ns(Memory) = ۱۱۰\ ns$$

حال با توجه به ضریب TLB ۵۰ درصد خواهیم داشت:

$$Avarage\,Time = \frac{۱}{۲} \times ۶۰\ ns + \frac{۱}{۲} \times ۱۱۰\ ns = ۸۵\,ns$$

حال درصورتی که TLB نداشته باشیم داریم:

$$Avarage\,Time = ۵۰\ ns(PageTable) + ۵۰\ ns(Memory) = ۱۰۰\ ns$$

با قیاس کردن این دو حالت می‌توان فهمید که روش TLB با ضریب ۵۰ درصد می‌تواند میانگین زمان دسترسی را ۱۵ نانو ثانیه کاهش دهد.

### ب

اگر ضریب hit rate را $H$ در نظر گرفته و زمان میانگین را $T$ در نظر بگیریم داریم:

$$T = H \times (Time\,for\,TLB\,hit) + (۱ - H) \times (Time\,for\,TLB\,miss)$$

پس داریم:

$$T = ۶۱\ ns, \quad Time\,for\,TLB\,hit = ۶۰\ ns, \quad Time\,for\,TLB\,miss = ۱۱۰\ ns$$

$$\implies ۶۱ = H \times ۶۰ + (۱ - H) \times ۱۱۰ \implies ۵۰H = ۴۹ \implies H = ۰/۹۸$$

بنابراین ضریب TLB می‌بایست ۹۸ درصد باشد تا میانگین زمان ۶۱ نانو ثانیه شود..

## پاسخ مسئله‌ی ۵.

### الف

برای آنکه کمترین تعداد جدول صفحات را پیدا کنیم می‌دانیم می‌دانیم که Page Size ما ۶۴ کیلوبایت بوده و سایز Entry ما ۴ بایت بوده، به این ترتیب داریم:

$$\text{Entries of Number} = \frac{\text{Size Page}}{\text{Size Entry Table Page}} = \frac{۶۵۵۳۶\text{bytes}}{۴\text{bytes/entry}} = ۱۶,۳۸۴$$

هر جدول صفحه می‌تونه ۱۶۳۸۴ ورودی داشته باشه (همانطور که محاسبه شد). چون هر جدول صفحه در یک صفحه جا می‌گیره، حداقل اندازه لازم برای هر جدول صفحه، اندازه یک صفحه است، که ۶۴ کیلوبایت است.

### ب

در ابتدا می‌بایست بیت‌های آفست را تعیین کنیم، به این ترتیب داریم:

$$\text{Bits Offset} = \log_۲(۶۵۵۳۶) = ۱۶\text{bits}$$

آز آنجایی که آدرس مجازی ما ۲۴ بیت بوده خواهیم داشت:

$$\text{Bits Table Page} = \text{Bits Address Virtual Total} - \text{Bits Offset} = ۲۴ - ۱۶ = ۸\text{bits}$$

چون ۸ بیت برای جدول صفحه استفاده می‌شود، و هر ورودی مربوط به یک صفحه است، یک جدول صفحه سطح یک برای آدرس‌دهی کل فضای آدرس مجازی کافی است.

### ج

چون فضای آدرس فیزیکی ۳۲ـ بیتی است، ورودی‌های جدول صفحه باید شامل شماره صفحه فیزیکی به علاوه هر متادیتا دیگری باشند. با فرض اینکه کل آدرس فیزیکی ۳۲ـ بیتی برای آدرس‌دهی استفاده می‌شه (که شامل شماره صفحه فیزیکی و آفست درون صفحه می‌شه)، تعداد بیت‌های موجود برای متادیتا بستگی به این داره که چند بیت برای نمایش شماره صفحه فیزیکی لازمه.

شماره صفحه فیزیکی توسط فضای آدرس فیزیکی منهای بیت‌های آفست (که در این مورد برای صفحات فیزیکی و مجازی یکسان است) تعیین می‌شود:

$$\text{Bits Number Page Physical} = \text{Space Address Physical} - \text{Bits Offset} = ۳۲ - ۱۶ = ۱۶\text{bits}$$

بنابراین، تعداد بیت‌های موجود برای متادیتا در هر ورودی:

$$\text{Bits Metadata} = \text{Size Entry Table Page} - \text{Bits Number Page Physical} = ۳۲ - ۱۶ = ۱۶\text{bits}$$

پس، هر ورودی در جدول صفحه می‌تونه ۱۶ بیت متادیتا داشته باشد.

## پاسخ مسئله‌ی ۶.

### الف

در مدل FIFO قدیمی ترین صفحه را در مموری که ابتدا وارد شده بود را جایگزین می‌کند. در واقع همانند صف می‌ماند که صفحات جدید به انتها اضافه شده و صفحات قبلی از صف خارج می‌شوند. برای ترتیب درخواست‌های داده شده با استفاده از این روش خواهیم داشت:

۱. $Start$ : []
۲. $Request$ ۱ : [۱]
۳. $Request$ ۳ : [۱, ۳]
۴. $Request$ ۵ : [۱, ۳, ۵]
۵. $Request$ ۷ : [۱, ۳, ۵, ۷]
۶. $Request$ ۸ : [۳, ۵, ۷, ۸]
۷. $Request$ ۱ : [۵, ۷, ۸, ۱]
۸. $Request$ ۴ : [۷, ۸, ۱, ۴]
۹. $Request$ ۴ : [۷, ۸, ۱, ۴] $NoPageFault$
۱۰. $Request$ ۳ : [۸, ۱, ۴, ۳]
۱۱. $Request$ ۲ : [۱, ۴, ۳, ۲]
۱۲. $Request$ ۴ : [۱, ۴, ۳, ۲] $NoPageFault$
۱۳. $Request$ ۵ : [۴, ۳, ۲, ۵]
۱۴. $Request$ ۱ : [۳, ۲, ۵, ۱]
۱۵. $Request$ ۷ : [۲, ۵, ۱, ۷]
۱۶. $Request$ ۱ : [۲, ۵, ۱, ۷] $NoPageFault$

### ب

الگوریتم LRU با شانس دوم یک نوع از الگوریتم کمتر استفاده شده یا همان LRU هست که بعضی ویژگی‌های روش FIFO را هم شامل می‌شود. تو این الگوریتم، هر صفحه‌ای که تو حافظه هست یه بیت مرجع داشته، که اولش به صورت پیش‌فرض روی ۰ تنظیم شده. وقتی به یه صفحه دسترسی پیدا می‌شه، بیت مرجعش به ۱ تغییر می‌کنه. حالا، وقتی بخوایم یه صفحه رو با یه صفحه دیگه جایگزین کنیم، این الگوریتم صفحه‌ها رو به ترتیب FIFO (اولین صفحه‌ای که وارد شده، اولین صفحه‌ای که خارج می‌شه) بررسی می‌کنه و دنبال یه صفحه با بیت مرجع ۰ می‌گردد.

اگه به صفحه‌ای با بیت مرجع ۱ برخورد کرد، اون بیت رو دوباره به ۰ تغییر می‌دهد و به اون صفحه یه «شانس دوم» می‌ده، یعنی اونو می‌بره انتهای صف و جستجو رو ادامه می‌ده. این فرایند ادامه پیدا می‌کنه تا زمانی که به یه صفحه با بیت مرجع ۰ برسه و اونو جایگزین کنه.

نکته مهم اینه که بیت‌های مرجع هر بار که به صفحه‌ای دسترسی پیدا می‌شه، چه در مواقع برخورد (hits) و چه در مواقع عدم برخورد (misses)، به‌روزرسانی می‌شن. اگه یه صفحه چندین بار قبل از اینکه برای جایگزینی در نظر گرفته بشه، دسترسی پیدا کنه، بیت مرجعش همچنان روی ۱ باقی می‌مونه و این باعث می‌شه شانس بیشتری داشته باشه که تو حافظه بماند.

به این ترتیب برای درخواست‌های داده شده خواهیم داشت:

۱. $Start$ : []
۲. $Request$ ۱ : [۱]
۳. $Request$ ۳ : [۱, ۳]
۴. $Request$ ۵ : [۱, ۳, ۵]
۵. $Request$ ۷ : [۱, ۳, ۵, ۷]
۶. $Request$ ۸ : [۳, ۵, ۷, ۸]
۷. $Request$ ۱ : [۵, ۷, ۸, ۱]
۸. $Request$ ۴ : [۷, ۸, ۱, ۴]
۹. $Request$ ۴ : [۷, ۸, ۱, ۴] $NoPageFault$
۱۰. $Request$ ۳ : [۸, ۱, ۴, ۳]
۱۱. $Request$ ۲ : [۱, ۴, ۳, ۲]
۱۲. $Request$ ۴ : [۱, ۴, ۳, ۲] $NoPageFault$
۱۳. $Request$ ۵ : [۴, ۳, ۲, ۵]
۱۴. $Request$ ۱ : [۳, ۲, ۵, ۱]
۱۵. $Request$ ۷ : [۲, ۵, ۱, ۷]
۱۶. $Request$ ۱ : [۲, ۵, ۱, ۷] $NoPageFault$

## ج

الگوریتم LFU یا کمترین استفاده شده، یک روش مدیریت حافظه است که به صفحاتی که کمتر استفاده شده‌اند اولویت می‌دهد. هر صفحه یک شمارنده دارد که نشون می‌دهد چند بار بهش دسترسی پیدا شده است. وقتی نیاز به جایگزین کردن یک صفحه باشد، صفحه‌ای که کمترین تعداد دسترسی رو داشته انتخاب می‌شود.

اگه چند صفحه باشند که همه‌شون کمترین تعداد دسترسی رو داشته باشند (یعنی مساوی باشند)، معمولاً قدیمی‌ترین صفحه بین اون‌ها جایگزین می‌شود.

پیاده‌سازی الگوریتم LFU می‌تواند نسبت به سایر الگوریتم‌ها پیچیده‌تر باشد، چون نیاز داره شمارش دقیقی از تعداد دفعات دسترسی به صفحه‌ها نگه داشته بشه و ممکنه نیاز به مرتب‌سازی یا دسترسی سریع به صفحه‌ای که کمترین استفاده رو داشته باشد، پیش بیاد.

این الگوریتم فرض می‌کنه که در مواقعی که تعداد دفعات دسترسی برابر باشد، اصل FIFO برای تعیین اولویت به کار می‌رود، یعنی قدیمی‌ترین صفحه بین اون‌ها که تعداد دفعات دسترسی‌شون یکسانه، جایگزین می‌شود.

به این ترتیب برای خروجی این روش خواهیم داشت:

۱. $Start$ : []
۲. $Request$ ۱ : [۱]
۳. $Request$ ۳ : [۱, ۳]
۴. $Request$ ۵ : [۱, ۳, ۵]
۵. $Request$ ۷ : [۱, ۳, ۵, ۷]
۶. $Request$ ۸ : [۳, ۵, ۷, ۸]
۷. $Request$ ۱ : [۵, ۷, ۸, ۱]
۸. $Request$ ۴ : [۷, ۸, ۱, ۴]
۹. $Request$ ۴ : [۷, ۸, ۱, ۴] $NoPageFault$
۱۰. $Request$ ۳ : [۸, ۱, ۴, ۳]
۱۱. $Request$ ۲ : [۱, ۴, ۳, ۲]
۱۲. $Request$ ۴ : [۱, ۴, ۳, ۲] $NoPageFault$
۱۳. $Request$ ۵ : [۴, ۳, ۲, ۵]
۱۴. $Request$ ۱ : [۴, ۳, ۵, ۱]
۱۵. $Request$ ۷ : [۴, ۵, ۱, ۷]
۱۶. $Request$ ۱ : [۴, ۵, ۱, ۷] $NoPageFault$

**د**

الگوریتم جایگزینی صفحه بهینه یک رویکرد نظریه‌ایه که بیشتر برای مقایسه و ارزیابی سایر الگوریتم‌های جایگزینی صفحه استفاده می‌شود. این الگوریتم نیاز به دانستن ترتیب درخواست‌های صفحه در آینده دارد.

الگوریتم صفحه‌ای رو جایگزین می‌کنه که در آینده برای مدت طولانی‌تری استفاده نخواهد شد. به عبارت دیگه، از بین صفحات فعلی در حافظه، صفحه‌ای رو برای جایگزینی انتخاب می‌کنه که دسترسی بهش دورتر در آینده انجام خواهد شد.

یادداشت: الگوریتم بهینه در سیستم‌های واقعی قابل پیاده‌سازی نیست چون نیاز به دانستن درخواست‌های آینده دارد. با این حال، به عنوان یک معیار مفید برای مقایسه اثربخشی الگوریتم‌های جایگزینی صفحه عملی به کار می‌رود.

برای خروجی این روش نیز خواهیم داشت:

۱. $Start$ : []
۲. $Request$ ۱ : [۱]
۳. $Request$ ۳ : [۱, ۳]
۴. $Request$ ۵ : [۱, ۳, ۵]
۵. $Request$ ۷ : [۱, ۳, ۵, ۷]
۶. $Request$ ۸ : [۳, ۵, ۷, ۸]
۷. $Request$ ۱ : [۵, ۷, ۸, ۱]
۸. $Request$ ۴ : [۵, ۸, ۱, ۴]
۹. $Request$ ۴ : [۵, ۸, ۱, ۴] $NoPageFault$
۱۰. $Request$ ۳ : [۵, ۱, ۴, ۳]
۱۱. $Request$ ۲ : [۱, ۴, ۳, ۲]
۱۲. $Request$ ۴ : [۱, ۴, ۳, ۲] $NoPageFault$
۱۳. $Request$ ۵ : [۱, ۴, ۲, ۵]
۱۴. $Request$ ۱ : [۴, ۲, ۵, ۱] $NoPageFault$
۱۵. $Request$ ۷ : [۴, ۲, ۱, ۷]
۱۶. $Request$ ۱ : [۴, ۲, ۱, ۷] $NoPageFault$

## پاسخ مسئله‌ی ۷.

### الف

تخصیص‌دهنده اسلب یک مکانیزم مدیریت حافظه است که در تخصیص حافظه سطح هسته، به ویژه در سیستم‌های عامل مانند UNIX و لینوکس استفاده می‌شود. این تخصیص‌دهنده، حافظه را در بلوک‌هایی به نام "اسلب‌ها" سازماندهی می‌کنه که به قطعات کوچک‌تر با اندازه ثابت تقسیم می‌شوند. هر اسلب برای یک نوع خاص از شیء یا ساختار داده (مثلاً، شیء inode ، ساختارهای وظیفه، شیء فایل و غیره) اختصاص داده شده است. ین سازماندهی به تخصیص‌دهنده امکان می‌ده تا تخصیص‌ها و آزادسازی‌های حافظه را به طور مؤثر مدیریت کنه، به ویژه برای اشیاء که به طور مکرر استفاده می‌شن.

در رابطه با مزایای استفاده از این روش به این دو مورد می‌توان اشاره کرد:

- کاهش تکه‌تکه شدن: با تخصیص حافظه در قطعات با اندازه ثابت که مخصوص انواع خاصی از اشیاء هستند، تخصیص‌دهنده اسلب به طور قابل توجهی تکه‌تکه شدن حافظه رو کاهش می‌ده. این به ویژه برای سیستم‌هایی که به مدت طولانی اجرا می‌شن و جایی که تخصیص و آزادسازی حافظه پویا می‌تواند به مرور زمان منجر به تکه‌تکه شدن شود، مفید است.
- بهبود عملکرد برای اشیاء پرکاربرد: از آنجایی که اسلب‌ها به اشیاء خاصی اختصاص داده شده‌اند و پس از استفاده یک باره اولیه همچنان آماده استفاده هستند، فرایندهای تخصیص و آزادسازی سریع‌تر انجام می‌شوند. این امر به دلیل حذف نیاز به مقدمات اولیه و تخریب مداوم، منجر به استفاده مؤثرتر از چرخه‌های CPU و زمان پاسخ سریع‌تر برای درخواست‌های تخصیص حافظه می‌شود.

### ب

در سیستم‌های عامل Real Time (RTOS)، قابل پیش‌بینی بودن و به موقع بودن پاسخ‌ها بسیار مهم است. سیاست تخصیص محلی، جایی که هر پردازنده یا هسته دارای مخزن حافظه محلی خود است، به دلایل زیر می‌تواند برای RTOS مناسب باشد:

قابل پیش‌بینی بودن: تخصیص محلی وابستگی‌ها بین پردازنده‌ها یا هسته‌ها را به حداقل می‌رساند. در یک سیستم Real Time، قابل پیش‌بینی بودن کلیدی است و با داشتن مخازن حافظه محلی، سیستم می‌تواند از بی‌قاعدگی ناشی از منابع حافظه مشترک اجتناب کند. این به ویژه در سیستم‌های Real Time سخت که رعایت مهلت‌ها حیاتی است، مهم است و هر گونه تأخیر ناشی از رقابت برای منابع مشترک می‌تواند منجر به شکست سیستم شود.

کاهش تأخیر: دسترسی به حافظه محلی معمولاً سریع‌تر از دسترسی به حافظه مشترک یا دور است. در سیستم‌های Real Time، کاهش تأخیر برای برآورده کردن الزامات زمان‌بندی سختگیرانه ضروری است. تخصیص محلی اجازه می‌دهد هر پردازنده یا هسته حافظه خود را مدیریت کند، منجر به دسترسی سریع‌تر به حافظه و کاهش تأخیر در پردازش وظایف Real Time می‌شود.

اجتناب از بارهای اضافی همگام‌سازی: به اشتراک گذاشتن مخازن حافظه بین پردازنده‌ها یا هسته‌ها نیازمند مکانیزم‌های همگام‌سازی (مانند قفل‌ها یا سمافورها) برای جلوگیری از مشکلات دسترسی هم‌زمان است. این مکانیزم‌های همگام‌سازی می‌توانند بار اضافی و بی‌قاعدگی در زمان پاسخ‌دهی را به وجود آورند. با تخصیص محلی، نیاز به چنین همگام‌سازی‌هایی به شدت کاهش می‌یابد یا حذف می‌شود، بنابراین توانایی سیستم برای پاسخگویی سریع و قابل پیش‌بینی بهبود می‌یابد.