

Operating System Concepts

TENTH EDITION

ABRAHAM SILBERSCHATZ • PETER BAER GALVIN • GREG GAGNE



WILEY

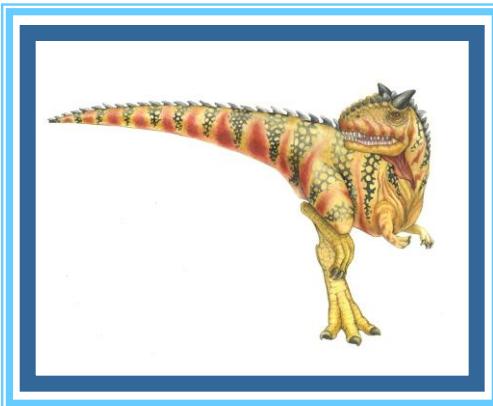
سیستم عامل

Dr. A. Taghinezhad



Website: ataghinezhad.github.io, Email: a0taghinezhad@gmail.com

فصل ۹: حافظه اصلی





فصل ۹: حافظه اصلی

- پیش زمینه
- تخصیص حافظه پیوسته
- صفحه‌بندی
- ساختار جدول صفحه
- جایگزینی (Swapping)
- مثال : معماری‌های ۳۲ و ۶۴ بیتی
- اینتل مثال : معماری ARMv8



اهداف Objectives

- این بخش به توصیف جزئیات روش‌های مختلف سازماندهی سخت‌افزار حافظه می‌پردازد
- همچنین به بحث در مورد تکنیک‌های مدیریت حافظه، توضیحات دقیق در مورد پردازنده اینتل پنتیوم که از هر دو حالت تقسیم‌بندی خالص و تقسیم‌بندی با صفحه‌بندی پشتیبانی می‌کند، می‌پردازد.



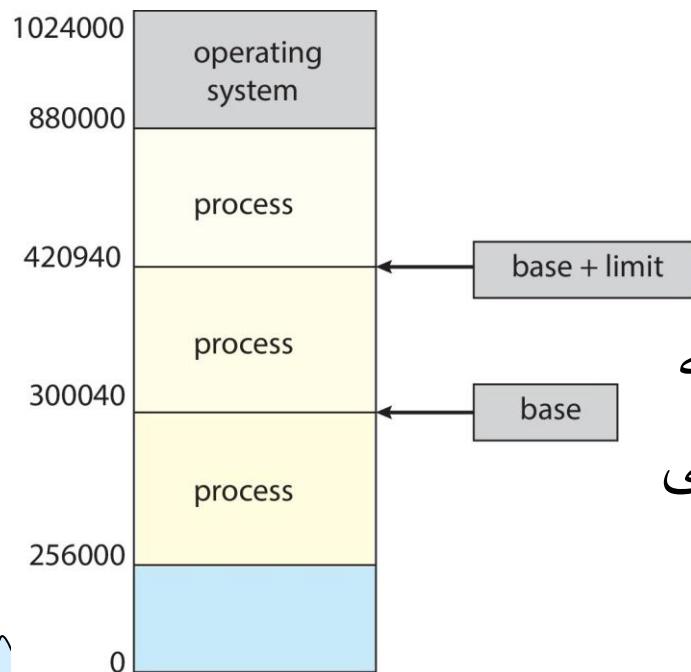
پیش زمینه

- برای اجرا، یک برنامه باید از دیسک به حافظه آورده شده و در فرآیندی قرار گیرد . حافظه اصلی و رجیسترها تنها حافظه‌هایی هستند که پردازنده می‌تواند به طور مستقیم به آن‌ها دسترسی داشته باشد .
- واحد حافظه تنها جریان‌های زیر را می‌بینید:
 - آدرسها + درخواست خواندن، یا
 - آدرس + داده و درخواست نوشتن
- دسترسی به رجیستر در یک سیکل ساعت پردازنده (یا کمتر) انجام می‌شود
- حافظه اصلی می‌تواند چندین سیکل طول بکشد و باعث توقف Stall پردازنده شود
- کش بین حافظه اصلی و رجیسترهای پردازنده برای کمتر کردن توقف قرار می‌گیرد.
- برای اطمینان از عملکرد صحیح، حفاظت از حافظه ضروری است.



محافظت

- باید مطمئن شویم که یک فرآیند فقط بتواند به آدرس‌های موجود در فضای آدرس خود دسترسی پیدا کند
- ما می‌توانیم این حفاظت را با استفاده از یک جفت رجیستر پایه **base** و محدوده **limit** ارائه دهیم که فضای آدرس منطقی یک فرآیند را تعریف می‌کند.

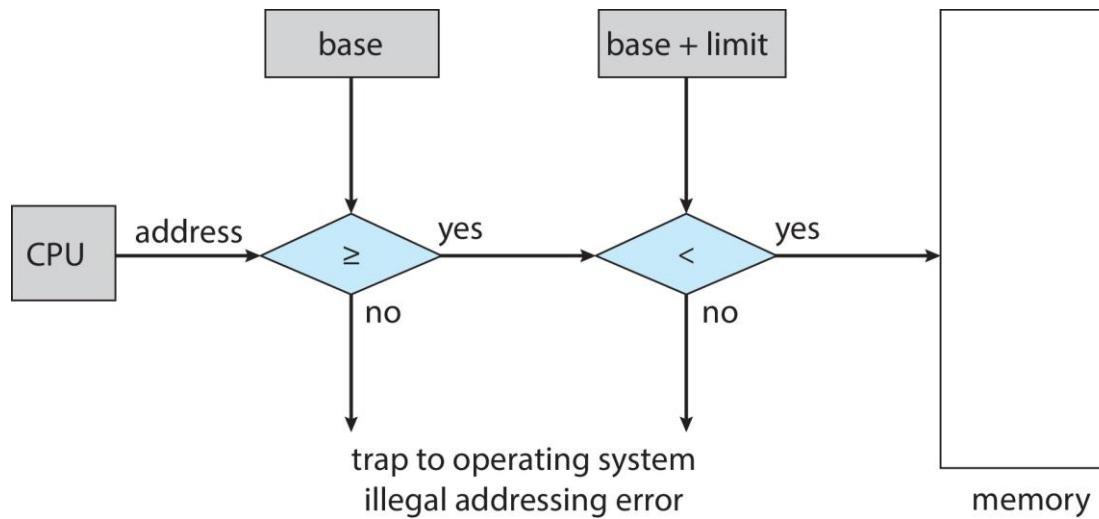


▪ به عنوان مثال، اگر ثبات پایه ۳۰۰۰۴۰ داشته باشد و رجیستر محدود ۱۲۰۹۰۰ باشد، برنامه می‌تواند به طور قانونی به همه آدرس‌ها از ۳۰۰۰۴۰ تا ۴۲۰۹۳۹ دسترسی داشته باشد.

محافظت از آدرس سخت افزاری



- پردازنده باید هر دسترسی به حافظه که در حالت کاربر ایجاد می شود را بررسی کند تا مطمئن شود بین پایه و محدوده برای آن کاربر قرار دارد.



- دستورالعمل های مربوط به بارگذاری رجیستر های پایه و محدوده دارای امتیاز هستند.



اتصال آدرس Address Bind

- برنامه‌های روی دیسک، آماده برای آمدن به حافظه برای اجرا، یک صفت ورودی را تشکیل می‌دهند .
- بدون پشتیبانی، باید در آدرس ۰۰۰۰ بارگذاری شوند
- اینکه آدرس فیزیکی اولین فرآیند کاربر همیشه در ۰۰۰۰ باشد، ناخوشایند است .
- آدرس‌ها در مراحل مختلف عمر برنامه به روش‌های مختلف نشان داده می‌شوند
- آدرس‌های کد منبع معمولاً نمادین هستند .
- آدرس‌های کد کامپایل شده به آدرس‌های قابل جابه‌جایی متصل می‌شوند
- به عنوان مثال، "۱۴ بایت از ابتدای این مژول
- لینکر (linker) یا لودر (loader) آدرس‌های قابل جابه‌جایی **relocatable addresses** را به آدرس‌های مطلق متصل می‌کنند
- به عنوان مثال، ۷۴۰۱۴
- هر اتصال binding یک فضای آدرس را به فضای دیگر نگاشت می‌کند.



اتصال دستورالعمل ها در حافظه

- اتصال یا bind آدرس دستورالعمل ها و داده ها به آدرس های حافظه می تواند در سه مرحله‌ی مختلف اتفاق بیفتد:
- زمان کامپایل (تجمیع) : اگر موقعیت حافظه از قبل مشخص باشد، کد مطلق (Absolute Code) قابل تولید است؛ اما اگر موقعیت شروع تغییر کند، کد نیاز به کامپایل مجدد دارد.
- زمان لود یا بارگذاری : اگر موقعیت حافظه در زمان کامپایل مشخص نباشد، باید کد قابل جابه‌جایی (Relocatable Code) تولید شود.
- زمان اجرا : اگر فرآیند در طول اجرا بتواند از یک بخش حافظه به بخش دیگر منتقل شود، اتصال آدرس تا زمان اجرا به تعویق می‌افتد . برای نگاشتهای آدرس (مانند رجیسترهای پایه و محدوده) به پشتیبانی سخت‌افزاری نیاز است.



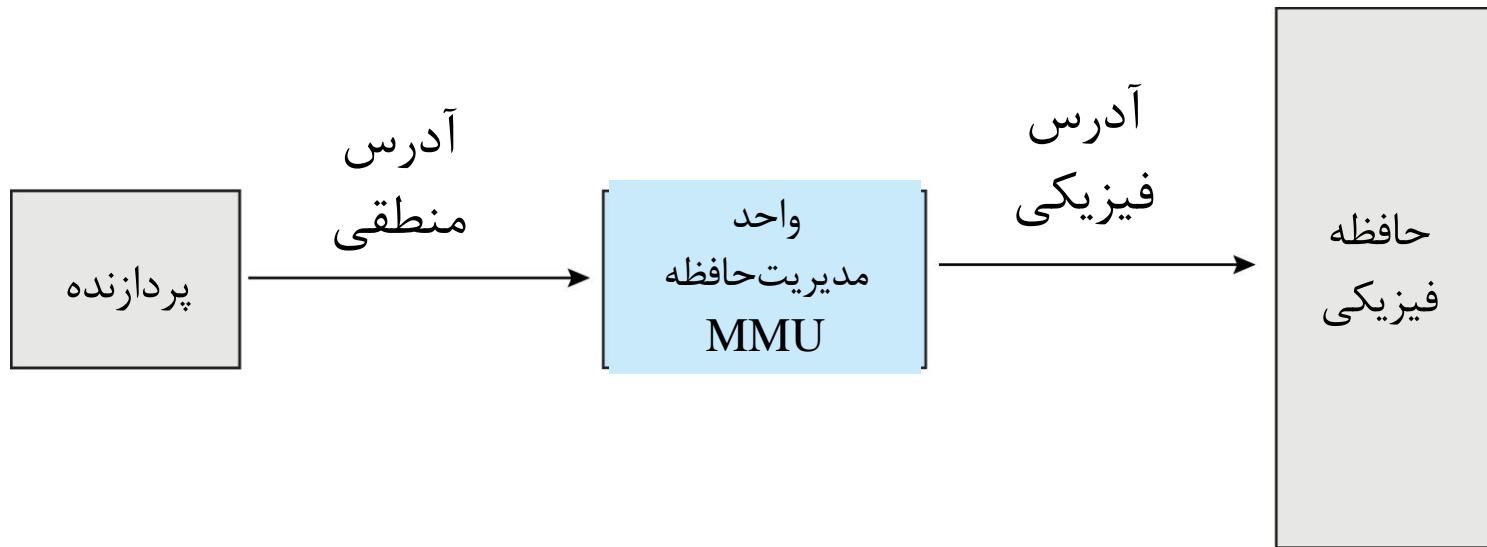
فضای آدرس فیزیکی و منطقی

- مفهوم فضای آدرس منطقی که به یک فضای آدرس فیزیکی مجزا متصل شده است، برای مدیریت صحیح حافظه ضروری است.
- **آدرس منطقی** (یا آدرس مجازی) : توسط پردازنده تولید می‌شود.
- **آدرس فیزیکی** : آدرسی که واحد حافظه می‌بیند.
- در الگوهای اتصال آدرس در زمان کامپایل و بارگذاری، آدرس‌های منطقی و فیزیکی یکسان هستند؛ در الگوهای اتصال آدرس در زمان اجرا، آدرس‌های منطقی (مجازی) و فیزیکی با هم تفاوت دارند.
- **فضای آدرس منطقی** : مجموعه‌ی تمام آدرس‌های منطقی تولید شده توسط یک برنامه است.
- **فضای آدرس فیزیکی** : مجموعه‌ی تمام آدرس‌های فیزیکی تولید شده توسط یک برنامه است.



واحد مدیریت حافظه (ادامه)

- یک سخت افزار در زمان اجرا، آدرس مجازی را به آدرس فیزیکی نگاشت می کند



- روش های زیادی برای این کار وجود دارد که در ادامه ای این فصل به آنها پرداخته می شود



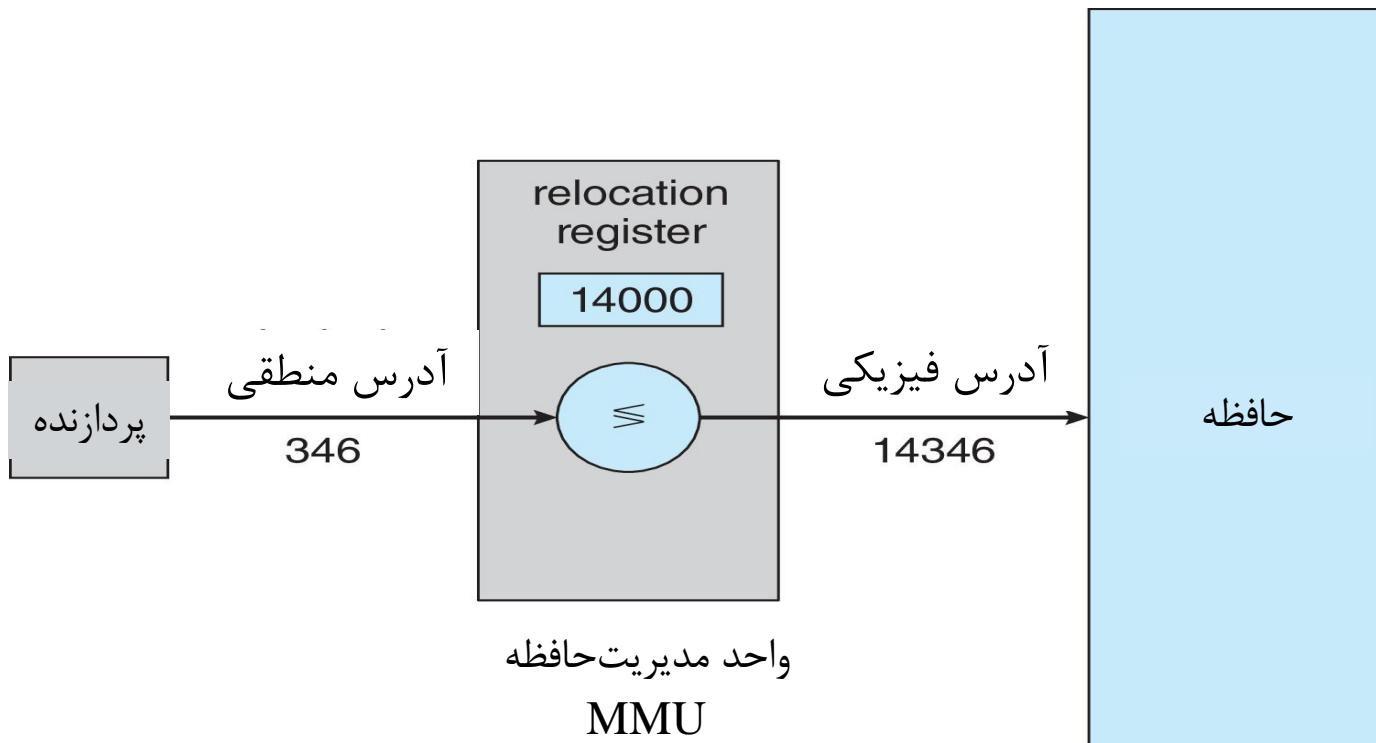
واحد مدیریت حافظه (ادامه)

- یک طرح ساده که تعمیمی از طرح رجیستر - پایه است را در نظر بگیرید:
- در این طرح، رجیستر پایه اکنون رجیستر جابجای Register نامیده می‌شود
- مقدار موجود در رجیستر جابجا بی به هر آدرسی که توسط یک فرآیند کاربر در زمان ارسال به حافظه تولید می‌شود، اضافه می‌شود.
- برنامه‌ی کاربر با آدرس‌های منطقی سروکار دارد و هرگز آدرس‌های فیزیکی واقعی را نمی‌بیند.
- اتصال آدرس در زمان اجرا زمانی اتفاق می‌افتد که به یک موقعیت در حافظه اشاره شود
- در این لحظه، آدرس منطقی به آدرس فیزیکی متصل می‌شود.



واحد مدیریت حافظه (ادامه)

- یک طرح ساده در نظر بگیرید که تعمیمی از طرح رجیستر پایه است:
- در این طرح، رجیستر پایه اکنون «رجیستر جابجایی (Relocation Register)» نامیده می‌شود
- مقدار موجود در رجیستر جابجایی relocation register به هر آدرسی که توسط یک فرآیند کاربر در زمان ارسال به حافظه تولید می‌شود، اضافه می‌شود.
- آدرس فیزیکی = آدرس منطقی + ثبات جابجایی





بارگزاری پویا Dynamic Loading

- در مدل سنتی اجرای برنامه، کل کد و داده‌های فرآیند باید به صورت کامل و همزمان در حافظه فیزیکی بارگذاری شوند تا اجرا شوند. این حالت را بارگذاری ایستا می‌نامند و اندازه برنامه را به اندازه حافظه فیزیکی محدود می‌کند.
- در مقابل، بارگذاری پویا (**Dynamic Loading**) به معنای آن است که تنها بخش اصلی برنامه ابتدا در حافظه بارگذاری می‌شود. سایر روال‌ها روی دیسک باقی می‌مانند و فقط زمانی که نیاز به اجرای آن‌ها باشد، به صورت پویا در حافظه لود می‌شوند.
- اگر روال مورد نظر قبلاً لود نشده باشد، لودر قابل جابه‌جایی آن را بارگذاری کرده، جدول آدرس‌ها را به روزرسانی می‌کند و سپس کنترل اجرای برنامه را به آن می‌دهد.



بارگزاری پویا Dynamic Loading

مزایای بارگزاری پویا به زبان ساده: ✓

- لازم نیست تمام برنامه از ابتدا در حافظه باشد؛ فقط بخش‌های لازم بارگزاری می‌شوند.
- هر روال (Routine) فقط وقتی که واقعاً نیاز باشد فراخوانی و بارگزاری می‌شود.
- این کار باعث می‌شود حافظه هدر نرود و فقط از بخش‌هایی استفاده شود که لازم‌اند.
- روال‌های بلااستفاده اصلاً وارد حافظه نمی‌شوند، چون نیازی به آن‌ها نیست.
- همه‌ی روال‌ها را می‌توان روی دیسک نگه داشت و هر وقت لازم بود، به حافظه آورد.
- مخصوصاً در شرایطی مفید است که حجم زیادی از کد داریم اما فقط گهگاهی استفاده می‌شود (مثلاً خطاهای خاص یا تنظیمات پیشرفته).
- این روش نیاز به تغییر خاصی در سیستم‌عامل ندارد و می‌تواند:
 - یا در خود طراحی برنامه لحاظ شود (توسط لودر دینامیکی)،
 - یا سیستم‌عامل کتابخانه‌هایی برای پشتیبانی از آن فراهم کند.



اتصال پویا Dynamic Linking

کتابخانه‌های پیوند پویا (DLL) به زبان ساده: ✓

- DLL‌ها مثل جعبه‌ابزارهایی هستند که برنامه‌ها فقط موقع نیاز، سراغشان می‌روند.
- برخلاف روش‌های قدیمی، این کتابخانه‌ها در زمان اجرا به برنامه وصل می‌شوند، نه از اول داخلش باشند.

•  مزایای اصلی:

- کاهش حجم برنامه‌ها: چون توابع مشترک بین چند برنامه فقط یک بار در DLL قرار دارند، برنامه‌ها سبک‌تر می‌شوند.
- صرفه‌جویی در حافظه: فقط یک نسخه از DLL در حافظه بار می‌شود و چند برنامه می‌توانند همزمان از آن استفاده کنند.
- به روزرسانی آسان: با آپدیت کردن خود DLL، همه‌ی برنامه‌های وابسته به طور خودکار از نسخه جدید استفاده می‌کنند. نیازی به دستکاری تک‌تک برنامه‌ها نیست.



■ کتابخانه های پیوند پذیر (DLL)

- مشابه مازول های شیء - توسط لودر در تصویر نهایی برنامه (باینری) ترکیب می شوند.
- پیوند دینامیک (برخلاف پیوند ایستا):
 - پیوند به زمان اجرا موکول می شود (شبیه بارگذاری پویا).
 - کاربرد : کتابخانه های سیستمی (مثل کتابخانه استاندارد زبان C)
- مزایای DLL ها:
 - کاهش حجم برنامه : هر برنامه نیازی به کپی جداگانه از کتابخانه ندارد.
 - بهینه سازی حافظه : یک نسخه از DLL در حافظه اصلی برای چندین فرایند قابل اشتراک است.
 - کاربرد گسترده : کتابخانه های DLL در سیستم عامل های ویندوز و لینوکس استفاده می شوند .



مرحله اول: زمان کامپایل (Compile Time)

1. برنامه‌ی منبع (source program): کد برنامه

2. کامپایلر یا اسambilر (compiler or assembler): کامپایلر یا اسambilر

این ابزار کدی که شما نوشتید را به زبان ماشین (که برای کامپیوتر قابل فهمه) تبدیل می‌کنه. نتیجه‌اش میشه یک فایل به نام شیء (object module) که هنوز قابل اجرا نیست.

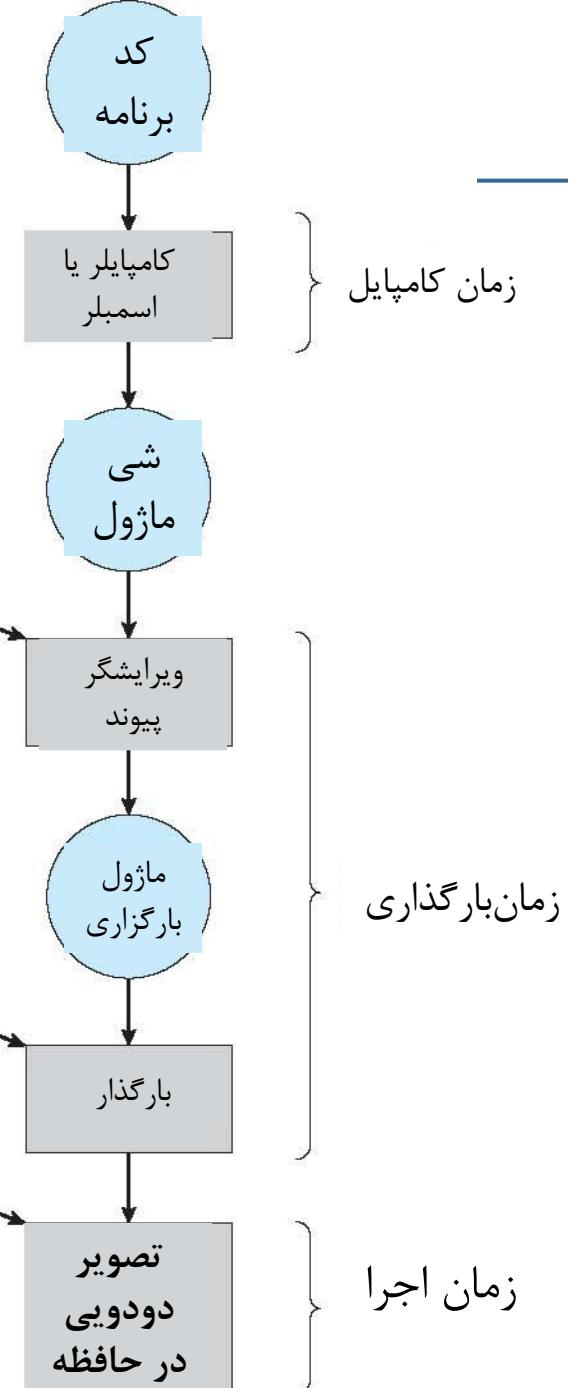
مرحله دوم: زمان بارگذاری (Load Time)

3. ویرایشگر پیوند (linkage editor):

اینجا فایل شیء شما به همراه بقیه‌ی مژوول‌های شیء (مثلاً کتابخانه‌ها یا بخش‌های دیگهه‌ی برنامه) با هم ترکیب می‌شن و یک فایل نهایی به نام مژوول بارگذاری (load module) ساخته می‌شه.

4. کتابخانه‌ی سیستم (system library):

این کتابخانه‌ها شامل کدهایی هستن که خیلی از برنامه‌ها ازشون استفاده می‌کنن (مثلاً توابع استاندارد مثل `printf()` در C). این‌ها هم در این مرحله به مژوول شما اضافه می‌شن.





تخصیص پیوسته Contiguous Allocation

 مدیریت حافظه اصلی :حافظه اصلی باید هم برای سیستم عامل جا داشته باشد و هم برای برنامه‌های کاربر.

- چون حافظه محدود است، باید هوشمندانه و کارآمد بین این دو تقسیم شود.

 تخصیص پیوسته (Contiguous Allocation): یکی از روش‌های قدیمی و ساده برای مدیریت حافظه است. در این روش، هر برنامه در یک قطعه‌ی پشت‌سرهم (پیوسته) از حافظه قرار می‌گیرد.

 تقسیم حافظه :معمولًاً حافظه به دو بخش کلی تقسیم می‌شود:

- هسته سیستم عامل (Kernel): اغلب در بالای حافظه قرار می‌گیرد (مثل ویندوز و لینوکس).

- فرآیندهای کاربر: در پایین حافظه جای می‌گیرند.

این جای‌گذاری به عواملی مثل محل بردار وقفه (interrupt vector) بستگی دارد.



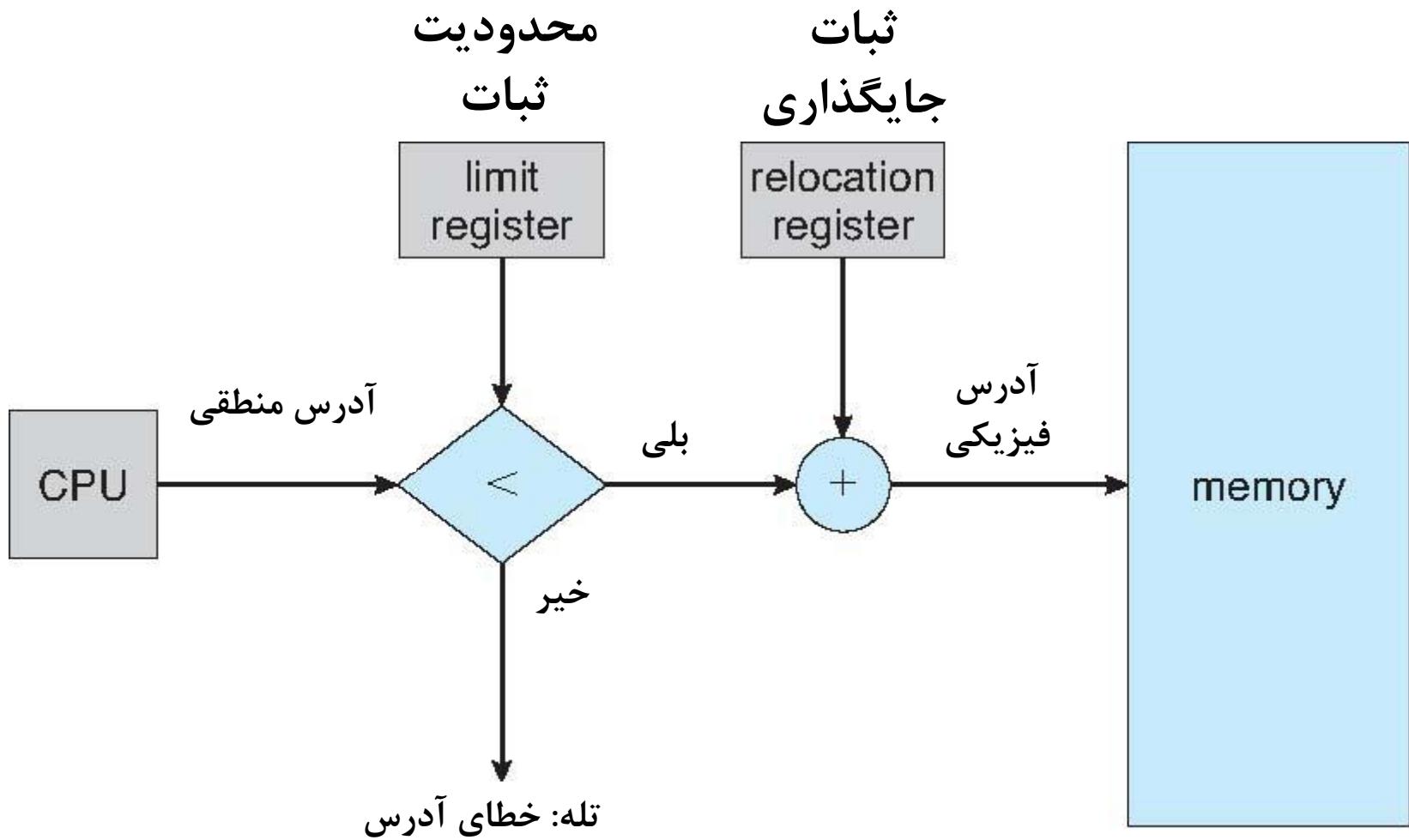
تخصیص پیوسته (ادامه)

رجیسترهاي جابجايی (Relocation Registers) : اين رجيسترها کمک می کنند که فرآيندهای کاربر نتوانند به هم یا به سیستم عامل آسیب بزنند.

- مثل نگهبان‌هایی هستند که تعیین می‌کنند یک برنامه تا کجا اجازه دسترسی به حافظه دارد.
- واحد مدیریت حافظه (MMU) آدرس‌های منطقی را به صورت پویا به آدرس‌های فیزیکی ترجمه می‌کند.
- این روش اجازه می‌دهد:
 - کد هسته موقتی باشد و در زمان لازم (بخش‌های غیرضروری هسته) حذف شود.
 - اندازه هسته در صورت نیاز تغییر کند، بدون آسیب به برنامه‌های دیگر.



Hardware Support for Relocation and Limit Registers



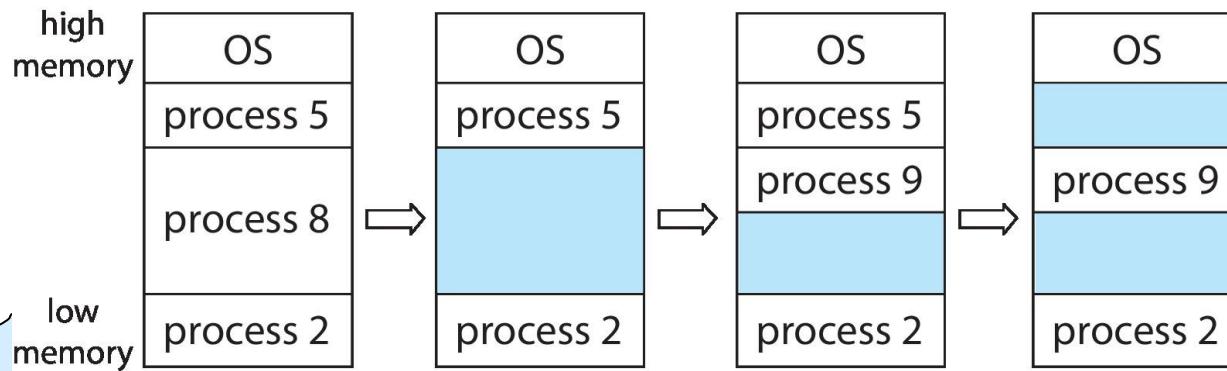


اندازه پارتیشن متغیر

تخصیص چندبخشی: (Multiple-Partition Allocation):

- حافظه به چند قسمت (پارتیشن) تقسیم می‌شود و هر برنامه در یک بخش جداگانه اجرا می‌شود.
- اما تعداد برنامه‌هایی که می‌توانند همزمان اجرا شوند، محدود به تعداد پارتیشن‌ها است.
- در این روش، درجه چندرنامگی (Multiprogramming) توسط تعداد پارتیشن‌ها محدود می‌شود.
- برای کارایی بیشتر، **اندازه‌های پارتیشن‌ها** متغیر هستند (متناسب با نیازهای یک فرآیند خاص).

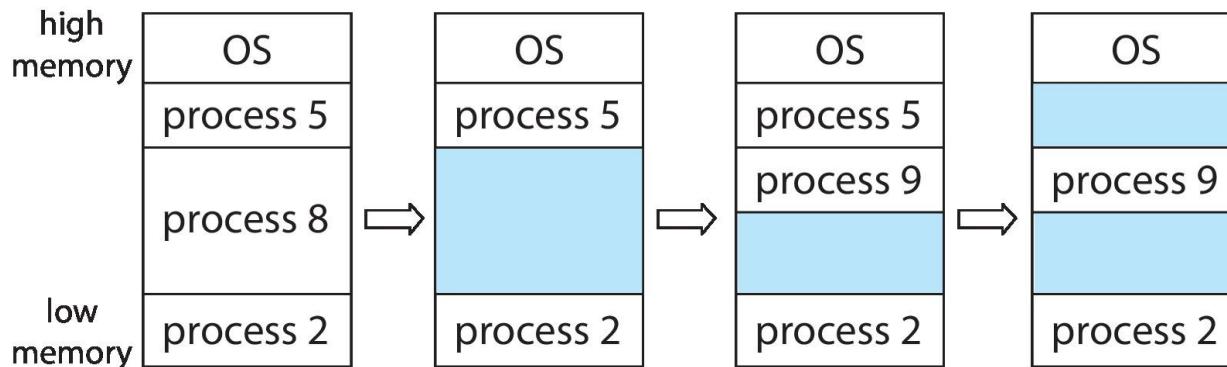
- فضای خالی (Hole):** بلوکی از حافظه در دسترس؛ سوراخ‌ها با اندازه‌های مختلف در سراسر حافظه پراکنده‌اند.





اندازه پارتیشن متغیر

- هنگامی که فرآیندی وارد می‌شود، حافظه‌ای از یک فضای خالی به اندازه‌ی کافی برای جای دادن آن به آن اختصاص داده می‌شود.
- خارج شدن یک فرآیند، پارتیشن آن را آزاد می‌کند و پارتیشن‌های مجاور آزاد با هم ترکیب می‌شوند.
- سیستم عامل اطلاعات زیر را نگهداری می‌کند:
 - الف) پارتیشن‌های اختصاص یافته ب) پارتیشن‌های آزاد : فضاهای خالی





مشکل تخصیص پویای حافظه Dynamic Storage-Allocation Problem

چگونه می‌توان درخواست فضای به اندازه‌ی n از لیستی از سوراخ‌های آزاد را برآورده کرد؟

- اولین برازش **(First-fit)**: اختصاص اولین فضای خالی که به اندازه‌ی کافی بزرگ است.

بهرترین برازش **(Best-fit)**: اختصاص کوچکترین فضای خالی که به اندازه‌ی کافی بزرگ است؛ مگر اینکه بر اساس اندازه مرتب شده باشند، کل لیست را باید جستجو کرد.
این روش کوچکترین فضای خالی باقی‌مانده را ایجاد می‌کند.

- بدترین برازش **(Worst-fit)**: اختصاص بزرگترین فضای خالی؛ همچنین باید کل لیست را جستجو کرد.

• این روش بزرگترین فضای خالی باقی‌مانده را ایجاد می‌کند.

اولین برازش و بهرترین برازش از نظر سرعت و استفاده از حافظه نسبت به بدترین برازش بهتر عمل می‌کنند



پراکندگی Fragmentation

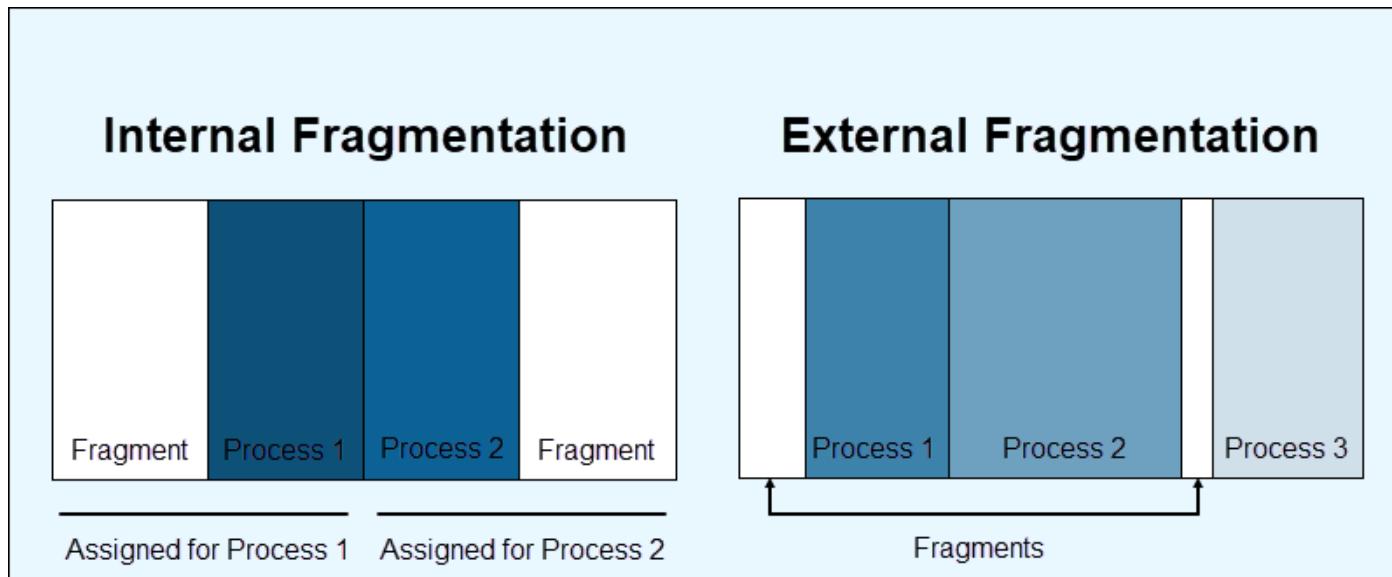
- فرض کنید یک طرح تخصیص چندبخشی (Multiple-Partition Allocation) با فضایی خالی به اندازه ۱۸۴۶۴ بایت داریم.
- اگر فرآیند بعدی ۱۸۴۶۲ بایت حافظه درخواست کند، چه اتفاقی می‌افتد؟
 - اگر دقیقاً همان بلوک درخواستی را اختصاص دهیم، با یک فضای خالی ۲ بایتی باقی می‌مانیم.
 - مدیریت این فضای خالی کوچک، از خود آن فضا، کارایی کمتری خواهد داشت.
- پراکندگی خارجی (External Fragmentation):
 - زمانی رخ می‌دهد که فضای خالی کافی در حافظه وجود داشته باشد، اما این فضا به قطعات کوچک غیرقابل استفاده تقسیم شود.
 - هیچ یک از قطعات به تنها یی برای برآورده کردن نیازهای حافظه یک فرآیند جدید کافی نیست.



پراکندگی Fragmentation

▪ پراکندگی داخلی (Internal Fragmentation): حافظه‌ی اختصاص‌یافته ممکن است کمی بزرگ‌تر از حافظه‌ی درخواست‌شده باشد؛ این اختلاف اندازه، حافظه‌ای داخلی یک پارتیشن است که استفاده نمی‌شود.

▪ تحلیل اولین برازش نشان می‌دهد که با اختصاص N بلوک، تقریباً نیم $\frac{N}{5}$ دیگری از بلوک‌ها دست می‌روند که تقریباً یک سوم حافظه می‌شود که غیرقابل استفاده می‌شود. این خاصیت به قانون ۵-۵ درصد شناخته می‌شود.





پراکندگی (ادامه)

❖ فشرده‌سازی حافظه چیست؟ وقتی حافظه دچار پراکندگی خارجی (فضاهای خالی کوچک و جدا از هم) شود، می‌توان با فشرده‌سازی (Compaction) این مشکل را کاهش داد.

✓ فشرده‌سازی یعنی:

- جابجا کردن برنامه‌ها در حافظه برای اینکه همه فضای آزاد در یک بلوک پیوسته جمع شود.
- شبیه این است که کتاب‌های پراکنده روی قفسه را کنار هم بچینیم تا فضای خالی وسطها از بین برود.

⚠ اما فقط وقتی ممکن است که:

- سیستم اجازه دهد آدرس حافظه در حین اجرا تغییر کند. (dynamic relocation).
- یعنی برنامه‌ها بتوانند بعد از شروع هم، جایشان را در حافظه عوض کنند.



پراکندگی (ادامه)

مشکل در عملیات O/I ورودی/خروجی:

وقتی برنامه‌ای در حال استفاده از دستگاه‌های O/I (مثلاً دیسک یا چاپگر) است، جابه‌جا کردن آن در حافظه ممکن است باعث از دست رفتن داده‌ها شود.

راه حل‌ها:

1. نگه داشتن کل برنامه در حافظه (امن ولی حافظه‌بر)
2. استفاده از بافرهایی که توسط سیستم عامل مدیریت می‌شوند: رویکرد بهتر و رایج‌تر



صفحه بندی

■ فضای آدرس فیزیکی یک فرآیند می‌تواند غیرپیوسته باشد. به این معنی که بخش‌های مختلف حافظه به فرآیند اختصاص داده می‌شود و این بخش‌ها الزاماً مجاور هم قرار ندارند، به شرطی که فضای خالی در دسترس باشد. این روش مزایای زیر را به همراه دارد:

- جلوگیری از پراکندگی خارجی حافظه (External Fragmentation)
- سازگاری با اندازه‌های مختلف حافظه: این روش دیگر با مشکل قطعه‌های حافظه با اندازه‌های نامناسب مواجه نیست و می‌تواند حافظه را به صورت بهینه‌تری مدیریت کند.



مراحل صفحه‌بندی

- ۱. تقسیم حافظه اصلی: حافظه اصلی به بلوک‌های با اندازه ثابت به نام **فریم (Frame)** تقسیم می‌شود. اندازه فریم معمولاً توانی از ۲ است و مقداری بین ۵۱۲ تا ۱۶ مگابایت دارد.
- ۲. تقسیم حافظه منطقی: حافظه منطقی فرآیند نیز به بلوک‌های با اندازه مشابه به نام صفحه (Page) تقسیم می‌شود.
- ۳. ردیابی فریم‌های خالی: سیستم‌عامل لیستی از تمام فریم‌های خالی در حافظه را نگهداری می‌کند.
- ۴. اجرای برنامه: برای اجرای یک برنامه با اندازه N صفحه، سیستم‌عامل باید N فریم خالی پیدا کند و برنامه را در آن فریم‌ها بارگذاری نماید.
- ۵. جدول صفحه‌بندی: برای ترجمه آدرس‌های منطقی برنامه به آدرس‌های فیزیکی حافظه، از یک جدول صفحه‌بندی (Page Table) استفاده می‌شود.
- ۶. حافظه جانبی (Backing Store): حافظه جانبی نیز به صفحاتی با اندازه مشابه تقسیم می‌شود. این صفحات محل ذخیره شدن برنامه‌هایی هستند که در حال حاضر در حال اجرا نیستند.
- ۷. پراکندگی داخلی: با اینکه صفحه‌بندی خیلی مفید است، ولی گاهی فضای داخل یک فریم به طور کامل استفاده نمی‌شود و این باعث هدر رفتن مقداری از حافظه می‌شود. به این مشکل "پراکندگی داخلی" می‌گویند.



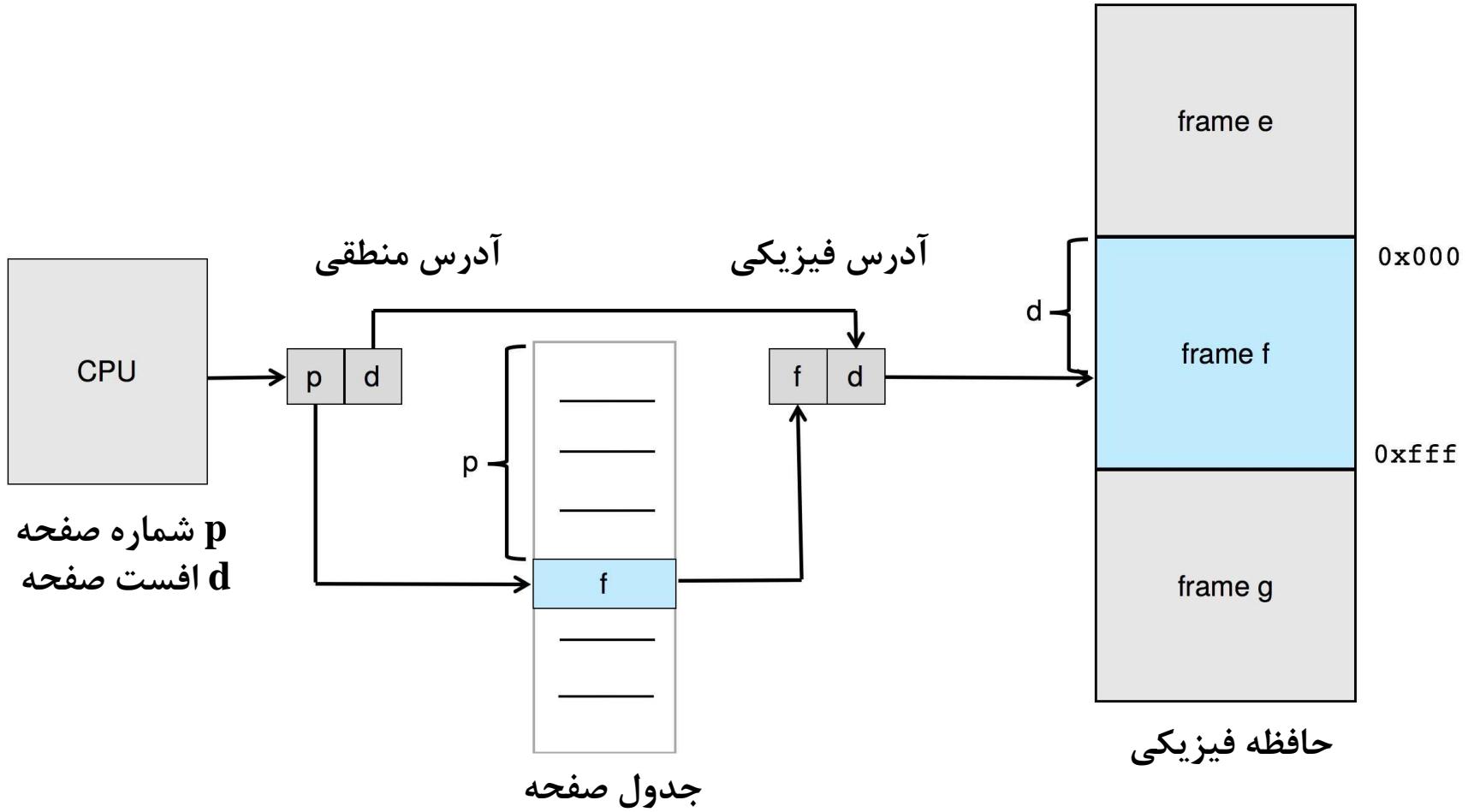
الگوی ترجمه آدرس

- وقتی پردازندۀ یک آدرس منطقی تولید می‌کند، این آدرس به دو بخش تقسیم می‌شود:
- شماره صفحه (p) : به عنوان یک شاخص در جدول صفحه عمل می‌کند که حاوی آدرس پایه هر صفحه در حافظه فیزیکی است.
- محدوده صفحه یا آفست (فاصله از مبدأ) صفحه (d) : این مقدار با آدرس پایه ترکیب می‌شود تا آدرس نهایی حافظه فیزیکی را که به واحد حافظه ارسال می‌گردد، تعیین کند.
- فرض کنید فضای آدرس منطقی 2^n بایت و اندازه صفحه 2^m بایت باشد

شماره صفحه	محدوده صفحه
p	d
$m - n$	n

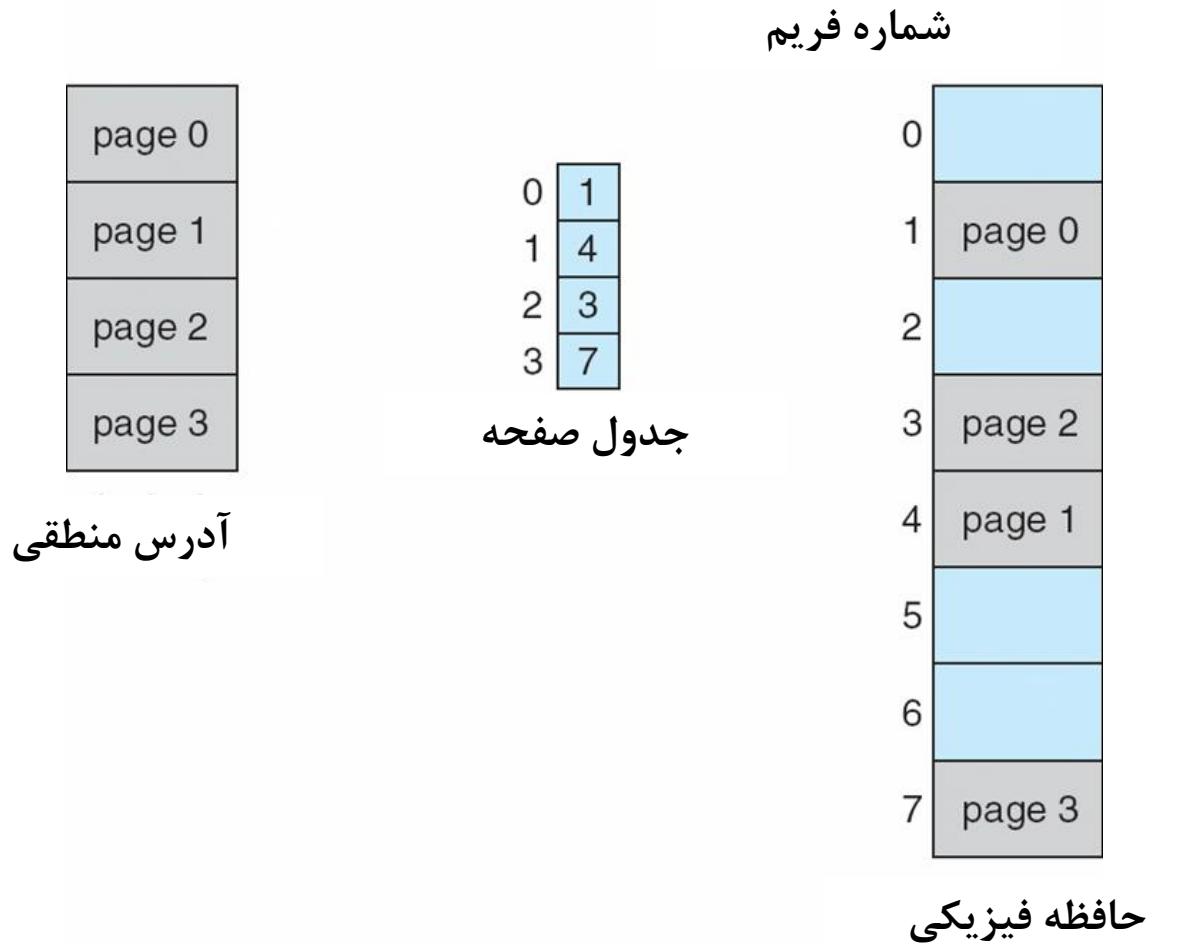


Paging Hardware



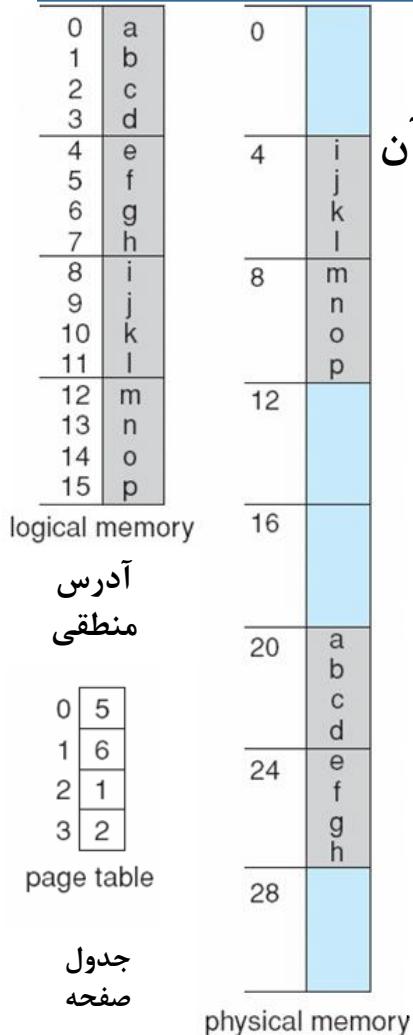


مدل صفحه‌بندی Paging Model برای حافظه منطقی و فیزیکی





مثال صفحه بندی



مثال ساده از صفحه بندی (Paging)

در این مثال، ما داریم یک برنامه را اجرا می‌کنیم و می‌خواهیم ببینیم آدرس‌های منطقی آن چطور به آدرس‌های فیزیکی نگاشت می‌شوند.

- اندازه هر صفحه $4 = 2^2$ بايت - اندازه کل حافظه فیزیکی $32 = 2^5$ بايت
- بنابراین حافظه فیزیکی شامل $(32/4) = 8$ فریم ۴ بایتی است.



اجزا:

- آدرس منطقی (**Logical Address**): چیزی است که برنامه تولید می‌کند.

- صفحه (**Page**): هر آدرس منطقی به یک صفحه و یک افست (**Offset**) تقسیم می‌شود.

- جدول صفحه‌بندی (**Page Table**): سیستم‌عامل از این جدول برای تبدیل آدرس منطقی به فیزیکی استفاده می‌کند.

- آدرس فیزیکی (**Physical Address**): آدرسی است که در حافظه واقعی (**RAM**) حافظه فیزیکی استفاده می‌شود.



مثال صفحه بندی

چند مثال از نگاشت آدرس‌ها:

آدرس منطقی ۰: صفحه = ۰، افست = ۰ ✓

جدول صفحه‌بندی می‌گوید: صفحه‌ی ۰ ← فریم ۵

آدرس فیزیکی $(5 * 4) + 0 = 20$

آدرس منطقی ۳: صفحه = ۰، افست = ۳ ✓

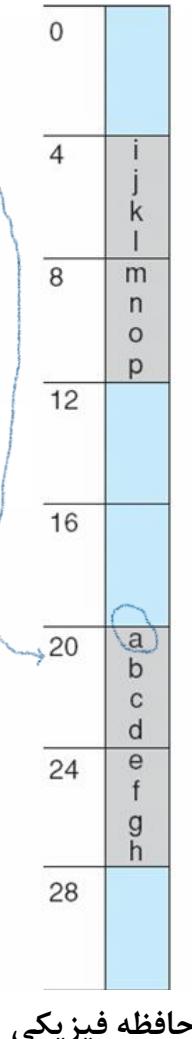
صفحه‌ی ۳ هنوز هم در فریم ۵ است

آدرس فیزیکی $(4 * 5) + 3 = 23$

آدرس منطقی ۴: صفحه = ۱، افست = ۰ ✓

جدول صفحه‌بندی می‌گوید: صفحه‌ی ۱ ← فریم ۶

آدرس فیزیکی $(4 * 6) + 0 = 24$





صفحه‌بندی - محاسبه پراکندگی داخلی

- فرض کنید اندازه هر صفحه حافظه ۲۰۴۸ بایت است و برنامه‌ای داریم با اندازه ۷۲,۷۶۶ بایت.

چه تعداد صفحه نیاز داریم؟ 

اگر ۷۲,۷۶۶ را بر ۲۰۴۸ تقسیم کنیم، حدوداً ۳۵,۵۳ به دست می‌آید.
این یعنی: ۳۵ صفحه کامل پر می‌شوند ($35 \times 2048 = 71,680$ بایت)

- و ۱۰۸۶ بایت باقی‌مانده در یک صفحه جدید قرار می‌گیرد.
- در مجموع، این فرآیند به ۳۶ صفحه نیاز دارد.

پراکندگی داخلی (Internal Fragmentation) : یعنی مقداری از حافظه داخل صفحه استفاده نمی‌شود و هدر می‌رود. 

- در این مثال: اندازه صفحه = ۲۰۴۸ بایت
- فقط ۱۰۸۶ بایت از آخرین صفحه استفاده شده پس ۹۶۲ بایت فضای استفاده نشده باقی مانده است که این مقدار پراکندگی داخلی است.



صفحه‌بندی – محاسبه پراکندگی داخلی

!
حالتهای خاص:

- بدترین حالت: فقط یک بایت از آخرین صفحه استفاده شود، یعنی تقریباً کل صفحه هدر می‌رود.
- میانگین پراکندگی: معمولاً انتظار داریم نصف یک صفحه هدر برود (در اینجا حدود ۱۰۲۴ بایت).

چرا اندازه صفحه مهم است؟

انتخاب اندازه صفحه یک موازن است:

- اگر صفحات خیلی کوچک باشند، جدول صفحه‌بندی بزرگ‌تر می‌شود و حافظه بیشتری مصرف می‌شود.
- اگر صفحات خیلی بزرگ باشند، ممکن است پراکندگی داخلی زیاد شود.
- به همین دلیل در طول زمان، اندازه صفحات به تدریج بزرگ‌تر شده تا تعادل بین این دو برقرار شود.



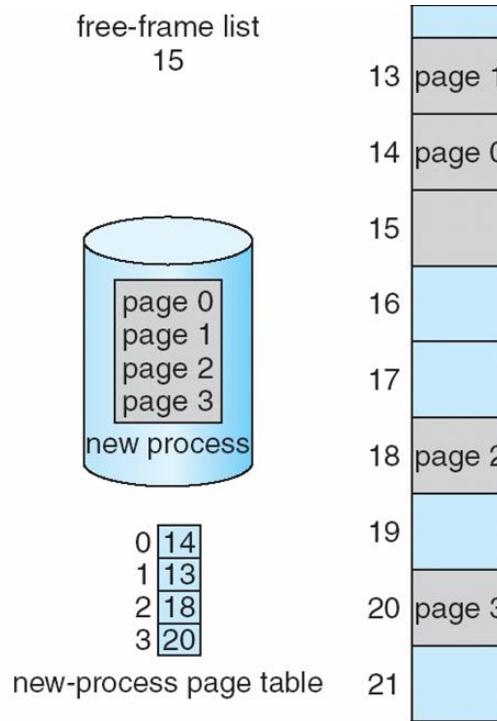
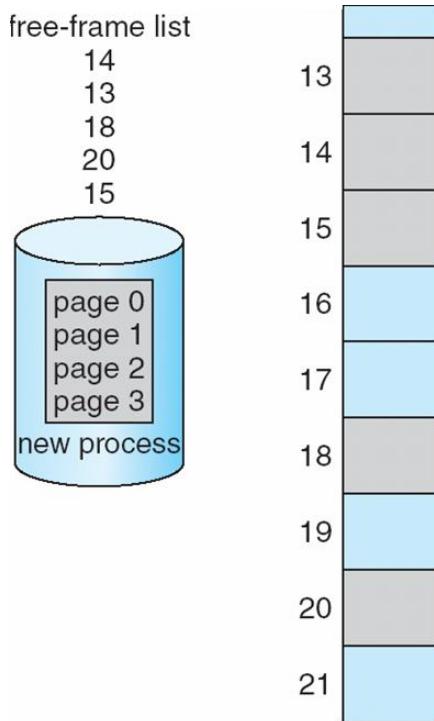
فریم‌های آزاد

نحوه تخصیص فریم‌ها به صفحات یک فرآیند (Page Allocation)

فرض کنیم یک فرآیند جدید داریم که به ۴ صفحه Page ۰ تا Page ۳ نیاز دارد.

قانون اصلی: اگر فرآیند به n صفحه نیاز دارد، سیستم باید حداقل n فریم خالی در حافظه داشته باشد تا بتواند این صفحات را بارگذاری کند.

لیست فریم‌های خالی



قبل از تخصیص

بعد از تخصیص

تخصیص فریم‌ها به صفحات:

به ترتیب، صفحات فرآیند در فریم‌های خالی
بارگذاری می‌شوند:

Page 0 → Frame 14•

Page 1 → Frame 13•

Page 2 → Frame 18•

Page 3 → Frame 20•

این اطلاعات در جدول صفحه برای این فرآیند ذخیره می‌شود:



End of Essential part