

Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department
1401

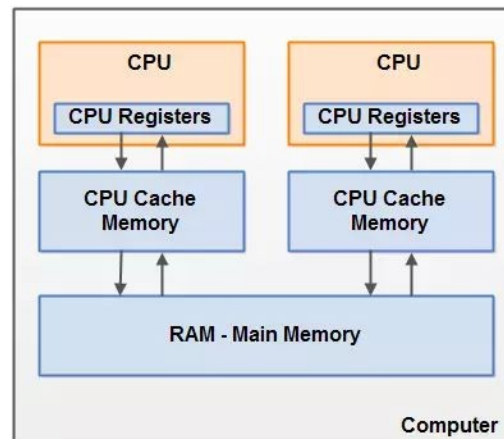
Zeinab Zali

Memory Management



Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- **Main memory** and **registers** are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- But Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation



هر برنامه ای وقتی که بخواد اجرا بشه و تبدیل بشه و تبدیل به یک پروسس بشه یک فضایی رو در حافظه اصلی سیستم یی در مین مموری خواهد گرفت یی در مین مموری یک فضا برایش در نظر گرفته میشه که کدش، دیتاهاش، استک و هیپ و ... باید در یک قسمت هایی از مموری قرار بگیره

cpu وقتی که میخواد دستورات یک برنامه رو اجرا بکنه دسترسی مستقیمش به رجیسترهای cpu است اگر بخواد مقادیری رو داشته باشه یا به رجیسترها دسترسی داره یا به مین مموری از طرفی مموری اون چیزی که می بینه یک ادرس است و درخواستی که روی اون ادرس است مثلا ممکنه درخواست خواندن از یک ادرس مموری رو داشته باشیم یا اینکه بخوایم یک مقداری رو توی این ادرس قرار بدیم

توی شکل: cpu مستقیم به رجیسترهای خودش دسترسی داره و این دسترسی خیلی سریع است در حد یک کلاک cpu ولی وقتی که میخواد یک مقداری رو از مموری برداره یی مین مموری دسترسی بهش کندتر از دسترسی به رجیسترها است--> حتی برای کش هم همین موضوع برقراره که توی این شکل چه کش و چه مین مموری دسترسی بهشون کندتر است

اگر cpu بخواد به مین مموری دسترسی داشته باشه چندین سیکل از cpu رو نیاز داره که بتونه با اون قسمت از مموری دسترسی پیدا کنه

نکته: توی اون سیکل هایی که cpu داره مقداری رو از مموری برمیداره یا کلا دسترسی به مموری داره بهش میگیم stall --> در واقع دسترسی به مموری باعث stall میشه بخاطر اینکه یک زمان قابل توجهی رو می بره حتی بیشتر از یک کلاک cpu بخاطر همین است که ما کش رو بین رجیسترها و مین مموری قرار میدیم که باز دسترسی به کش سریعتر از مین مموری است ولی کندتر از رجیسترها است و فضای بزرگتری داره نسبت به رجیسترها پس از این کش استفاده میشه که قسمت هایی از مموری که در هر بازه زمانی بیشتر بهش مراجعه میشه توی این قسمت کش ذخیره بشه تا از دسترسی به مین مموری جلوگیری بشه

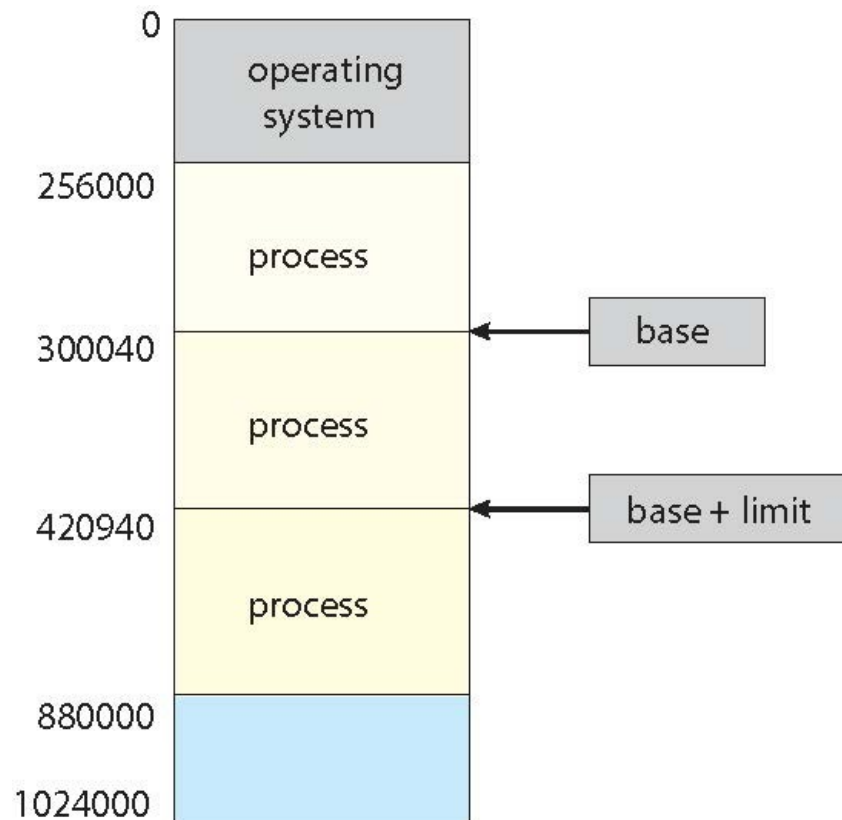
اینجا میخوایم در مورد مین مموری صحبت کنیم --> یی فرض میکنیم puc میخواد دسترسی به ادرس های مموری داشته باشه تا اینکه مقادیری رو از ادرس های مموری بخونه یا مقادیری رو توی اونها قرار بده

و چیزی که مهمه --> Protection این مموری برای ما خیلی مهمه و باید مطمئن باشیم که مقادیر به درستی توی مموری قرار میگیره یی مثلا مقادیر توی ادرس های اشتباه قرار نمی گیره یا روی هم قرار نمیگیره بنابراین ما دسترسی مستقیم از لایه یوزر نباید داشته باشیم به این مموری



Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check **every memory access generated in user mode** to be sure it is between base and limit for that user



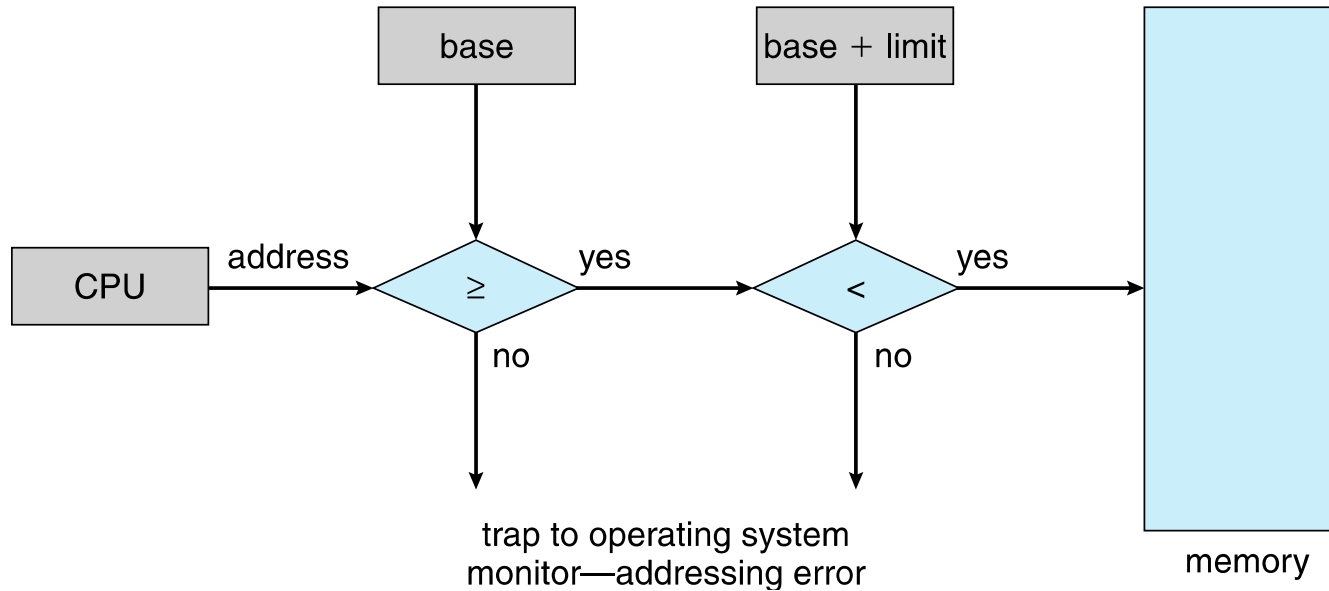
اگر خیلی ساده فکر کنیم می‌تونیم فضای حافظه هر پروسس رو پشت سرهم در نظر بگیریم مثل اینجا 3 تا پروسس توی حافظه رم قرار گرفته

وقتی **cpu** مشغول اجرای یکی از این پروسس‌ها است یه دایره دستورات اون پروسس مشخص رو فچ میکنه و به دیتاها، استک و هیپ ایناش دسترسی دایره برای همین باید **cpu** بدونه که باید کجا دنبال دستورات و دیتاهای مربوط به این پروسس بگرده توی حافظه اصلی به همین دلیل ما معمولا دوتا رجیستر داریم توی **cpu** یکی به نام رجیستر **base** که میخواد نشون بده اولین خونه ای که مربوط به این پروسس توی حافظه است ادرشش چنده و یکی هم رجیستر **limit** است که مشخص میکنه که این پروسس چقدر فضا گرفته توی حافظه

این رجیسترهای **base** , **limit** در روش‌های امروزی هم باز استفاده میشه که اونجاها مفهوم خودش رو دایره



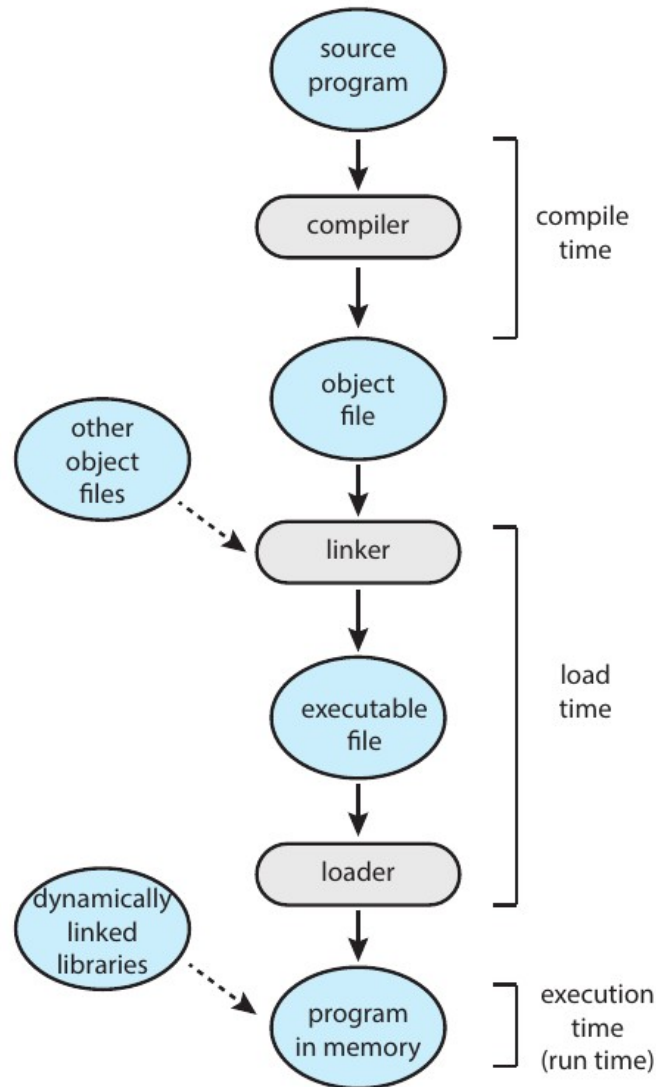
Hardware Address Protection



حالا اگر ما خواستیم به یک قسمتی از این فضای پروسس توی مموری دسترسی پیدا کنیم cpu ادرسی رو می بینیم که باید چک کنه که ایا این ادرس از ادرس base این پروسس بیشتر است یا نه و همینطور باید مطمئن هم بشیم که این ادرس از ادرس base + limit هم کوچکتر است که این کار به نوعی Protection برای ما فراهم میکنه و مانع این میشه که وقتی که cpu داره این پروسس رو اجرا میکنه این پروسس بتونه دسترسی به ادرس های پروسس های دیگه هم داشته باشه ینی جلوگیری میکنه از این که بتونه دسترسی به ادرس پروسس های دیگه داشته باشه و Protecte میکنه از پروسس های دیگه



Multistep processing of a user program



وقتی که یک برنامه داریم برای ادرس دهی از یکسری متغیر استفاده می کنیم ینی مستقیما با ادرس ها کاری نداریم --> به طور کلی در یک برنامه وقتی که ما کد می نویسیم از متغیر استفاده میکنیم و با ادرس ها کاری نداریم

وقتی که برنامه ما کامپایل میشه چه اتفاقی می افته؟ اون ادرس های سمبلیک که متغیرها هستن تبدیل میشن به یک ادرسی به نام **relocatable** ادرس چرا بهش میگیم **relocatable**؟ بخاطر اینکه در واقع ادرسی هست که هر جای حافظه اصلی ممکنه قرار بگیره ینی امکان جابه جایش وجود داره ینی همچنان نرسیدیم به اون ادرس واقعی حافظه ینی حتی وقتی که برنامه کامپایل میشه توی کامپایل شده برنامه یکسری ادرس هایی می بینیم که در واقع **relocatable** هستن نکته: ادرس هایی که تولید می کنیم بعد از کامپایل ادرس هایی هست که نسبت به صفر خود اون پروسس در نظر گرفته میشه

وقتی می خوایم برنامه ای رو اجرا کنیم ینی **load** اش میکنیم اون برنامه رو اینجا چه اتفاقی میافته؟ **loader** باید یک همچین کاری بکنه که این ادرس های **relocatable** رو که به نسبت به خود پروسس بوده یه جوری تبدیلش بکنه به ادرس های رم مثال:

یک برنامه **source** داریم کامپایل میشه و یک تعدادی ابجکت فایل ازش ساخته میشه حالا یک لینکری داریم که این ابجکت فایل های مختلف رو با هم لینک میکنه و نهایتا یک **executable** فایل یا یک باینری فایل تولید میشه که **loader** سیستم میتونه اون باینری فایل رو لود کنه و در اخر برنامه توی مموری قرار بگیره

حتی وقتی که برنامه توی مموری هست یکسری **dynamica library** هم ممکنه لینک بشن به این برنامه ای که ما نوشتیم ینی این **library** ها جز برنامه اصلی ما نیستن ولی موقع اجرا لینک میشن به این برنامه و قابل اجرا میشن



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at different steps
 - **Compile time:** if at some later time the starting location changes, then recompiling is necessary
 - **Load time:** binding relocatable addresses (from compile time) to absolute addresses
 - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
 - ▶ Need hardware support for address maps (e.g., base and limit registers)



ما کی باید دستور ها و دیتاها رو واقعا به ادرس های واقعی مموری bind کنیم؟

این ها به صورت منطقی توی فضای ادرس پروسس قرار دارن --> اول که ما برنامه رو مینویسیم متغیر هستن و بعد هم می شن relocatable ادرس و حالا کی میخوان واقعا تبدیل بشن به ادرس های مموری واقعی؟

اگر ما بیایم موقع Compile time این کارو بکنیم چه اتفاقی می افته؟ ینی ما relocatable ادرس ایجاد نکنیم توی Compile time و ادرس های واقعی مموری رو ایجاد کنیم ینی توی همین مرحله که داریم کامپایل میکنیم به جای اینکه relocatable ادرس داشته باشن دستورهایی که توی این فایل باینری هستن ادرس واقعی داشته باشن بعد این چه اتفاقی می افته؟ ینی چه مشکل هایی خواهد داشت؟ اگر ما دقیق ادرس مشخص بکنیم توی کامپایل تایم ینی اون پروسسمون رو میخوایم خیلی دقیق ببریم توی یک قسمت مشخصی از رم قرار بدیم و اگر ادرس ها رو مشخص کرده باشیم ینی دیگه باید طبق همون ادرس هایی که توی Compile time تولید شده اینو ببریمش توی رم و این خیلی بده --> ینی اینکه هرباری که برنامه رو اجرا میکنیم دقیقا توی یک جای مشخصی از رم اون برنامه باید لود بشه و این توی یک سیستم مالتی تسک خیلی مشکل ایجاد میکنه و خیلی واضحه که غیرقابل اجراست پس اینجا جای مناسبی نیست برای این bind کردن

Load time: اگر توی Load time بخوایم این کارو بکنیم ینی relocatable ادرس ها رو توی Load time به absolute ادرس تبدیل کنیم --> اینجا چه اتفاقی می افته؟ ایا خوبه یا نه؟ اگر اینجا ین کارو کردیم دیگه وقتی که برنامه اومد توی مموری دیگه نمی تونیم جابه جاش کنیم ینی از ابتدای اجرای یک برنامه تا انتهای اجرا باید دقیقا توی یک جای مشخصی از حافظه باشه اگر این کارو نکردیم و گذاشتیم این کارو برای Execution time چی میشه حالا

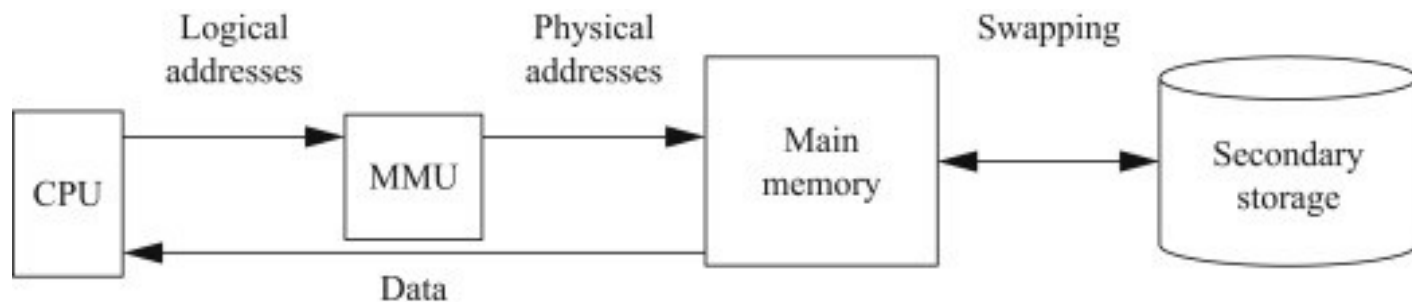
Execution time: ینی توی زمان اجرا این ها رو مشخص کردیم --> اگر بخوایم یک همچین کاری رو بکنیم نیاز به یکسری امکانات سخت افزاری هم در کنار روش های نرم افزاریمون داریم که بشه این کارو کرد و مزیتش این است که در این صورت ما می تونیم موقعی که برنامه داره اجرا میشه هم حتی برنامه رو جابه جا کنیم توی حافظه

نکته: جاهایی ما نیاز داریم که حتی توی Execution time بتونیم برنامه ای رو جابه جا کنیم --> یکی از زمان هایی که نیاز به این کارو انجام بدیم موقعی است که ما swapping داریم



Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses corresponding to the program logical addresses



دو نوع ادرس داریم:

یکی ادرس هایی مثل اون relocatable ادرس که ادرس های Logical هستند و یکی ادرس واقعی که Physical ادرسه ینی ادرس واقعی مین مموری است که مشخص میکنه ما کجای مین مموری قرار داریم

cpu همیشه یکسری Logical ادرس می بینه که بعد با استفاده از یک سخت افزاری که در کنار cpu قرار داره به اسم MMU اون ادرس رو می تونه تبدیل بکنه به یک ادرس فیزیکی که ادرس واقعی مین مموری است

اینکه گفتیم توی Execution time تازه ادرس واقعی مشخص میشه همین جاست که از طریق MMU دقیقاً ادرس واقعی مشخص میشه

پس cpu هم ادرس هایی که می بینه که مربوط به یک پروسس است ادرس های Logical است ینی ادرس هایی که بعد از جنریت شدن اون فایل باینری یا لود شدنش وجود داره ادرس Logical یا ادرس virtual: که cpu اون هارو تولید میکنه یا Cpu اونها رو می بینه و از طریق اونها به حافظه دسترسی داره

ادرس فیزیکی: ادرس واقعی unit های مموری است

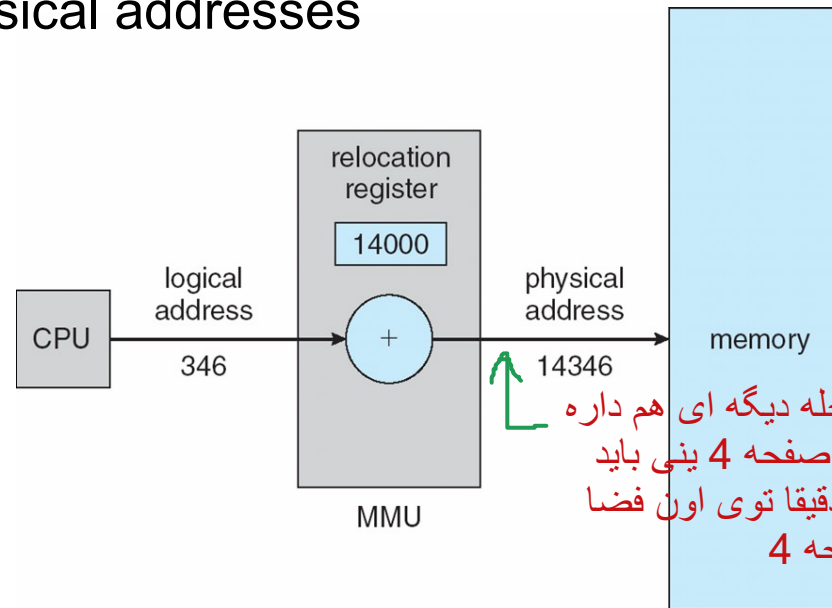
به کل اون ادرس های logical که یک برنامه مشخص تولید میکنه Logical address space اون برنامه می گیم

اون مجموعه ادرس های مین مموری که مربوط به ادرس های logical است رو Physical address space می گیم



Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
 - Usually a part of CPU
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
 - **Base register now called relocation register**
- The user program deals with *logical* addresses; it never sees the *real* physical addresses



اینجا یک مرحله دیگه ای هم داره
که میشه دقیقاً صفحه 4 بنی باید
چک بشه که دقیقاً توی اون فضا
باشه مثل صفحه 4



MMU توی زمان اجرا می تونه ادرس virtual رو به فیزیکیال ادرس مپ کنه

این MMU یک قسمتی از cpu است

اون رجیستر base که گفتیم اینجا استفاده میشه

اینجا باید ادرس های virtual باید با اون رجیستر base جمع بشه که ادرس فیزیکیال رو بسازه :

این برای حالتی که است که فضای ادرس پروسس رو پشت سر هم توی مموری فرض کرده ایم

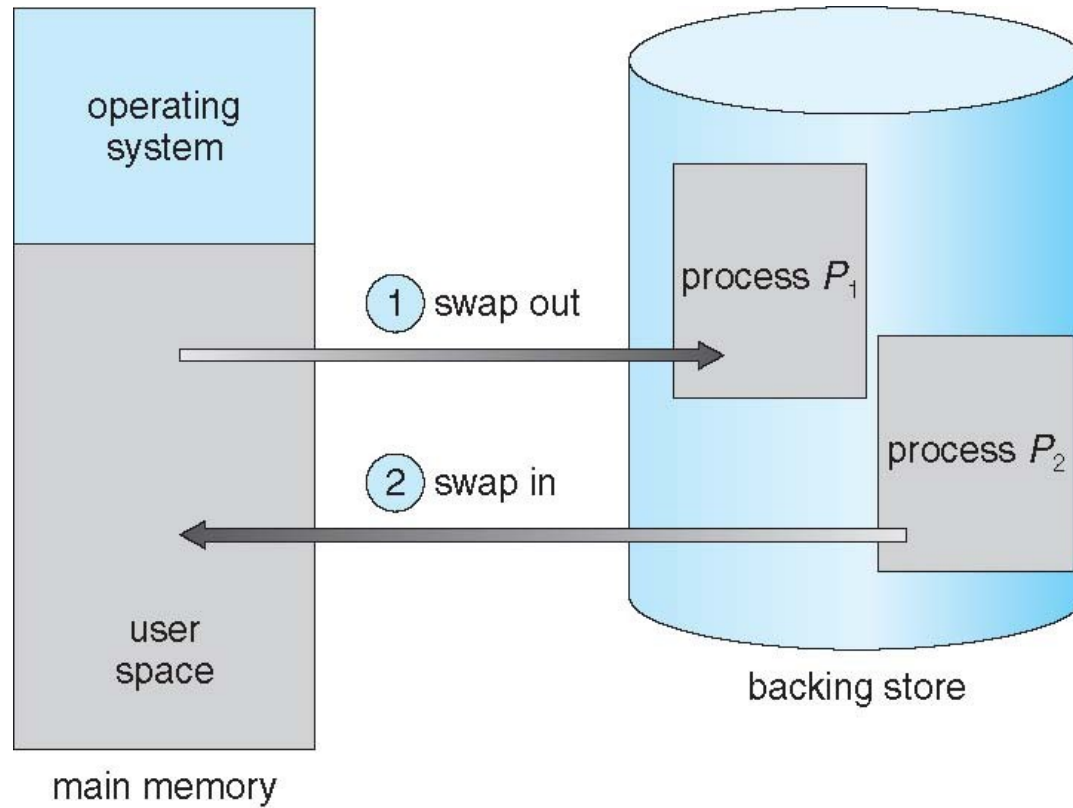
پس برنامه های اپلیکیشن ادرس های virtual رو می بینن و اصلا ادرس های واقعی مموری رو

هیچ وقت نمی بینن و این MMU است که ادرس های virtual رو تبدیل میکنه به ادرس های

واقعی



Schematic View of Swapping



Swapping: ینی یک قسمتی از دیسک سخت ما به نوعی داره جبران محدودیت حافظه اصلی رو میکنه و برای بالا بردن درجه مالتی تسکینگ سیستم ما این کارو میکنیم و اگر مین مموری ما کم اومد ممکنه یک برنامه ای یا قسمت هایی از یک برنامه رو ببریمش توی دیسک سخت که اون لحظه نیازش نداریم و بعد دوباره توی زمانی که دقیقا می خوایم اون برنامه رو اجرا بکنیم از دیسک سخت بیاریمش توی مین مموری قرارش بدیم

اگر توی زمان loading اومده باشیم ادرس ها رو مشخص کرده باشیم اینجا اگر چیزی رو swap out اش کردیم توی دیسک بعدا که میخوایم همین پروسس رو swap in کنیم توی مموری باید دقیقا توی جای قبلی قرارش بدیم و اینو ما نمی تونیم تضمین کنیم



Dynamic Loading

- Until now we assumed that the entire program and data has to be in main memory to execute
- **Dynamic loading** allows a routine (module) to be loaded into memory only when it is called (used)
- Results in better memory-space utilization; an unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases (e.g., exception handling)
- No special support from the operating system is required
 - It is the responsibility of the users to design their programs to take advantage of such a method
 - OS can help by providing libraries to implement dynamic loading

```
Void * hndl dlopen("libname.so", RTLD_NOW);  
Void * lib_func = dlsym(hndl, "func_name");
```



-
دو مزیت دیگر:

Dynamic Loading: با استفاده از **Dynamic Loading** ما می‌تونیم بعضی از قسمت‌های یک برنامه رو موقع اجرا به تناسب این که ایا نیاز پیدا میکنیم که ازش استفاده بکنیم یا نه لودش بکنیم. توی مموری ینی اینکه ما یک برنامه کامل نوشتیم یک راه اینه که وقتی که میخوایم این برنامه رو اجرا بکنیم کل این فانکشن‌ها و کلاس‌های مختلف رو لود بکنیم. توی حافظه ینی بیاریمشون توی مموری اصلیمون و بعد از اون شروع بکنیم به اجرا کردن برنامه ولی خب وابسته به اینکه برنامه چجوری اجرا بشه مثلا این برنامه یه عالمه کلاس، فانکشن و ایناها داره و یک مین هم داره که نقطه آغازین اجرای برنامه از اون مین است.

حالا وابسته به زمان اجرا بعضی از این فانکشن‌ها می‌تونن فراخوانی بشن و بعضی‌ها فراخوانی بشن یا از بعضی از این کلاس‌ها ابجکت ساخته بشه یا نشه پس ما اگه همه این‌ها رو از اون ابتدا براش فضا توی رم در نظر بگیریم بهینه نیست.

یک امکانی که برای ما قرار دادن **Dynamic Loading** ینی ما یک قسمت از کدهایی که ممکنه نیاز پیدا بکنیم رو از ابتدا لود نکنیم بلکه حین نوشتن برنامه هرجایی که می‌دونیم توی اون قسمت برنامه بهش نیاز پیدا میشه به اجراشون اون‌ها رو لود بکنیم و ازشون استفاده بکنیم.

یکی از روش‌های این کار استفاده از کتابخونه‌های داینامیکی است. مثلا توی لینوکس کتابخونه‌هایی که با پسوند **so** داریم ینی اون قسمت‌هایی از کد رو که میخوایم در زمان اجرا لودشون بکنیم به صورت کتابخون‌های **so** می‌نویسیم و بعد موقعی که کد رو می‌نویسیم هرجا نیاز به کتابخونه مورد نظر باشه اونو **open** میکنیم.

خوبیش چیه؟ مثلا توی بحث **exception handling** این می‌تونه کاربرد داشته باشه --> اگر برنامه توی زمان اجرا به اون **exception**‌ها نخورد اون کد‌ها اصلا لود نمیشن توی حافظه --> اینجوری استفاده از حافظه بهینه است.

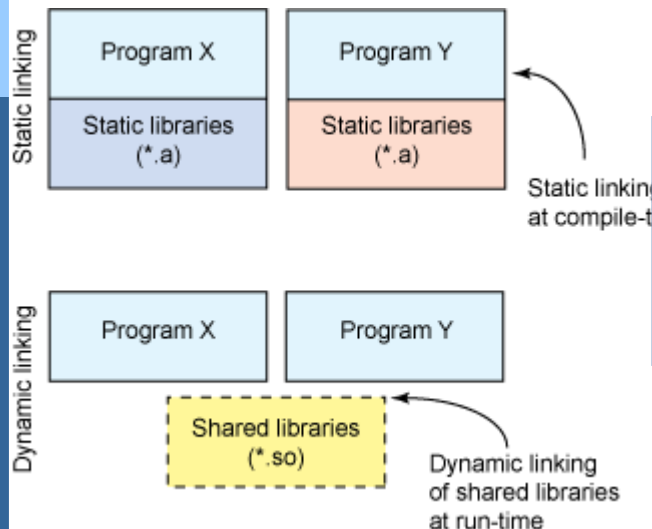
پس از ابتدا برای همه قسمت‌هایی که احتمال داره توی برنامه توی زمان اجرا بیاد اجرا بشه ما حافظه در نظر نمی‌گیریم.

امکان **Dynamic Loading** ربطی هم به سیستم عامل به طور خاص پیدا نمیکنه. برنامه نویس توی کدش مشخص میکنه که کجاها نیازه که ما مثلا فلان کلاس مشخص رو لود کنم یا فلان کتابخونه مشخص رو لود کنم و فانکشن‌هاش استفاده کنم.



Dynamic Linking

- **Dynamically linked libraries (DLL)** – system libraries that are linked to user programs when the programs are run.
 - Similar to dynamic loading. But, linking rather than loading is postponed until execution time
- Operating system checks if routine is in processes' memory address
 - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**



```
include library headers in your code.  
In compile time, introduce the dynamic library  
g++ sampleCode.c -ldynamicLibraryName
```



-
Dynamic Linking: وقتی که یک برنامه ای رو می نویسیم یکسری کلاس این ها داریم و این ها رو در کنار هم همه رو با هم دیگه کامپایل میکنیم و لینکشون می کنیم بهم و یک فایل باینری نهایی به دست میاد که اون فایل باینری رو اجرا میکنیم

معمولا باز هم اگر از کتابخونه ها استفاده کرده باشیم ینی خودمون کتابخونه هایی نوشته باشیم از کتابخونه های استاتیک استفاده میکنیم که مثلا توی لینوکس با پسوند **a** هستن--> مثل توی شکل:
مثلا یک برنامه ای می تونه یک استاتیک لایبرری بهش لینک شده باشه ینی موقعی که ما کامپایل می کنیم برنامه رو و بعد میخوایم همه ابجکت فایل های مورد نیاز رو کنار هم بذاریم و یک فایل باینری نهایی تولید کنیم این لایبرری ها هم بهشون میدیم و این لایبرری ها هم باینریشون لینک میشه به باینری ما ینی از توابع این لایبرری ها توی برنامه **x** استفاده میکنیم ولی کدش توی برنامه **x** نیست و موقعی که لینک میشن بهم می چسبن و موقع اجرا در کنار هم هستن

ولی اون چیزی که توی **Dynamic Linking** داریم شکل پابینیش است: توی **Dynamic Linking** امکان این هست که ما اصلا موقعی که باینری برنامه رو می سازیم اون لایبرری بهش لینک نشده باشه ینی مستقلا برنامه **x** رو داریم و برنامه **y** رو هم داریم یک لایبرری هم داریم که توی حافظه قرار داره حالا موقع زمان اجرا این برنامه **x** می تونه از توابع یا کلاس های این لایبرری استفاده بکنه --> اینجا این امکان برای ما فراهم میشه که به صورت شیرد هم بتونیم از لایبرری ها استفاده بکنیم ینی مثلا برنامه **x** و برنامه **y** هر دوتاشون دارن از یک لایبرری استفاده می کنن در حالی که توی استاتیک لایبرری مجبور بودیم که هر استاتیک لایبرری رو به برنامه خودمون به صورت جدا لینکش بکنیم

توی **Dynamic Linking** امکان این هست که یکبار این لایبرری رو توی حافظه لودش بکنیم و بعد به صورت شیرد برنامه ها دارن از اون لایبرری واحد استفاده می کنن

توی این حالت که داریم از **Dynamic Linking** استفاده میکنیم که به لایبرری هاش **DLL** گفته میشه <-- توی ویندوز با پسوند **DLL** است و توی لینوکس با پسوند **so** است اینجا کافیه که ما یک برنامه ای بنویسیم و هدر فایل های مربوط به این شیرد لایبرری رو اینکلود بکنیم توی کدمون و از فانکشن ها و کلاس هاش استفاده بکنیم و بعد توی زمان کامپایل با **-l** بهش میگیم که برنامه مون داره از فلان لایبرری استفاده میکنه <-- اینجا الان لینک نمیشه این لایبرری به **sampleCode** که اینجا داریم بلکه فقط به این **sampleCode** معرفی شده که این بعدا توی زمان اجرا ممکنه از این **dynamicLibraryName** استفاده بکنه اینجا با **-L** می تونیم ادرس این شیرد لایبرری توی سیستم رو هم بدیم ک اگر توی زمان اجرا بهش نیاز داشت ادرسشو داشته باشه و اگر ادرس ندادیم یی توی ادرس های مشخصی اون **so** قرار داره که سیستم می شناسه پس اینجوری بهش معرفی میکنیم و توی زمان اجرا این کد می تونه اون لایبرری رو استفاده بکنه حالا یا اون لایبرری لود شده از قبل مثلاً به برنامه ای قبلش اینو لود کرده یا اینکه لود نشده پس اول لود میشه و بعد ازش استفاده میکنه

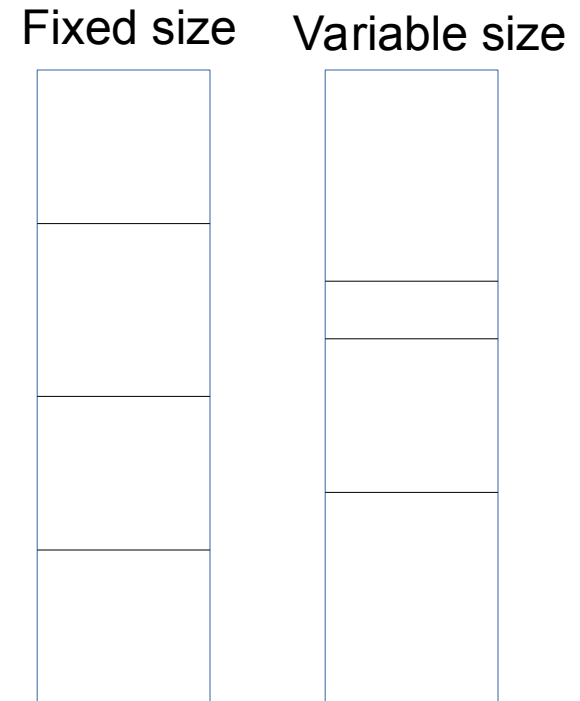
این حالت با حالت **Dynamic Loading** فرق داره توی **Dynamic Loading** ما توی کد مشخص کردیم که کجا یک لایبرری رو لودش بکن ولی اینجا فرضمون این است که لایبرری توی زمان اجرا توی سیستم لود شده یا اگر لود نشده اول لودش می کنیم و بعد ازش استفاده میکنیم این **Dynamic Loading** باز بیشتر وابسته به نیاز ما قابل اجرا شدن است یا قابل لود شدن است

خوبی حالت **Dynamic Linking** این است که به صورت شیرد می تونه چندتا برنامه از اون لایبرری واحدی که توی حافظه است استفاده بکنه



Memory allocation for process

- Contiguous Allocation
 - Fixed size partitions
 - ▶ Causes **internal fragmentation**
 - Variable size partitions
 - ▶ Causes **external fragmentation**
- Segmentation
 - Variable size segments
- Paging
 - Fixed size pages



روشی که بر مبنای Variable size وجود داره Segmentation است و
روشی که بر مبنای Fixed size وجود داره Paging است



برای یک پروسس مشخص به چه صورت مموری توی حافظه می تونیم اختصاص بدیم:
 یک راه ساده اینه که پروسس به صورت پشت سر هم ینی اون فضای ادرسی که نیاز داره این فضا رو به صورت پشت سر هم توی یک قسمتی از حافظه در نظر بگیریم --> در ادامه می بینیم اینطوری نیست

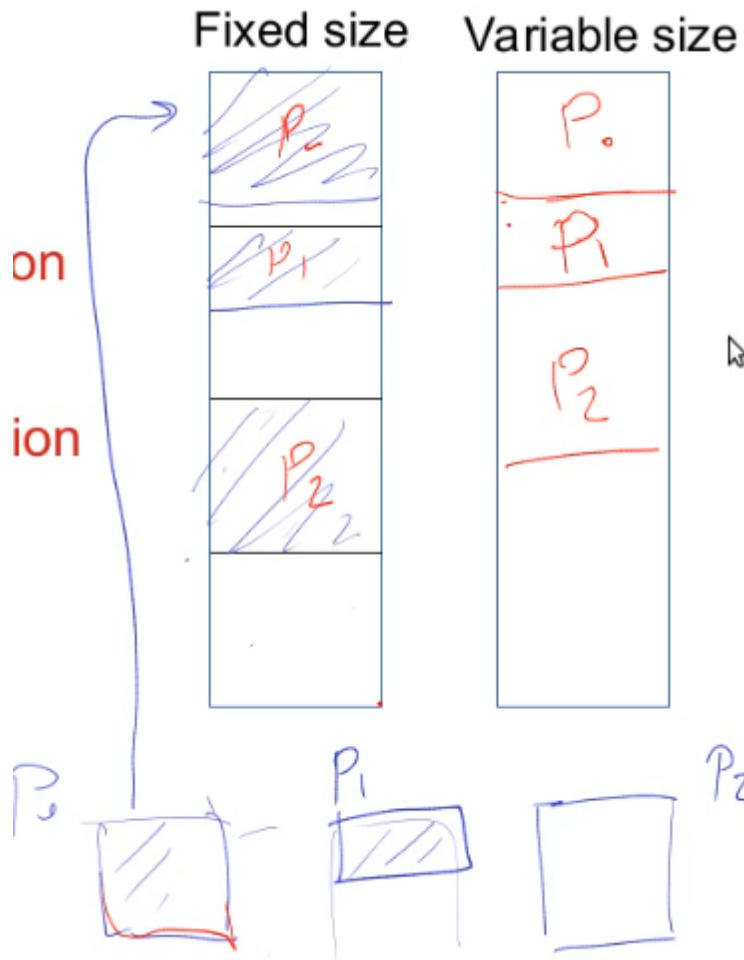
فرض می کنیم یک پروسس یا یک بلاکی از پروسس رو می خوایم توی حافظه قرار بدیم به صورت پشت سر هم دو حالت میتونه وجود داشته باشه:

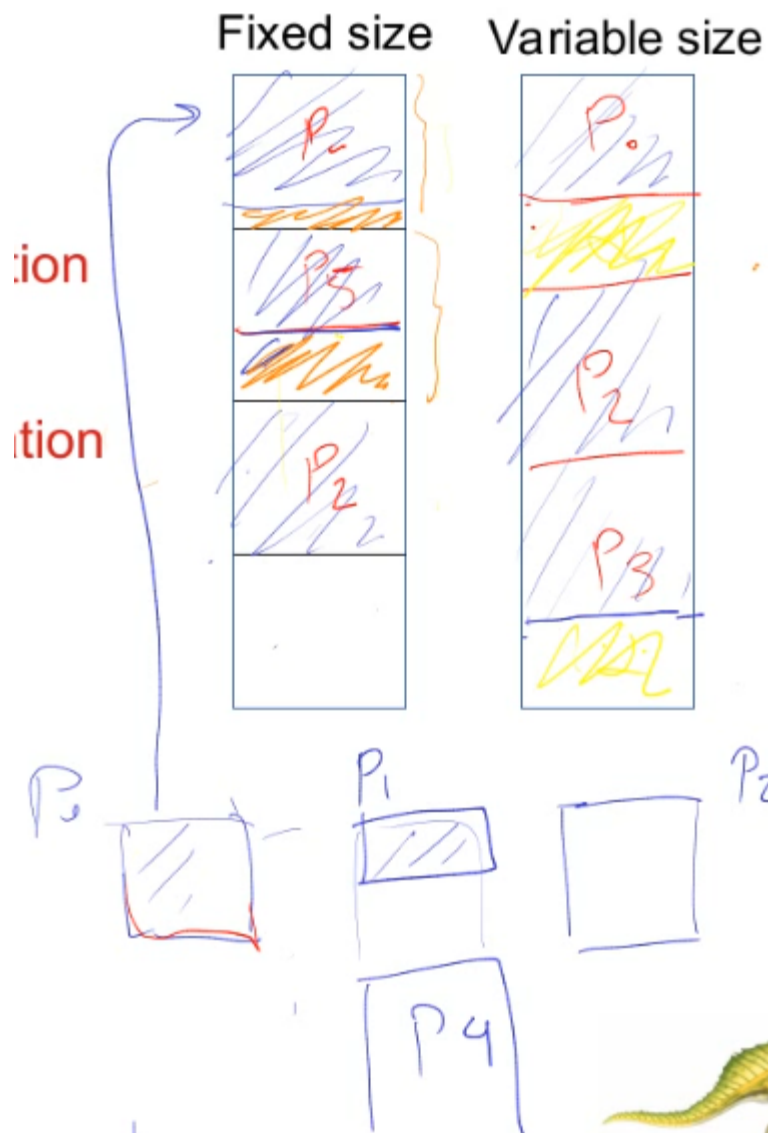
- 1- حافظه رو به قسمت های مساوی تقسیم بکنیم ینی **fixed size** و بعد وقتی که می خوایم در واقع پروسس ها رو توی حافظه **allocate** کنیم بیایم هر پروسسی رو توی یکی از این قسمت های حافظه قرار بدیم ینی هر بلاک رو برای یک پروسس در نظر می گیریم (فرضمون است)--> عکس
 - 2- چیزی رو ثابت در نظر نگیریم اصلا نیایم به تقسیم بندی به این صورت توی حافظه داشته باشیم بلکه پروسس ها رو با همون اندازه هایی که دارن بذاریمشون توی حافظه --> اینجا از قبل تقسیم بندی ثابتی توی حافظه نکردیم --> اجرای بعضی از این پروسس ها می تونه تموم بشه و وقتی که اجراشون تموم شد از حافظه درشون میاریم مثلا اجرای **p1** تموم شده پس از حافظه درش میاریم اتفاقی که می افته اینه که قسمت بین **p0** و **p2** الان خالیه --> در این حالت **p4** نمی تونه توی این حافظه باشه با وجود اینکه فضای خالی داریم --> پس این نوع مدیریت حافظه باعث شد که حافظه تیکه تیکه بشه و ما نتونیم از این تیکه های کوچیک استفاده بکنیم
- توی حالت سائز ثابت ما اگر **p1** رو حذف می کردیم یکی از بلاک های حافظه که سائزش مشخص بود ازاد میشد

مشکل **Fixed size** این است که ما یک قسمت هایی داریم که از درون این بلاک های ثابت این قسمت ها هدر رفته مثل قسمت پایین **p0**

پس توی هر دوی این حالت ها یکسری هدر رفتن های حافظه داریم:

- 1- توی حالت اول: تیکه های کوچیک کوچیک داخل بلاک ها ایجاد میشه که ما ازشون نمیتونیم استفاده بکنیم رو می گیم **internal fragmentation**
- 2- توی حالت دوم: بهش می گیم **external fragmentation** چون کاملاً خارج از پروسس ها هستن و اگر این قسمت های زرد برامون یه اندازه مناسبی بود می تونستیم ازشون استفاده بکنیم







Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used



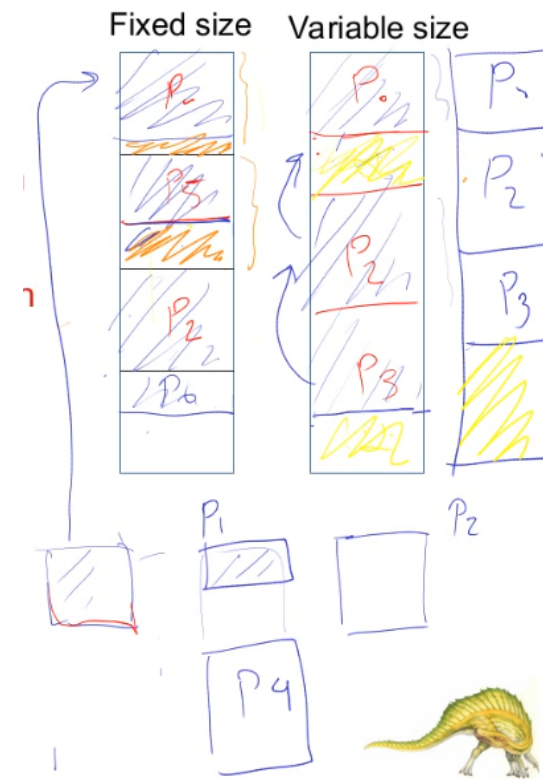
External Fragmentation: ینی اون مموری اسپیس مون ممکنه که اون قسمت ازادی که وجود داره برای ما کافی باشه که یک پروسس رو بیاریم توی حافظه ولی اون ها توی یک قسمت دنبال هم نیستن

Internal Fragmentation: ینی اون قسمتی که ازاده مربوط به یک بلاکی که یک قسمتیش توسط یک پروسس دیگه گرفته شده بنابراین ما پروسس دیگه ای رو غیر از اون نمی تونیم توش جا بدیم



Solution to Fragmentation problem

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time



مسئله ای که در مورد Fragmentation پیش میاد:

یکی از روش ها اینه که ما compaction کنیم این قسمت های fragmentation که فقط توی روش external fragmentation این امکان پذیره ینی بیایم اون قسمت های زرد رو جابه جا بکنیم و کنار هم قرار بدیم که یک فضای خالی به وجود بیاد و ازش بتونیم استفاده بکنیم ولی برای روش internal fragmentation نمی تونیم این کارو بکنیم چون اون قسمت هایی که ازاده داخل بلوک هاست و ما اینارو نمی تونیم جابه جاشون کنیم

توجه: یک همچین کاری هم در صورتی امکان پذیره که ما حین اجرا بتونیم پروسس ها رو جابه جا بکنیم ینی ادرس ها رو توی زمان اجرا bind کنیم --> حالت execution time



Segmentation

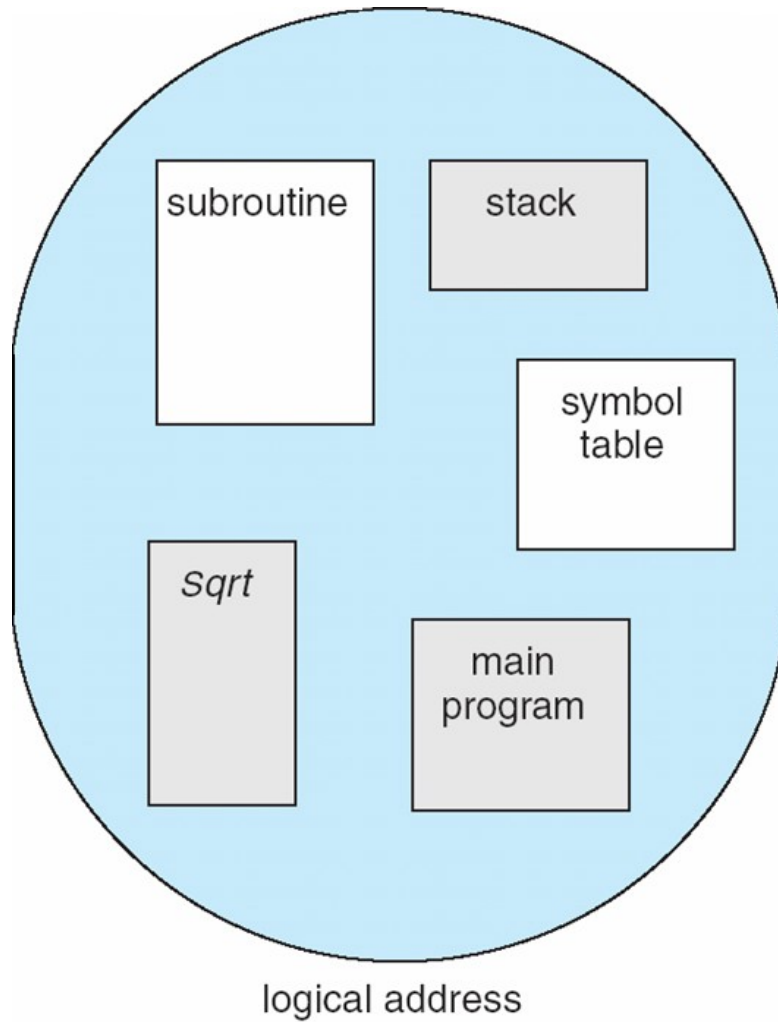
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
 - A segment is a logical unit such as:
 - main program
 - procedure
 - function
 - method
 - object
 - local variables, global variables
 - common block
 - stack
 - symbol table
 - arrays



روش اول مدیریت حافظه یا روش Segmentation: این روش بر مبنای ایده پارتیشن بندی حافظه با اندازه های متفاوت است
این روش کجاها میتونه استفاده بشه؟
وقتی که ما به صورت منطقی بتونیم پروسسمون رو از قسمت های مختلفی تصور کنیم یه بشکنیمش به قسمت های مختلفی --> مثل صفحه بعدی
پس ما می تونیم یک برنامه مشخص رو به Segment های مختلفی که از لحاظ منطقی تعریف میشن تقسیم بکنیم



User's View of a Program

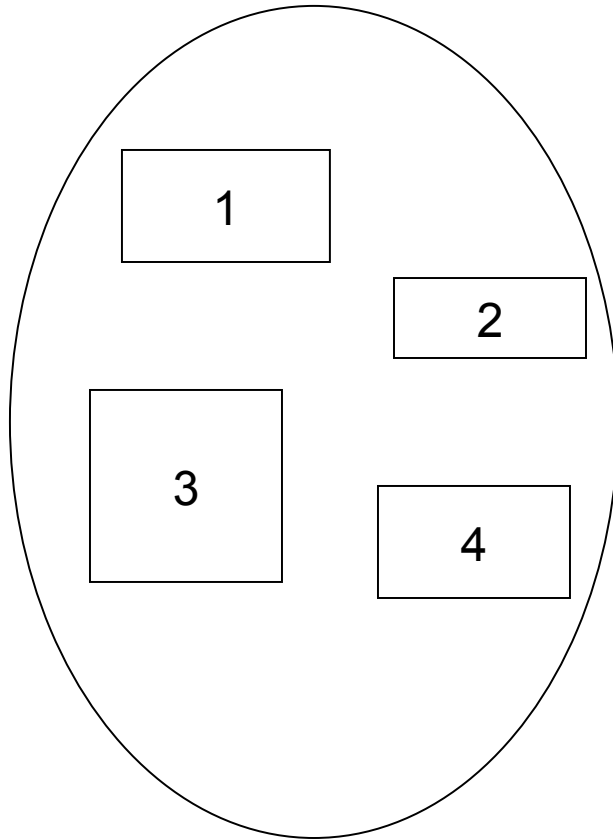


این ابیه فضای ادرس پروسس ما است
یک قسمتش رو برای استک در نظر گرفتیم
یک قسمتش رو برای یک فانکشن خاص در نظر گرفتیم مثل `sqrt`
یک قسمتش `symbol table` است
یک قسمتش برنامه مین است
یک قسمتش ساب روتین است

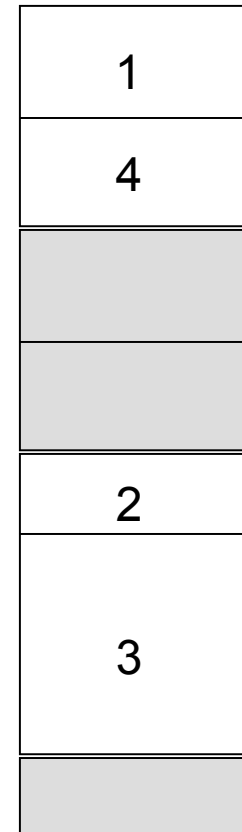
ما می‌تونیم هر کدوم از این قسمت‌ها رو که از لحاظ مفهومی و کاربردی معنای مختلفی دارن رو
توی یک `Segment` تصور کنیم



Logical View of Segmentation



user space



physical memory space



مثال:

Segment های مختلف رو می تونیم توی جاهای مختلف فیزیکیال مموری قرار بدیم

سوال:

توی زمان اجرا از کجا بفهمیم که دستور و دیتایی که توی Segment 3 است دقیقا کجای حافظه قرار داره؟

یه جایی باید ذخیره بکنیم که هر کدوم از این Segment ها توی کدوم یکی از این قسمت های حافظه اصلی قرار گرفته



Segmentation Architecture

- Logical address consists of a two tuple:
 <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
 - **base** – contains the starting physical address where the segments reside in memory
 - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
 segment number **s** is legal if **s** < **STLR**



می‌تونیم تصور بکنیم که توی روش Segmentation ادرس logical ما از دو قسمت تشکیل شده: segment-number, offset

offset: مشخص میکنه که اون قسمتی که بهش می‌خوایم اشاره بکنیم توی بایت چندم اون سگمنت مشخص شده قرار داره

یک Segment table باید یه جایی داشته باشیم که به ما بگه که هر کدوم از این سگمنت‌های مربوط به پروسس ما کدوم قسمت حافظه واقعی قرار گرفته

پس Segment table میاد یک مپی رو انجام میده ینی ادرس logical رو به ادرس فیزیکی مپ میکنه

Segment table باید دوتا اطلاعات داشته باشه:

base: مشخص میکنه سگمنت مورد نظر ما ابتداش توی حافظه واقعی از چه ادرسی است

limit: مشخص میکنه که این سگمنت اندازه اش چقدره

پس توی Segment table به ازای هر سگمنتی باید دوتا اطلاعات رو ذخیره بکنیم: base , limit

به ازای هر پروسسی باید یک Segment table داشته باشیم --> پس برای هر برنامه‌ای باید

یک Segment table جدا داشته باشیم که ادرس سگمنت‌های اون برنامه رو بهمون بده --> خود این Segment table ها هم باید یه جایی توی حافظه ذخیره بکنیم:

STBR: مشخص میکنه که اون لوکیشن Segment table ها توی مموری کجا هست

STLR: مشخص میکنه که این پروسس مشخص چندتا سگمنت داره ینی مشخص میکنه که

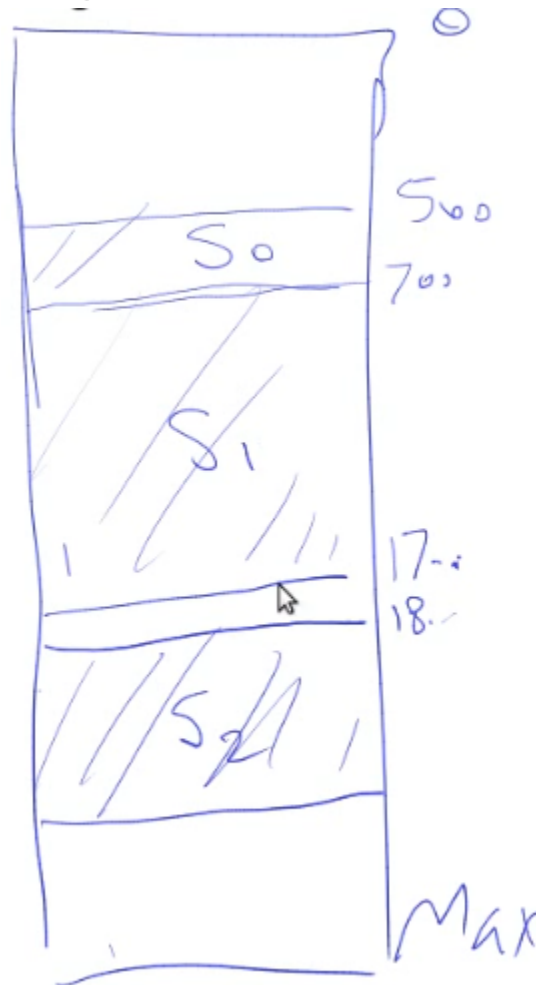
Segment table اش چندتا سطر داره



Segmentation: Example

- What is the physical addresses of below logical addresses according to the segment table?

- 2,700
- 0,150



Base	Length
500	200
700	1000
1800	600



مثال:

یک همچین Segment table داریم برای یک پروسیسی
مثلا میگه سگمنت صفرم توی ادرس 500 حافظه قرار میگیره و اندازه اش هم 200 است
فرض میکنیم این ادرس های logical رو cpu داره می بینه و این ادرس های logical وقتی که
ترجمه میشن به ادرس فیزیکی واقعی چه ادرسی به ما میدن؟



Segmentation: Example

- What is the physical addresses of below logical addresses according to the segment table?

- 2,700

- 0,150

- 2,700: Invalid

- 0,150: $500+150=650$

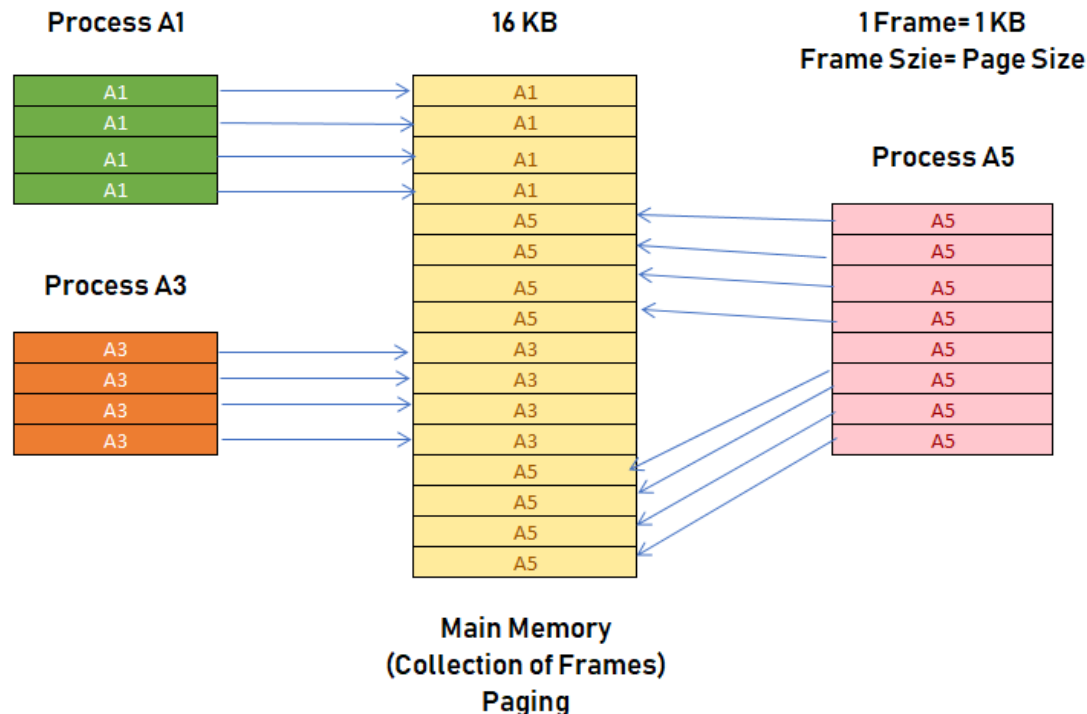
Base	Length
500	200
700	1000
1800	600





Paging

- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**



کنیم

Paging: توی این روش می خوایم از ایده تقسیم حافظه به قسمت هایی با اندازه های یکسان استفاده

این مین مموری به اندازه های یکسانی تقسیم شده که به هر کدوم از این ها یینی به هر کدوم یکی از این A ها یک **frame** گفته میشه توی مین مموری یا مموری فیزیکی ما

از اون طرف هر کدوم از پروسس ها رو هم به همون اندازه های ثابت تقسیم کردیم و هر کدوم از این اندازه های ثابت توی پروسس ها بهش **page** میگی

پس اندازه **page** ها با اندازه **frame** ها یکسان است

توی روش **Paging** می خوایم هر کدوم از **page** های یک پروسس توی یکی از **frame** های

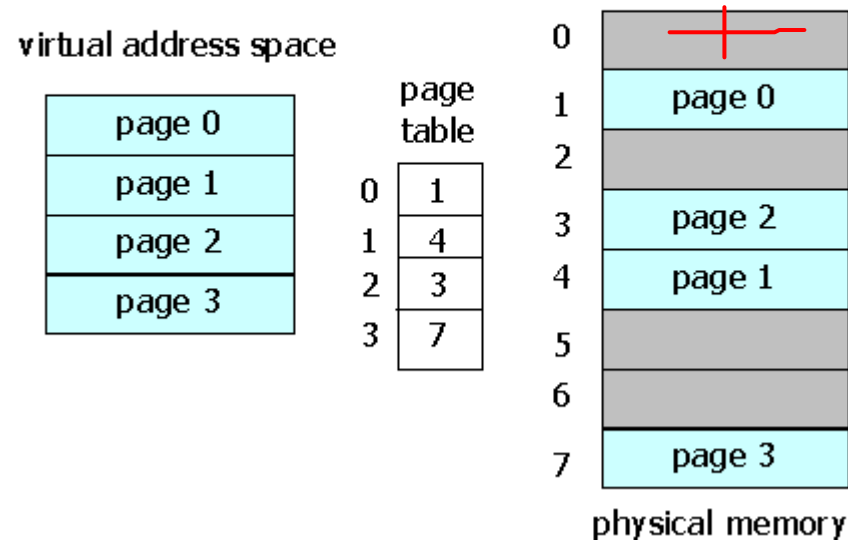
حافظه فیزیکالمون قرارش بدیم --> مثل شکل

نکته: همه **page** یک پروسس ممکنه پشت سر هم توی حافظه قرار نگیره مثل پروسس A5



Paging

- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- **Still have Internal fragmentation**



از طرف دیگر ما باید توی فیزیکال مموری به نوعی بتونیم frame هایی که الان آزاد هستن رو داشته باشیم تا بعد که خواستیم page یک پروسس رو براش حافظه واقعی اختصاص بدیم بتونیم از page هایی که آزاد هستن استفاده بکنیم

+ --< این frame ها الان ازادن ینی رنگ های طوسی

برای هر پروسس خاصی ما باید یک page table داشته باشیم که بهمون بگه هر page این پروسس به کدوم یک از frame های حافظه مپ شده

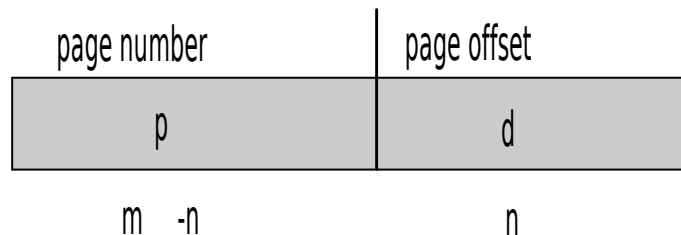
اینجا مشکل External Fragmentation رو نداریم چون اون قسمت هایی که ازادن دقیقا برابر یک page هستن هر کدومشون که ما کاملا یک page رو میتونیم توی هر کدوم از این frame ها قرار بدیم

فقط ممکنه یک page کامل پر نباشه توی فضای ادرس اون پروسس هم و مثلا یک قسمتیش داده وجود داشته باشه و ما کاری به این نداریم و کل یک page رو میایم میذاریم توی یکی از frame های حافظه فیزیکالمون پس همچنان ممکنه این frame ها Internal fragmentation داشته باشن --< که ما نمیتونیم اینو کاریش بکنیم توی این روش



Address Translation Scheme

- Address generated by CPU is divided into:
 - **Page number** (p) – used as an index into a **page table** which contains base address of each page in physical memory
 - **Page offset** (d) – combined with base address to define the physical memory address that is sent to the memory unit



- For given logical address space 2^m and page size 2^n



ما باید یک ادرس ویرچوال داشته باشیم که cpu اون رو می بیند و بعد cpu این ادرس ویرچوال رو می‌ده به MMU و MMU تبدیلش می‌کنه به ادرس واقعی که ادرس واقعی فیزیکیال ما است این ادرس ویرچوالی که cpu می بیند از بخش های زیر تشکیل شده:

page offset: ینی مشخص می‌کنه که اون ادرس مورد نظر کجای این page توی ادرس فیزیکیال قرار می‌گیره پس d این رو مشخص می‌کنه

page number: این مشخص می‌کنه که شماره page ما چنده

پس به طور کلی اگر ادرس ویرچوال اندازش m بیت باشه و n بیت رو برای offset پیچ در نظر گرفته باشیم m-n بیت برای page number است پس اندازه page های ما می‌تونه 2 به توان

d باشه پس هر page مون می‌تونه 2 به توان d بایت داشته باشه که بستگی به این داره که ما ادرس دهمون روی چه قسمتی قرار میدیم مثلا بایت به بایت رو ادرس میدیم یا مثلا word به

word --> این بستگی به سیستم داره ولی به طور کلی ما بایت در نظر می‌گیریم ینی فرض می‌کنیم هر بایت حافظه ادرس دهی میشه پس ما توی هر page می‌تونیم 2 به توان d تا بایت داشته باشیم

و فضای ادرس ویرچوالمون حداکثر چندتا page میتونه داشته باشه؟ به تعداد page number

هایی که می‌تونیم اینجا با این تعداد بیت نشون بدیم ینی 2 به توان m-n که تعداد page هامون میشه پس یک همچین سیستمی ینی اگر سیستم ادرس دهی ما ادرس ویرچوالش m بیتی باشه که n بیتش

رو هم برای page offset در نظر گرفته باشیم 2 به توان m-n تا page توی این ویرچوال مموری اسپیس می‌تونیم در نظر بگیریم ینی هر پروسسی می‌تونه به این تعداد page داشته باشه

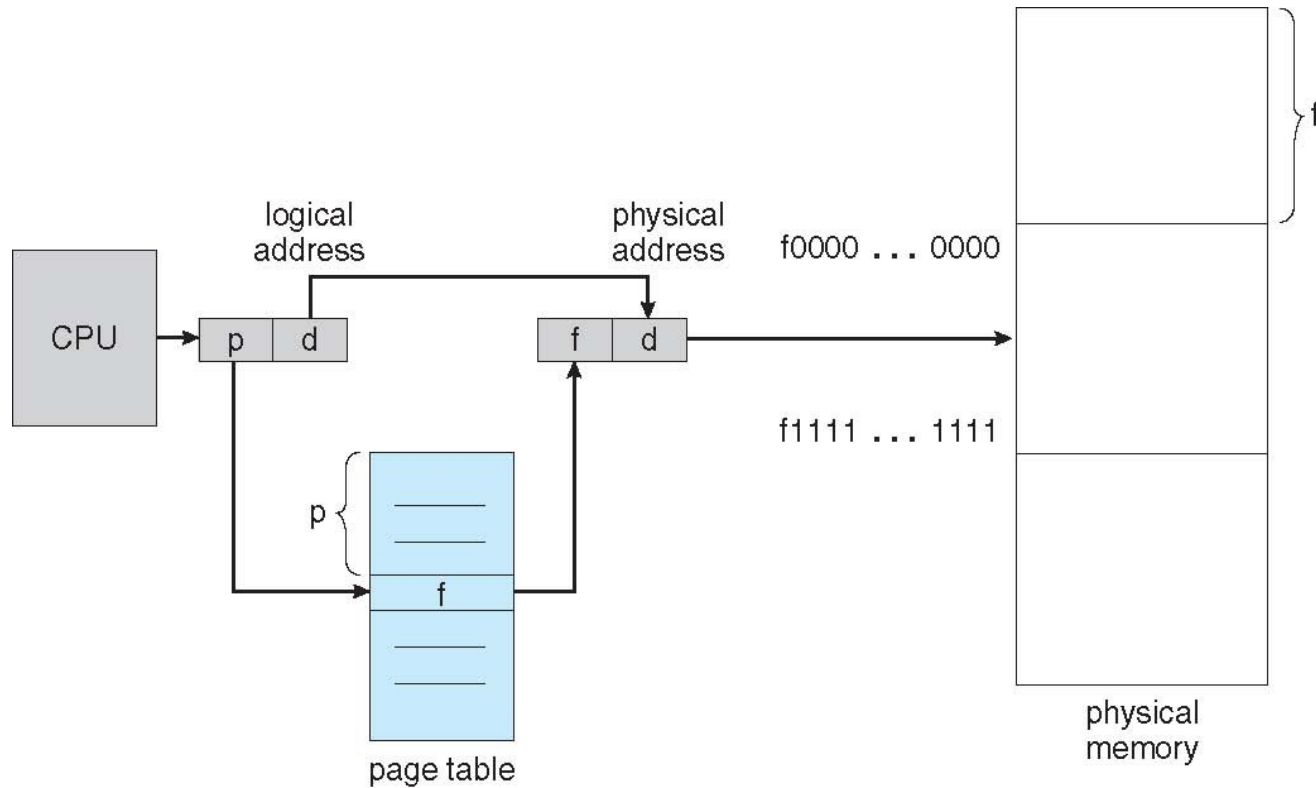
حداکثر

نکته: این ویرچوال اسپییسی که برای هر پروسس در نظر می‌گیریم این ویرچوال اسپیس ممکنه

بعضی از page هاش خالی باشه



Paging Hardware

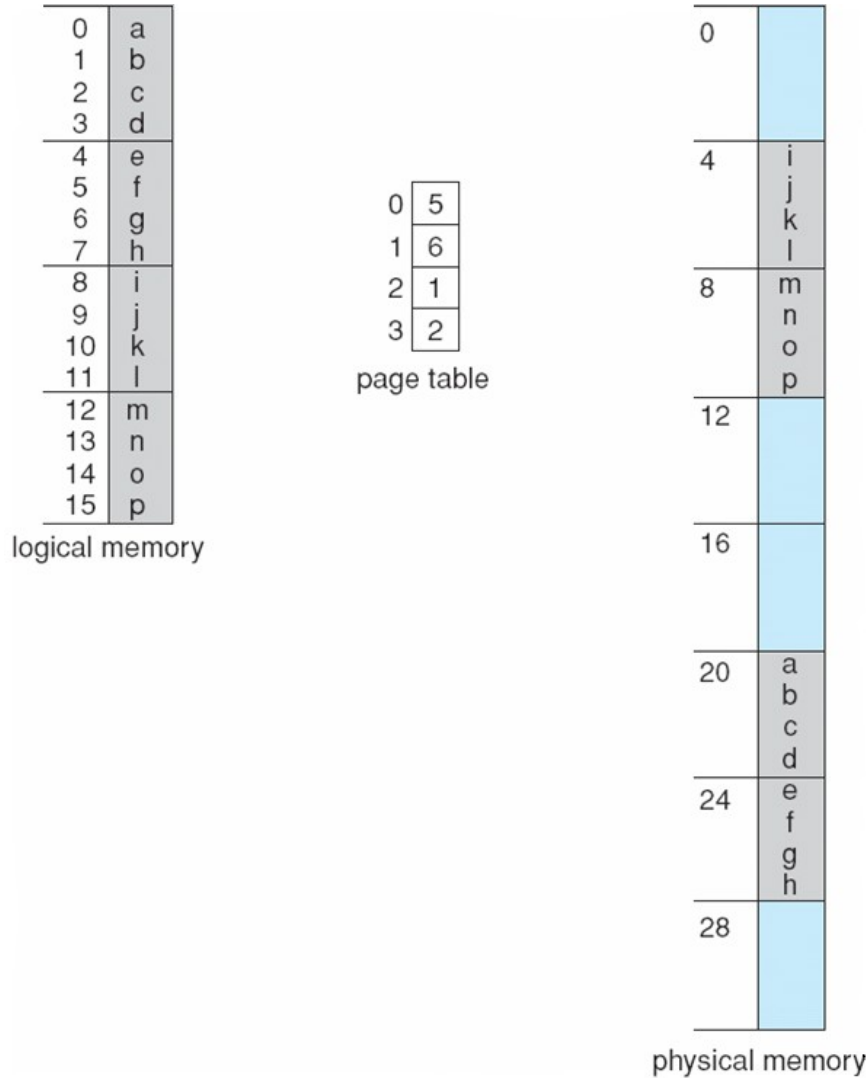


مثال:

cpu این ادرس ویرچوال رو می بینه ینی p , d
و بعد این page number رو باید بره توی page table پیدا کنه ینی به اندازه p تا باید بریم
جلو توی page table و این عددی که اینجا توی page table ذخیره شده ینی f داره شماره
 frame رو توی حافظه فیزیکی رو به ما میده پس این f رو استخراج میکنیم
الان ادرس فیزیکی قسمت اولش میشه همون شماره frame ینی f و offset رو هم از همون جای
قبلای برمی داریم و تغییری نکرده ینی d
این ادرس فیزیکی داده میشه به مموری فیزیکال و از طریق این ادرس فیزیکی می تونیم به اون
قسمت که مدنظرمون بوده دسترسی پیدا کنیم



Paging Example



$n=2$ and $m=4$ 32-byte memory and 4-byte pages



مثال:

این ویرچوال مموری ما 4 تا page داره که هر page هم تعدادی بایت داره
خوبی page table: اندازه page ها یکسان است و دیگه نیاز نیست که یک مقداری هم برای
اندازه page ها نگهداری بکنیم --> کافیه page tableمون به ترتیب مثل یک ارایه باشه شماره
frame ها به ترتیب توش قرار گرفته



Paging: Example

- If the page size is equal to 1KB, in logical address 0000010111011110, how many pages are there in the logical memory space?



فرض میکنیم اگر توی یک سیستمی page size ما 1KB باشه و یک ادرس logical به صورت 0000010111011110 داشته باشیم حالا می‌خوایم ببینیم که با توجه به این ادرس logical و این page size می‌دونیم 1KB است چقدرتا page توی فضای مموری اسپیس logicalمون قرار داره؟



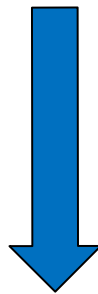
Paging: Example

- If the page size is equal to 1KB, in logical address 0000010111011110, how many bits are required for the page number?

- 1 KB = 10 bits for page offset

- 0000010111011110

16 bits



$16 - 10 = 6$ bits for the page #

So 2^6 pages in memory space



جواب:

هر کیلوبایتی ۲ بیتی ۱۰ تا ۱۰ بیت نیاز دارد برای نشون دادن page offset
از این ۱۶ بیت اگر ۱۰ بیت رو از سمت راست برداریم ۶ بیت دیگه می مونه پس ۲ به توان ۶ تا
page توی فضای ادرس logical ما می تونه وجود داشته باشه



Paging: Example

- There are 4 pages in a logical address space and each page has 2 words. If these pages are assigned to a physical address space with 8 frames, how many bits are required for physical and logical addresses?



-

مثال:

فرض میکنیم توی فضای ادرس logical مون 4 تا page داشته باشیم و هر page هم 2 تا word باشه $1\text{word} = 2\text{byte}$
اگر page ها رو به یک فضای ادرس فیزیکی با 8 تا frame مپ کنیم چندتا بیت لازمه برای اینکه ادرس فیزیکی و ادرس ویرچوال رو نشون بدیم؟



Paging: Example

- There are 4 pages in a logical address space and each page has 2 words. If these pages are assigned to a physical address space with 8 frames, how many bits are required for physical and logical addresses?
- Logical address: 4 pages = 2bits, 2 words=1 bit => 3 bits
- Physical address: 8 frames=3 bits, 2 words=1 bit => 4 bit



جواب:

نکته: اندازه فضای ادرس ویرچوال ما ربطی به اندازه فضا ادرس فیزیکیال ما می تونه نداشته باشه
و ما می تونیم فضای ادرس فیزیکیال بزرگتری داشته باشیم
از اون طرف می تونیم حتی فضای ادرس ویرچوالی به اندازه فضای ادرس فیزیکیالمون داشته باشیم



Paging (Cont.)

■ Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of $2,048 - 1,086 = 962$ bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = $1 / 2$ frame size

getconf PAGESIZE

■ So small frame sizes desirable?

- But each page table entry takes memory to track
- Page sizes growing over time
 - Windows 10 supports page sizes of 4 KB and 2 MB .
 - Linux also supports two page sizes: a default page size (typically 4 KB) and an architecture-dependent larger page size called huge pages



تحلیل روش paging:

می‌خوایم ببینیم که internal fragmentation داریم؟

حالا اگر بخوایم ببینیم چقدرتا page لازمه برای اینکه این پروسس رو قرار بدیم توی حافظه کافیه

که process size تقسیم بر page size بکنیم که میشه 35 تا page و یک تعداد بایت هم باقی می‌مونه پس ما لازمه که 36 تا page در نظر بگیریم که page اخر 1086 بایتش پره و بقیش خالیه پس مقداری که الان داره هدر می‌ره 962 بایت است --> اگر ما همینو در نظر بگیریم یکنی بهترین حالتو همین در نظر بگیریم اندازه Worst case fragmentation ما حالتی میشه که ما از ازون یک frame که داریم فقط 1 بایتش رو پر کرده باشیم و بقیش خالی باشه ---> اینجا بدترین حالت میشه از لحاظ هدر رفت حافظه به طور میانگین گفته میشه با این روش 1/2 اندازه frame ممکنه که از بین بره اگر دستور getconf PAGESIZE رو بزنین می‌تونیم ببینیم که الان سیستم با چه اندازه page کا می‌کنه مدیریتش

این بحث رو واسه چی کردیم؟ برای اینکه بتونیم تحلیل بکنیم که اگر خواستیم یک سیستم paging داشته باشیم اندازه frame ها یا اندازه page ها رو چند بگیریم بهتره؟ یکنی چه حالتی مناسب تر است؟ الان اندازه page باید بزرگ باشه یا کوچیک باشه؟

اگر اندازه page ها رو بزرگ بگیریم هدر رفت حافظمون زیاده پس تصمیم می‌کنیم کوچیکش بگیریم ولی اگر کوچیک هم بگیریم یکنی اندازه page table مون رو بزرگ تصور کردیم به عبارتی هر چی تعداد page های اون فضای ادرس ویرچوالمون بیشتر بشه سطرهای page table مون هم بیشتر میشه یکنی ما حافظه بیشتری برای ذخیره page table نیاز داریم و این page table هم باید یه جا ذخیره بشه پس اینجا مشکل پیدا میکنیم پس یه جورایی باید تعادل بین این دوتا رو رعایت بکنیم

با این حال در گذر زمان می‌بینیم که سیستم ها اندازه page size شون رو بزرگ کردن --> مثلا مموری ها بزرگ تر شدن و ما محدودیت حافظه کمتری نسبت به قبل داریم و سخت افزارهای ما هم بهبود پیدا کردن و مناسب تر شدن و توشون تدابیری شده که بتونیم page table ها هم توش مدیریت بکنیم حتی اگر اندازه page table هامون بزرگ بشه

اینجا معمولا توی سیستم ها دوتا اندازه page رو به صورت پیش فرض در نظر گرفتن برای سیستم که سیستم با یکیش کار میکنه اما امکان این که با اون بزرگتره هم کار بکنه وجود داره که اینجا ما می‌تونیم با توجه به اون اپلیکیشن هایی که روی سیستم اجرا میشه با توجه به اون اندازه page size رو مشخص بکنیم



Paging (Cont.)

- Process view and physical memory now very different
 - The programmer views memory as one single space, containing only this one program
 - By implementation process can only access its own memory
- The **frame table** has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).



این مدل فکر کردن برای مدیریت حافظه باعث میشه که دیدی که پروسس از ادرس ها داره متفاوت باشه با اون دیدی که توی ادرس های فیزیکی ما است ینی پروسس می تونه دقیقا خودش یک فضای ادرس از شماره صفر تا اون ماکزیمم اندازه ویرچوال ادرسی که می تونیم داشته باشیم برای خودش در نظر بگیره و اصلا کاری هم به این نداشته باشه که این ادرس ها واقعا کجای ادرس فیزیکی قرار دارن

پس برنامه نویس هم اگر بخواد یکسری بهینه سازی هایی در مورد دسترسی به حافظه داشته باشه می تونه با توجه به این دید که ادرس شماره page ها از صفر تا اون ماکزیمم هست و هر page به صورت جدا اختصاص داده میشه از این دید استفاده بکنه و کلا اون برنامه نویس هم اون اون فضای ادرس ویرچوال پروسس خودش رو داره می بینه و کاری به فضای ادرس ویرچوال بقیه پروسس ها نداره

یک مدل protection هم میشه ینی هر پروسسی کلا فقط access داره به فضای ادرس خودش از طرفی اگر سیستم عامل بخواد روش هایی داشته باشه برای allocated کردن frame های حافظه ینی برای هر page بخواد یک frame ازادی در نظر بگیره توی حافظه باید یک frame table رو خودش ذخیره بکنه که این frame table به ما میگه که هر کدوم از این frame های حافظه فیزیکی کدومشون الان خالی است که اگر خواست برای یک پروسسی برای یکی از page هاش حافظه بگیره بدونه کجاها این frame های ازاد وجود دارن



Increasing or Decreasing page size

- What is the benefits of increasing page size?
- In what situation do you suggest to increase the page size?



-
اگر page size بزرگ بگیریم چه خوبی هایی داره و توی چه شرایطی دوست داریم اندازه page size مون بزرگتر باشه؟

به هر حال اگر اندازه page size رو بزرگ کردیم داریم هدر رفت حافظه رو بیشتر میکنیم و باعث میشه که یه جورایی درجه مالتی پروگرامینگ سیستم رو آوردیم پایین و تعداد پروسس های همزمانی که می تونن وجود داشته باشن کم شده و هی باعث میشه که نیاز به Swaping پیدا بشه پس بنابراین اگر ما سیستممون تعداد پروسس هاش بدونیم کمه ولی توی هر پروسسی زمان برامون مهم باشه و با حافظه کار داشته باشن تصمیم می گیریم page ها رو بزرگ تر کنیم که دسترسی هامون سریعتر بشه و سرعت اجرای برنامه ها بخاطر سرعت دسترسی بالاتر به مموری بالا بره ولی از طرفی درجه مالتی پروگرامینگ رو آوردیم پایین ینی توی همچین سیستمی باید احتمالا تعداد ماکزیمم پروسس هایی که توی سیستم هستن کم باشه که امکان گرفتن حافظه براشون باشه



Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**



سیستم ها اون page table رو چجوری پیاده سازی می کنن؟

چون به ازای هر پروسسی ما یک page table داریم بنابراین این page table هم باید توی مین مموری ذخیره بشه

وقتی که برنامه می خواد اجرا بشه دوتا رجیستر cpu می تونه داشته باشه که ادرس ابتدای page table اون پروسس رو و اندازه اون page table رو نگهداری کنه:

PTBR: مشخص میکنه که page table این پروسس توی کجای حافظه قرار داره

PTLR: سائز page table رو مشخص میکنه ینی cpu مطمئن بشه که حافظه ptotect میشه و به ادرسی که خارج از فضای ادرس اون پروسس هست دسترسی پیدا نکنه

هر دیتا یا دستوری که میخوایم بهش دسترسی پیدا کنیم دوتا دسترسی به مموری اینجا نیاز هست:

1- یکی این که بریم سراغ page table و بعد تازه وقتی که ادرس تبدیل شد به ادرس فیزیکی ما می تونیم به اون قسمت مموری واقعی دسترسی پیدا کنیم که اون دیتا و دستوری که می خوایم رو

برداریم <-- این یکمی بده چون ما به ازای هر دیتا یا دستوری دو بار دسترسی به مموری داریم

توی سیستم ها سعی کردن اینو یک مقدار بهتر کن و اون زمانی که می بره که دسترسی به page table داشتیم باشیم رو کاهش بدن <-- یکی از روش هایی که استفاده میشه استفاده از

associative memory ها است یا چیزی که اینجا برای مدیریت مموری یا page table ها

استفاده میشه TLB است که یک مدل از associative memory است



TLB

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
 - If the item is found, the corresponding value field is returned. The search is fast;
- a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty.
- To be able to execute the search within a pipeline step, however, the TLB must be kept small



پس TLB یک associative memory است و هر ایتm این TLB دو قسمت داره یک کلید و یک مقدار

و هر موقع که یکی از ایتm ها رو بخوایم همه کلید ها به صورت همزمان می تونه سرچ بشه که بتونیم اون ایتm مورد نظر رو توی TLB بتونیم پیدا کنیم

این TLB یک قسمت سخت افزاری است که توی سیستم های مدرن جدید در نظر گرفته شده و خوبیش اینه که یه جورایی این کاری که انجام میشه توی زمان instruction pipeline مون انجام میشه ینی همون موقع که cpu داره یک دستوری رو اجرا میکنه اگر یک دسترسی به حافظه ای هم توی این دستور وجود داشته باشه اون دسترسی به حافظه همزمان توی همون pipeline در نظر گرفته میشه و زمانش توی همون pipeline قرار گرفته

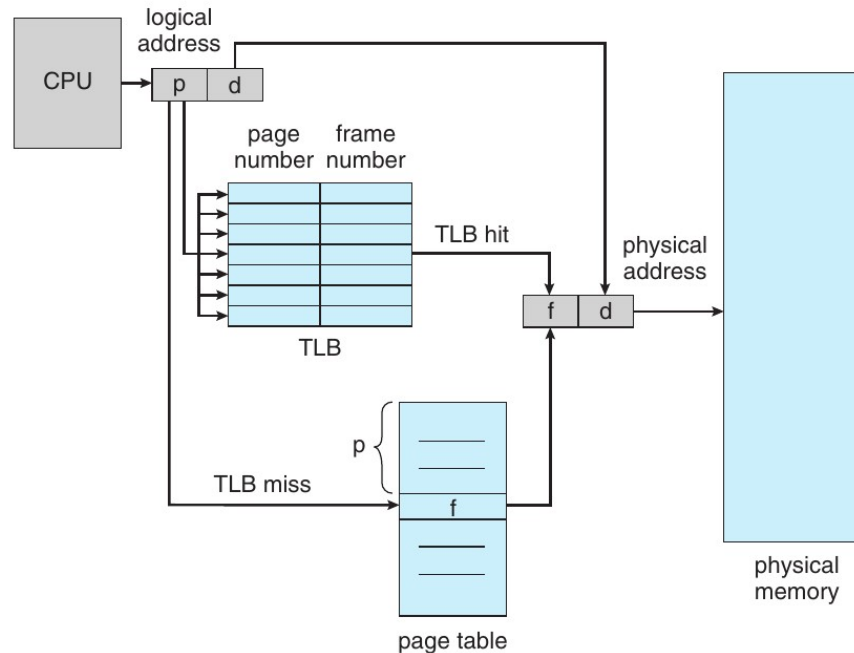
ایده اینه که ما اون page table مون رو توی TLB ذخیره بکنیم ینی هر بار که یک دسترسی داره توی cpu اجرا میشه page table اش لود بشه توی TLB پس دیگه لازم نیست که برای اینکه ما ادرس فیزیکی رو در بیاریم به مموری دسترسی پیدا کنیم به جاش به رجیسترهای TLB دسترسی داریم که خیلی سریع تر است اما مسئله اینه که ما هر بار که قضیه رو به سخت افزار بکشونیم چه محدودیتی داریم؟ سخت افزار محدودیت بیشتری برای ما داره

این TLB داره توی cpu در نظر گرفته میشه پس نمیتونه خیلی هم بزرگ باشه پس از لحاظ محدودیت های سخت افزاری و برای اینکه ما بتونیم سرچش رو توی یک pipeline در نظر بگیریم این باید کوچیک باشه پس ... ادامهش صفحه بعدی..



TLB

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
 - If the page number is found, its frame number is immediately available and is used
 - If the page number is not found, memory reference to the page table must be made.



پس ما مجبوریم هر page table که داریم مثلا توی هر لحظه ای که داره پروسس اجرا میشه یک قسمتی از page table رو لود بکنیم توی TLB

page number توی این TLB همون کلیدمون میشه

page هایی رو که احتمالا توی یک بازه زمانی مدام باهاش کار داریم اونارو میاریم توی TLB قرار میدیم اطلاعاتش رو از page table که سریعتر بتونیم ادرس فیزیکالش رو در بیاریم بنابراین ممکنه cpu یک ادرس ویرچوالی رو ببینه که شماره page اش توی این TLB نباشه پس توی اولین دسترسی که به TLB پیدا میکنه می بینه توی TLB نیست و تازه مجبوره که دوباره بیاد و چون TLB ما miss شده از توی page table ینی از توی حافظه اینو برداره و ادرس فیزیکالش رو دربیاره

اگر تعداد page ها کم باشه ینی page size ها رو بزرگ گرفتیم که تعداد page هامون کم شده باشه احتمال اینکه همش توی TLB جا بشه بیشتره یا تعداد بیشتری page رو بتونیم توی TLB نگه داریم بیشتره و باز همین دوباره باعث میشه که اون افزایش سرعت اجرا رو داشته باشیم برای برنامه هامون و خود این می تونه یک دلیل باشه که اندازه page هارو بزرگ بگیریم



Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’ logical address space
 - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel



: Memory Protection

گاهی گفته میشه که فرض کنید یک پروسس ممکنه دسترسی به یک page داشته باشه ولی فقط دسترسی خواندن داشته باشه و نتونه اون page رو عوض کنه ولی بتونه از اطلاعاتش استفاده بکنه این حالت برای چه مواقعی مفید است و اتفاق می افته؟

پس توی حالتی مثل حالتی که داریم از شیرد مموری استفاده میکنیم مثلا پروسس ها دارن از لایبرری مشترکی استفاده میکنن نمیخوان بهشون اجازه بدیم که همشون دسترسی نوشتن داشته باشن ینی فقط می تونن بخونن از اون page ها پس ما یک بیت اضافه میکنیم توی page table که به ازای هر page که نشون بده اون page برای اون پروسس در چه حالتی است

Valid-invalid: اصلا این می تونه جلوگیری بکنه ینی مثلا اگر یک بیتی invalid باشه اصلا پروسس دیگه بهش دسترسی نداره این کجا مفید است؟

پروسسمون یک ماکزیم فضای ادرسی داره و از طرفی ممکنه یک page مشخصی کلا توی حافظه اختصاص پیدا نکرده اما توی page table ما اینو داریمش ینی اگر page tableمون کامل باشه اینارو پشت سرهم توی page table داریم پس اگر این page توی حافظه قرار

نگرفته باشه ما باید یه جوری اینو مشخص بکنیم که اینجا با Valid-invalid مشخص میکنیم ینی اگر بیت اون page ما invalid باشه ینی اصلا توی حافظه نیست و اگر ما به ادرسی توی این قسمت بخوایم دسترسی پیدا کنیم اون وقت باعث میشه ک سیستم دچار خطا بشه پس برای اینکه

بتونیم اینو مشخص بکنیم این بیت رو اضافه میکنیم که cpu بتونه یه جورایی به کرنل اطلاع بده که دسترسی خطایی پیش اومده



Valid (v) or Invalid (i) Bit In A Page Table

00000	page 0
	page 1
	page 2
	page 3
	page 4
10,468	page 5
12,287	

frame number		valid-invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

page table

0	
1	
2	page 0
3	page 1
4	page 2
5	
6	
7	page 3
8	page 4
9	page 5
	⋮
	page <i>n</i>



مثال:

پس یک بیت به page table اینجا برای Protection اضافه کردیم و اون بحث read only هم میتونه اینجا جدا در نظر گرفته بشه ینی یک بیتی باشه که الان براش مشخص بکنه اون page الان فقط خواندنی است یا نه می تونه نوشتن هم باشه

اینجا فضای ادرس پروسس ممکنه بیشتر از این 6 تا page باشه ولی ما فقط این 6 تا رو داریم و بقیه page هامون توی page table نامعتبر میشن ینی invalid پس اگر یه موقعی یک دستوری توی این پروسس بود که داشت به یک ادرس ویرچوالی توی خونه 7 توی page table دسترسی پیدا میکرد باید سیستم بتونه trap ارسال بکنه پس از بیت invalid استفاده میکنه و اگر i بود می فهمه که اشتباهی رخ داده



Shared Pages

■ Shared code

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for interprocess communication if sharing of read-write pages is allowed

■ Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space



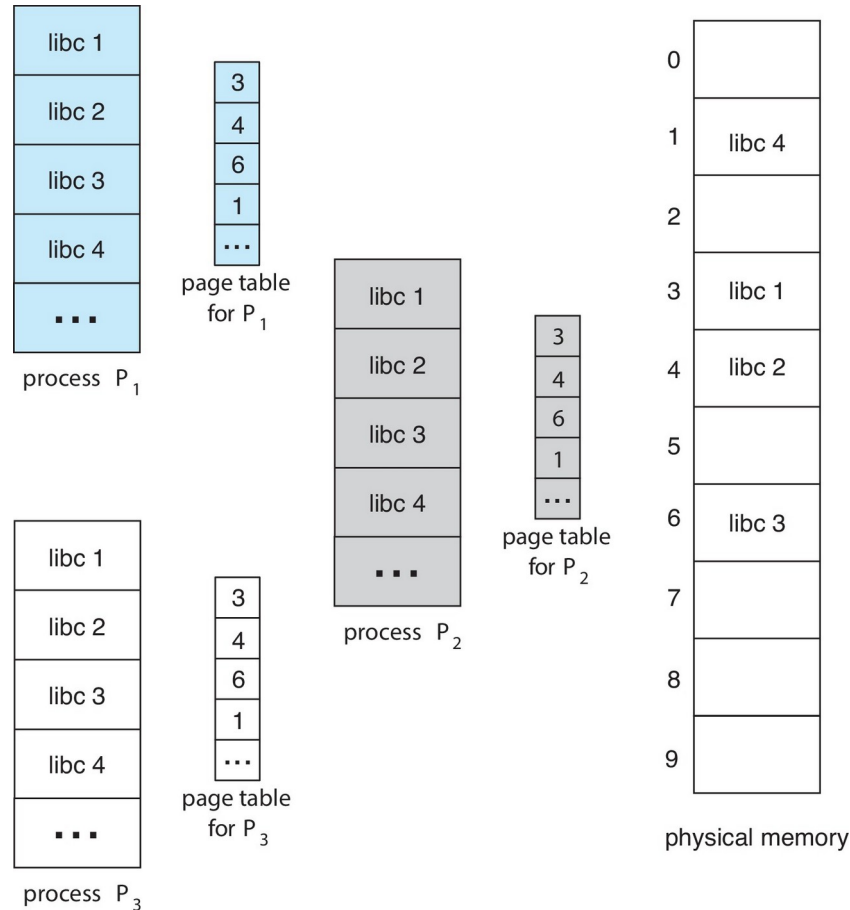
کلا page های ما می تونن یا شیرد باشن یا Private باشن

اگر Private باشن ینی اون page دیگه فقط مختص خود اون پروسس است و پروسس دیگه ای از اون page استفاده نمی کنه

ولی اگر شیرد باشه چندتا پروسس هم ممکنه همزمان استفاده بکنن



Shared Pages Example



مثال:


این لایبرری رو هم p1 و هم p3 دارن استفاده می کنن پس توی page table های این دوتا پروسس این تکرار شده

اینجا مثلا p3 , p1 از یک لایبرری p2 دارن استفاده می کنن و این p2 براشون page table هایی هستن که بینشون شیرد است برای همین توی page table شون همین ها تکرار شده ولی توی فیزیکیال مموری یکبار این ها اختصاص داده شدن و فقط کافیه که توی page table این ها همشون اشاره بکنن به همین frame هایی که توی حافظه وجود داره پس از حافظه به این صورت بهتر استفاده میکنیم

نکته: این روش paging و روش های ویرچوال و ادرس دهی ویرچوال به ما کمک میکنه که بتونیم فضای ادرس شیرد داشته باشیم بین پروسس های مختلف



Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes  each process 4 MB of physical address space for the page table alone
 - ▶ Don't want to allocate that contiguously in main memory



اندازه page table اینها:

هر چه اندازه page ها کوچکتر باشه احتمالا اندازه page table بزرگتر میشه
فرض کنیم logical address space مون بتونه 32 بیتی باشه ینی 2 به توان 32 بیت ادرس متفاوت توی logical address مون در نظر بگیریم
اگر page size هم 4KB در نظر بگیریم ینی 2 به توان 12
حالا page table باید چندتا سطر داشته باشه؟

2 به توان 32 تقسیم بر 2 به توان 12 که میشه 2 به توان 20 پس به اندازه 2 به توان 20 تا سطر
باید توی page table مون داشته باشیم ینی page table مون باید در واقع 4MB باشه ینی هر
پروسس 4MB از حافظه رو داره برای page table اش خرج میکنه
از طرفی گفتیم که page table رو اینطوری فرض میکنیم ینی یه دونه مقدار برای هر سطرش
در نظر گرفتیم چون فرض کردیم که page ها پشت سر هم اینجا ذخیره شده ینی از شماره صفر تا
اون ماکزیمم تعدادی که داریم توی page table مون قرار گرفته
اما الان که اندازه page table مون خیلی بزرگ شد اینجا سخت میشه که این 4MB رو پشت سر
هم توی حافظه ذخیره کنیم پس باید به دنبال روشی باشیم که حتی بتونیم یک page table رو
بشکنیمش و توی چندتا قسمت حافظه ذخیره کنیم
کل حرفمون این بود که اینجا اندازه page table مون 4MB شد بدون در نظر گرفتن اون بیت
های خواندنی، valid و invalid این چیزها --> این نشون میده که ما اگر بخوایم این
page table رو به صورت پشت سر هم توی حافظه نگه داریم یک مقدار برامون سخت میشه این
کار

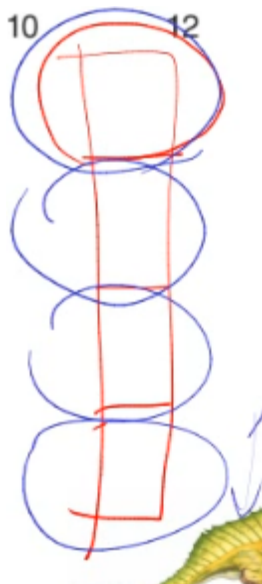


Structure of the Page Table

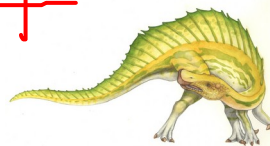
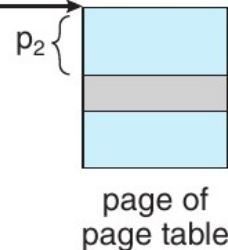
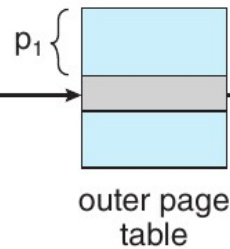
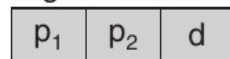
- One simple solution is to divide the page table into smaller units
 - Hierarchical Paging
 - Hashed Page Tables
 - Inverted Page Tables

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12



logical address



برای همین میان از روش های دیگه ای استفاده می کنند که بتونن اون **page table** بزرگ رو بشکنن به اندازه های کوچکتر و هر کدوم رو مجزا بتونن پشت سر هم توی حافظه ذخیره بکنن یک سری روش داریم:

روش های سلسله مراتبی

روش هایی که از هش استفاده میشه

روش های **Inverted Page Tables**

* روش **Hierarchical Paging**:

کاری که روش سلسله مراتبی انجام میده اینه که برای اون قسمت شماره **page** دوتا بخش در نظر می گیره چرا این کارو میکنیم؟ چون میخوایم اون **page table** بزرگ رو بشکنیمش به **page table** های کوچک تر ینی هر کدوم از قسمت های ابی پشت سر هم توی حافظه باشه --> عکس

اگر بخوایم یک همچین کاری بکنیم دیگه این ابی ها دنبال هم نیستن که راحت بتونیم با یک عدد ادرس دهی بکنیم پس مجبوریم قسمت اول ادرس رو بشکنیم به دو قسمت و یک همچین کاری بکنیم که اگر این 4 تا **page** داریم یک **page** دیگه هم ما باید جدا تعریف بکنیم که بهمون نشون بده که **page** اول ادرس اولش کجای حافظه است و **page** دوم ادرس اولش کجای حافظه است و **page** سوم و **page** چهارم

پس برای اینکه بتونیم هر تیکه **page table** رو یک جایی ذخیره بکنیم توی حافظه **page table** رو می شکنیم به تیکه های مختلف و الان اگر بخوایم هر قسمت از این تیکه ها رو به جایی ذخیره بکنیم تازه به جای دیگه باید ذخیره بکنیم که هر کدوم از این تیکه ها کجای حافظه است پس یک **page** دیگه هم باید بسازیم که برای این مثالی که زدیم 4 تا سطر داره که نشون میده **page table** اول از کجای حافظه شروع شده و همینطور برای **page 3** دیگه هم به همین صورت پس برای همین این تیکه اول ادرس رو دو قسمت کردیم و قسمت اولش که اسمش **outer page** همونی میشه که توی یکی **page table** دیگه قرارداره ینی برای دسترسی به این جدول استفاده میشه و قسمت دوم که اسمش **inner page** برای دسترسی به اون تیکه های **page table**

پس اینجا ما باید دوباره به صورت سلسله مراتبی دسترسی پیدا کنیم به این **page table** ها یکبار این **p1** رو برمیداریم و بعد پیدا میکنیم **page table** مورد نظر کجای حافظه است و ادرسه که پیدا شد تازه می ریم سراغ **page table** اش و بعد اینجا با استفاده از اون عددی که به دست میاد **frame** ما توی حافظه اصلی به دست میاد و بعد توی + بهش دسترسی پیدا میکنیم

مثالی که توی خود اسلاید است ینی ما داریم یک **page table** دو مرحله ای رو پیاده سازی میکنیم و اگر خواستیم بیشتر باشه می شه 3 مرحله ای