

Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali
1401

Virtual Memory



Background

- Code needs to be in memory to execute, but entire program rarely used
 - **Error code, unusual routines, large data structures**
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running -> more programs run at the same time
 - ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time
 - Less I/O needed to load or swap programs into memory -> each user program runs faster



یک برنامه وقتی که میخواد اجرا بشه باید توی مموری قرار بگیره شاید اون لحظه که داره یک قسمتی از برنامه اجرا میشه نیاز به همه کدها یا دستورهای مربوط به اون برنامه هست نداشته باشیم بنابراین وقتی یک برنامه در حال اجرا است نیاز نیست که همه اون حافظه ای که مربوط به اون برنامه است توی حافظه فیزیکال قرار گرفته باشه

از همین قضیه استفاده میکنیم برای اینکه درجه مالتی پروگرامینگ سیستم رو بالا ببریم در واقع اگر ما همه یک برنامه رو کامل لود نکنیم توی فیزیکال مموری می تونیم برنامه های بیشتری به این صورت توی فیزیکال مموری لود کنیم چون برای فیزیکال مموری یک اندازه محدودی داریم پس اگر بتونیم پروسسهای بیشتری توش لود بکنیم بهتر است و اگر ما هر پروسسی رو لود میکنیم لازم باشه تمام page های مربوط به اون پروسس رو لود کنیم می تونیم page های بیشتری از پروسس های دیگه رو لود کنیم در نهایت

بدین ترتیب CPU utilization ما هم بالا می ره بخاطر اینکه برنامه های بیشتری اماده به اجرا می تونن باشن و throughput سیستم هم بالا می ره حتی وقتی که این حرف رو می زنیم لازم نیست اگر توی سیستم نیاز به swaping داشتیم کل یک پروسس رو Swap کنیم توی اون سخت افزاری که Swap رو داره پس اینجا می تونیم page هایی که نیاز نداریم رو Swap کنیم و این کار باعث میشه که توی هر بار خارج کردن نیاز نباشه که کل یک برنامه و همه page های مربوط به یک برنامه swap بشه پس اینجا Swap کردن نیاز به I/O کمتری پیدا میکنه و ما کافیه هر موقع هر page که نمی خواستیم swap اش بکنیم و بعد از توی Swap به موقعش که نیاز داشتیم برش گردونیم توی مموری



Background (Cont.)

- **Virtual memory** – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes



Virtual memory: همون بحث قبلی که logical memory از physical memory جدا کردیم اینجا کامل تر میگیریم:

ما غیر از اینکه برای هر پروسس یک فضایی در نظر گرفتیم که خاص خودش است و اونجا گفتیم که page های این پروسس ادرس دهی جدایی دارن که بهش میگیریم ادرس ویرچوال

و مموری واقعی ما ادرس دهی دیگه ای داره که توی روش paging با شماره frame ها مشخص می کردیم و بعد لازم بود که ادرس page ها رو مپ کنیم به ادرس frame ها که از طریق روش هایی که روی MMU بود انجام میشد

اینجا ویرچوال مموری داره بهمون میگه که اون جدولی که برای روش paging استفاده میکنیم ینی جدول page table استفاده میکنیم لازم نیست که تمام page های یک پروسس رو توش داشته باشیم همیشه ینی برای همه page ها لازم نیست مقداری داشته باشیم ممکنه برای یک page هم واقعا حافظه اختصاص داده شده ولی ما توی موقع اجرا اون page رو توی فیزیکال مموری نداشته باشیم که اینو اینجا توضیح میدیم توی این اسلاید



Background (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - MMU must map logical to physical
- Virtual memory can be implemented via:
 - **Demand paging**
 - **Demand segmentation**



این به ما کمک می‌کند که فضای فیزیکیال مموری که داریم رو به صورت بهینه تری بین تعداد بیشتری برنامه استفاده بکنیم

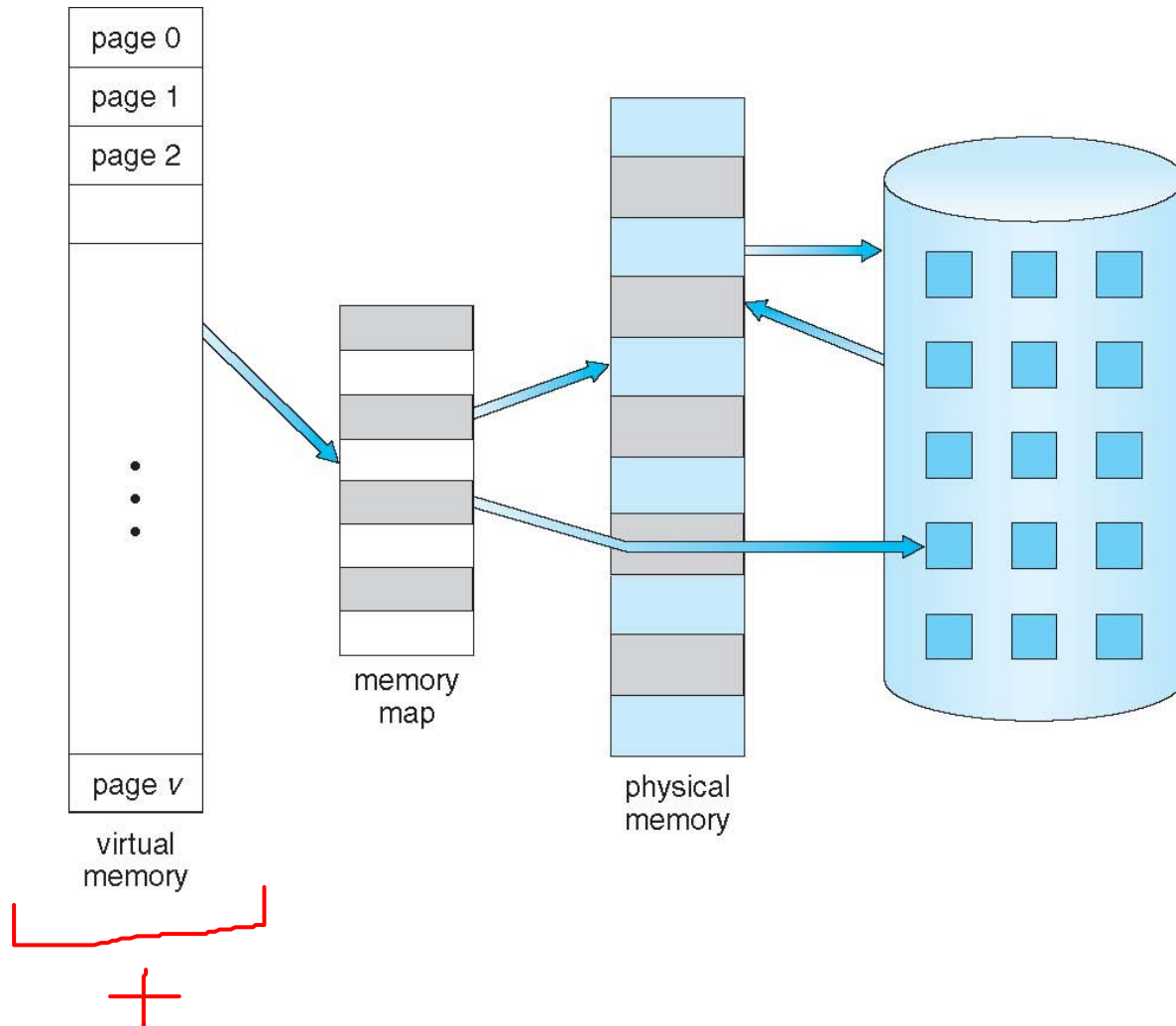
هر پروسی یک **logical view** خودش از ادرس های خودش داره که از شماره های صفر شماره گذاری شده و تا اخر برنامه تا اون ماکزیمم ادرس **logical** که براش امکان پذیر است میتونه ادرس در فضای ادرس خودش داشته باشه

ما **page** های اون پروسس باید بعدا مپ کنیم به **frame** های حافظه واقعی با توجه به اینکه توی صفحه قبلی گفتیم نمیخوایم تمام **page** های یک برنامه رو لود کنیم توی حافظه پس باید به چه صورت عمل بکنیم؟

روشی که مطرح میشه **Demand paging** یا **Demand segmentation**



Virtual Memory That is Larger Than Physical Memory



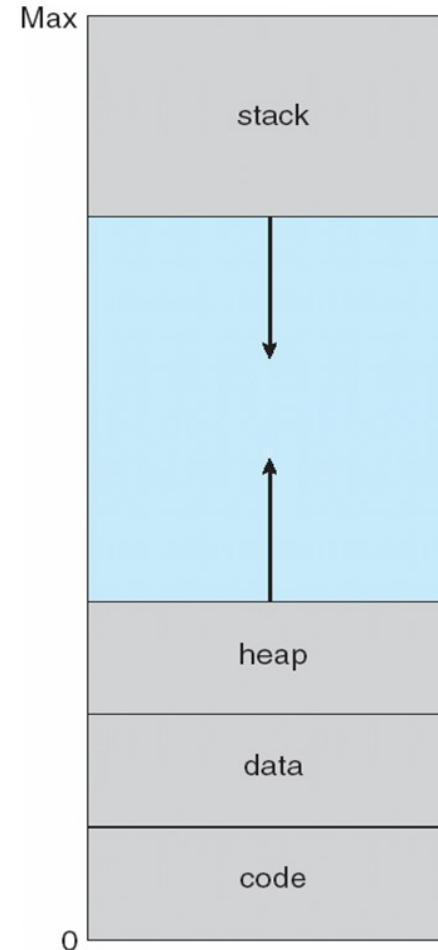
این روش می‌گه که:

وقتی که ما این برنامه رو داریم اجرا میکنیم توی حافظه + فقط کافیه که اون page هایی که الان بهشون نیاز داریم توی حافظه باشه و بقیه نیاز نیست که توی حافظه باشه که به این روش می‌گیم Demanding ینی براساس نیازمون ما page ها رو توی حافظه داشته باشیم



Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
 - Maximizes address space use
 - Unused address space between the two is hole
 - ▶ No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



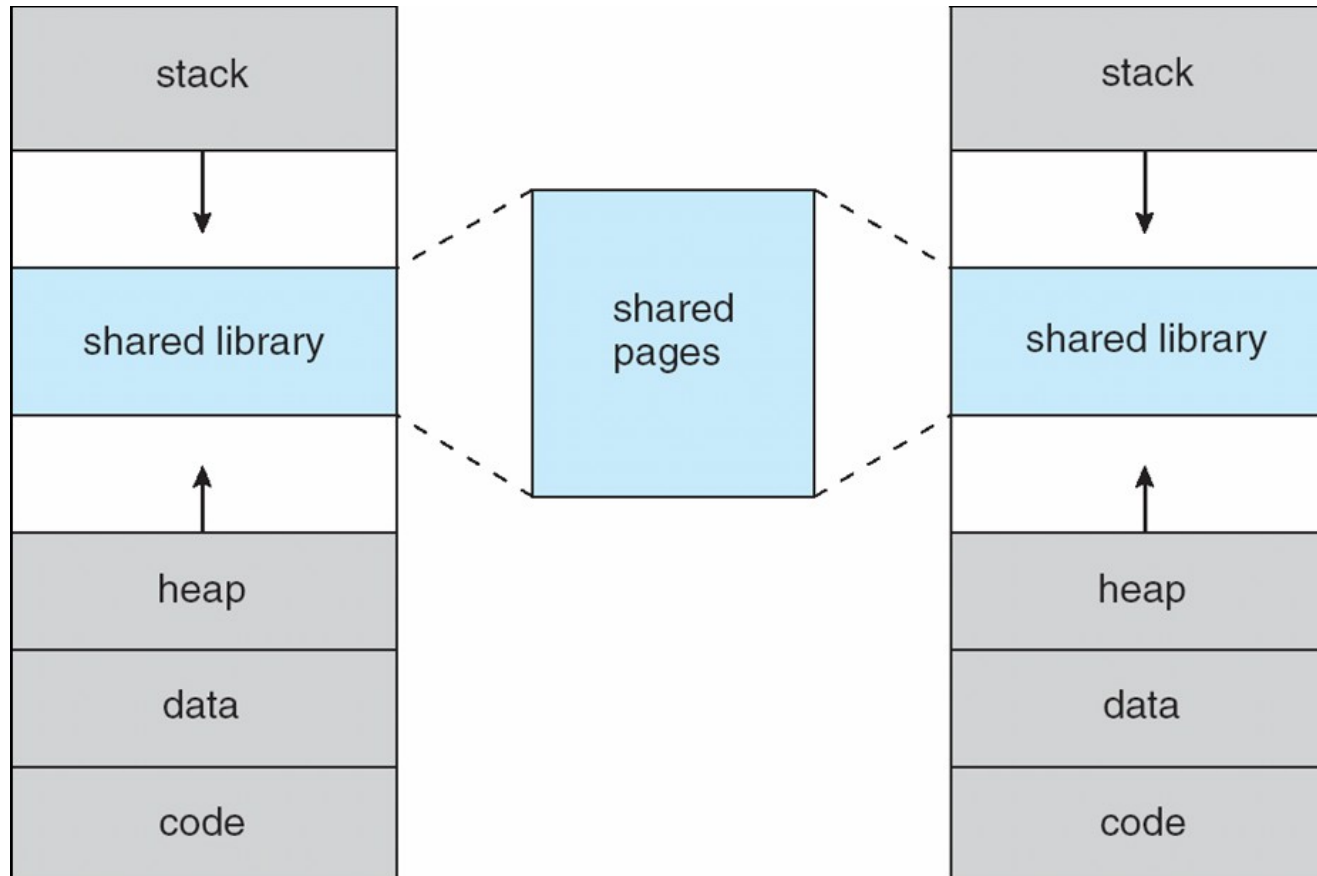
اینکه ما بر حسب نیازمون یا براساس Demanding بیاریم توی حافظه page ها اصلا بهمون کمک میکنه که بسیاری از مسائلی که قبلا گفتیم فرض میکنیم رو بتونیم پیاده سازی بکنیم مثلاً همین فضای حافظه ای که به این صورت تصور کردیم توی بحث پروسس ها و اینجوری فرض کردیم که کد از شماره صفر ادرس ویرچوال پروسس شماره گذاری میشه و بعد دیتا و بعد هیپ قرار می گیره و بعد برعکس استک از قسمت نهایی ادرس دهی میشه برای پروسس

اگر ما بحث ویرچوال ادرس و ویرچوال مموری رو نداشتیم به این صورت نمی تونستیم کار کنیم اینجا این وسط بین استک و هیپ یکسری page تصور کردیم برای پروسس که ممکنه این هارو نداشته باشیم ینی خالی باشن ینی هنوز استک و هیپ این ها رو نگرفتن و در عین حال این فضای خالی رو تونستیم بین استک و هیپ ایجاد کنیم

پس اینکه ما می تونیم Demanding داشته باشیم بهمون کمک میکنه که به این صورت بخوایم فضای ادرس رو در نظر بگیریم و براش frame اختصاص بدیم و حتی ممکنه که در Demanding الان این قسمت ابی که اصلاً وجود نداره ینی frame براشون در نظر گرفته نشده همین page های استک و هیپ و دیتا و کد هم همش ممکنه که در آن واحد توی حافظه ما نباشن و بعضی هاشون وجود داشته باشه این نوع تصور از ادرس اسپیس پروسس به ما کمک میکنه از شیرد مموری بتونیم استفاده بکنیم یا از لایبرری های شیرد بتونیم بین پروسس های مختلف استفاده کنیم



Shared Library Using Virtual Memory



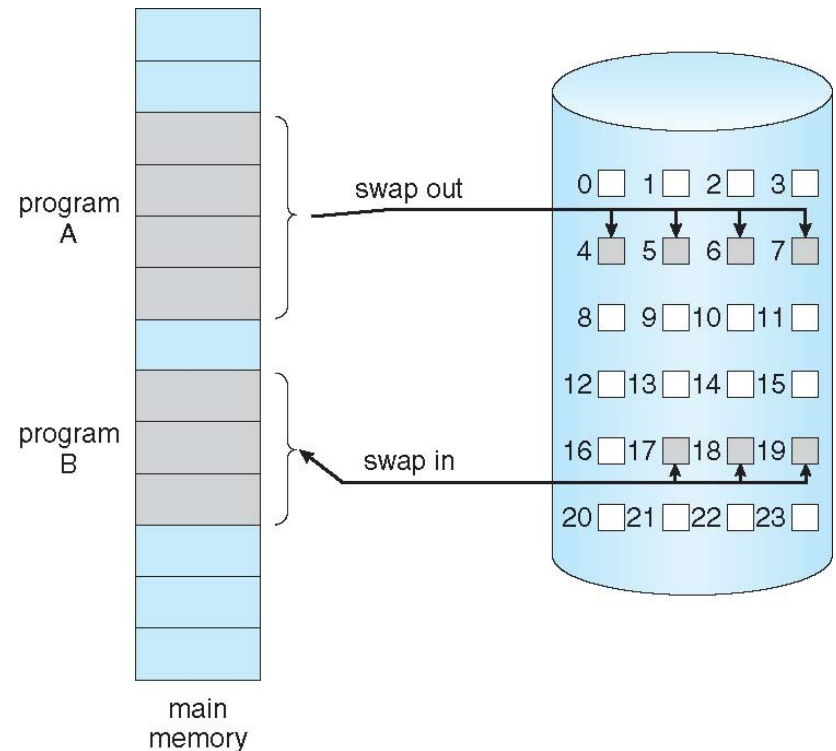
مثلا:

یکسری page شیرد بین این دوتا پروسس وجود داره که به موقع نیازشون این قسمت page هاشون که مربوط به شیرد لایبرری است مپ میشه به frame هایی که مربوط به اون شیرد page است و هر دوشون دارن از یک قسمت استفاده می کنن و اگر توی لحظه اجرا هر کدومشون در حال دسترسی به این page ها باشن این page ها روی مموری قرار میگیره



Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**



پس با استفاده از Demanding یا Demand Paging اینکه ما page ها رو براساس نیاز بیاریم توی مین مموری

قبلا میگفتیم وقتی به یک پروسس نیاز نداریم ینی اون لحظه ای که یک پروسس در حال اجرای واقعی نیست اونو swap میکنیم توی دیسک سخت ولی الان میگیریم که حتی می تونیم از یک برنامه بعضی از page هاشو ببریم روی دیسک سخت و یا وقتی که میخوایم یک برنامه رو اجرا بکنیم بعضی از page ها رو بیاد توی حافظه پس لازم نیست کل یک پروسس swap بشه بنابراین I/O کمتری نیاز داریم برای Swap کردن

و می دونم مسئله I/O مسئله مهمی است و روی کارایی ما اثر می ذاره چون I/O زمان زیادی می بره و ما دوست داریم تعداد I/O رو کم کنیم که کارایی ما بهتر بشه
اگر ما Demand Paging داشته باشیم چه اتفاقی می افته؟
ادامش صفحه 11...



Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- pager brings in only those pages into memory
- How to determine that set of pages?
 - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
 - No difference from non demand-paging
- If page needed and not memory resident
 - Need to detect and load the page into memory from storage
 - ▶ Without changing program behavior
 - ▶ Without programmer needing to change code




اصلا چجوری باید این کار ها رو انجام بدیم؟
کدوم دسته از page ها رو از اون ابتدا بیاریم توی حافظه؟
دراین صورت نباید اختلالی توی روند برنامه رخ بده پس ما نیاز به ترفندهای سخت افزاری و هم نرم افزاری داریم



Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** \Rightarrow in-memory – **memory resident**, **i** \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	i
...	
	i
	i

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** \Rightarrow **page fault**



یکی از ترفندهای سخت افزاری همین بیت Valid-Invalid که به page table اضافه میکنیم که این بیت هم باید توی TLB باشه و هم توی page table که توی مموری نگهداری میکنیم که از طریقش بتونیم متوجه بشیم که الان page که بهش دسترسی پیدا کردیم الان توی مموری هست یا نیست

اگر این page مورد نظر که بهش دسترسی پیدا کردیم مثلاً اینجا + باشه که بیتش 1 باشه در این حالت میگیریم یک page fault اتفاق افتاده یکی page که مورد نظر ما بوده وجود نداشته این page fault رخ دادن خودش چیزی بدی است چون باعث میشه که اجرای برنامه یک تاخیری توش به وجود بیاد و لازم باشه که ما برگردیم توی swap و اون page مورد نظر رو لود کنیم و دوباره دستور رو اجرا کنیم



Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

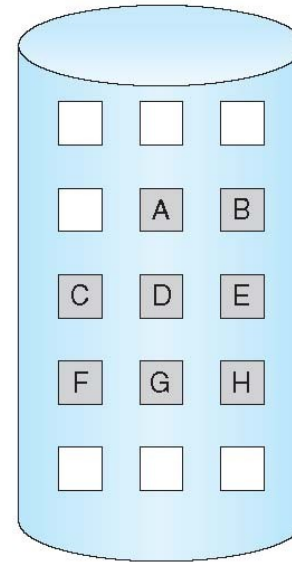
logical
memory

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

page table

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

physical memory



اگر ما Demand Paging داشته باشیم و جدول طوسی سمت چپ ویرچوال مموری یک پروسس باشه

وقتی که این پروسس داره اجرا میشه بعضی از page ها الان هنوز توی حافظه فیزیکیال ما نیستن مثلاً از این برنامه فقط A, C, F توی فیزیکیال مموری ما است و بقیشون نیستن سمت راست یک دیسک سخت است که همون swap در نظر گرفته شده که تمام page های این پروسس توی این دیسک سخت قرار دارن ولی فقط بعضی هاشون توی مین مموری است برای همین توی page table ما یک ستون اضافه کردیم به اسم ستون valid-invalid که در واقع یک بیت است که مشخص میکنه که یک page مشخصی الان واقعا توی مموری قرار داره یا توی مموری نیست

ممکنه توی همین page table برای بقیه سطرهاش هم مقدار وجود داشته باش ولی توی مموری نباشه و اون بیتش i باشه مثلاً سطر 4 یک مقدار داره ولی بیت valid-invalid اش i هست پس متوجه میشیم که این شماره page الان توی فیزیکیال مموری ما نیست و باید بریم از روی دیسک سخت بیاریمش توی فیزیکیال مموری و هر کجا رو که بهش اختصاص دادیم توی فیزیکیال مموری اون شماره frame رو داخل page table قرار میدیم

اینجا یک اتفاقی می افته: وقتی که ما همه page های لود نکرده باشیم یک موقعی ها هست که برنامه وقتی که داره اجرا میشه به یک page رفرنس پیدا میکنه که وجود نداره الان توی حافظه فیزیکیال ما پس باعث میشه یک trap اتفاق بیوفته ینی cpu که میخواد اون دستور رو اجرا کنه نتونه اجرا بکنه چون به یک دیتایی که باید دسترسی پیدا میکرد توی page هست که الان توی مموری نیستش پس trap میده و سیستم عامل متوجه میشه که trap اتفاق افتاده و بعد بخواد فکر کنه که چه کاری انجام بده

سیستم عامل باید توی این حالت چک بکنه که این trap که اتفاق افتاد بخاطر اینه که یک رفرنس اشتباهی به یک page صورت گرفته

این trap ممکنه به دو دلیل اتفاق بیوفته:

1- یا مثلا توی قسمت طوسی یک کدی داشته اجرا میشده از این پروسس که دسترسی به page شماره 8 رو نیاز داشت که اصلا 8 وجود نداره توی این فضای ادرس

2- یا اصلا دسترسی به page خواسته داشته باشه که توی قسمت خالی برنامه است بخش ابی رنگ صفحه 6 منظوره <-- که این قسمت هنوز نرفته توی قسمت ادرس ویرچوالی که در حال استفاده باشه در این دو حالت cpu یک trap ایجاد میکنه برای سیستم عامل و سیستم عامل باید اون موقع چک بکنه اون page که الان قرار بوده بهش دسترسی پیدا بشه جز page های مجاز اون پروسس است یا نه اگر نبود که اصلا باید abort بود ینی اگر اصلا اون page که میخواست بهش دسترسی پیدا کنه خارج از این فضای ادرس ویرچوال پروسس بود مثل همین شماره 8 یا نه داخل این فضا بود ولی اصلا هنوز برنامه ازش استفاده نکرده بود ینی حافظه بهش اختصاص نداده بود اینجا باید کلا abort بوده

یا اینکه نه این page بوده که استفاده میشه توی اون برنامه ولی الان توی مموری نیست و اگر الان توی مموری نیست الان باید یک ماژولی از سیستم عامل بیاد اونو از توی swap بیاره توی مموری و بعد از اون باید page table اپدیت بشه و بعد که page table اپدیتش بشه و اون بیت valid-invalid اش هم بشه v و الان تازه باید دوباره دستور رو بدیم به cpu تا از ابتدا اجرا کنه <-- ینی این وسط داشت یک دستوری اجرا میشد که این اتفاقات افتاد و trap ایجاد شد بعد از اینکه page مورد نظرش اومد توی مموری باید ریستارت بشه و دوباره اون دستور از ابتدا اجرا بشه



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

page fault

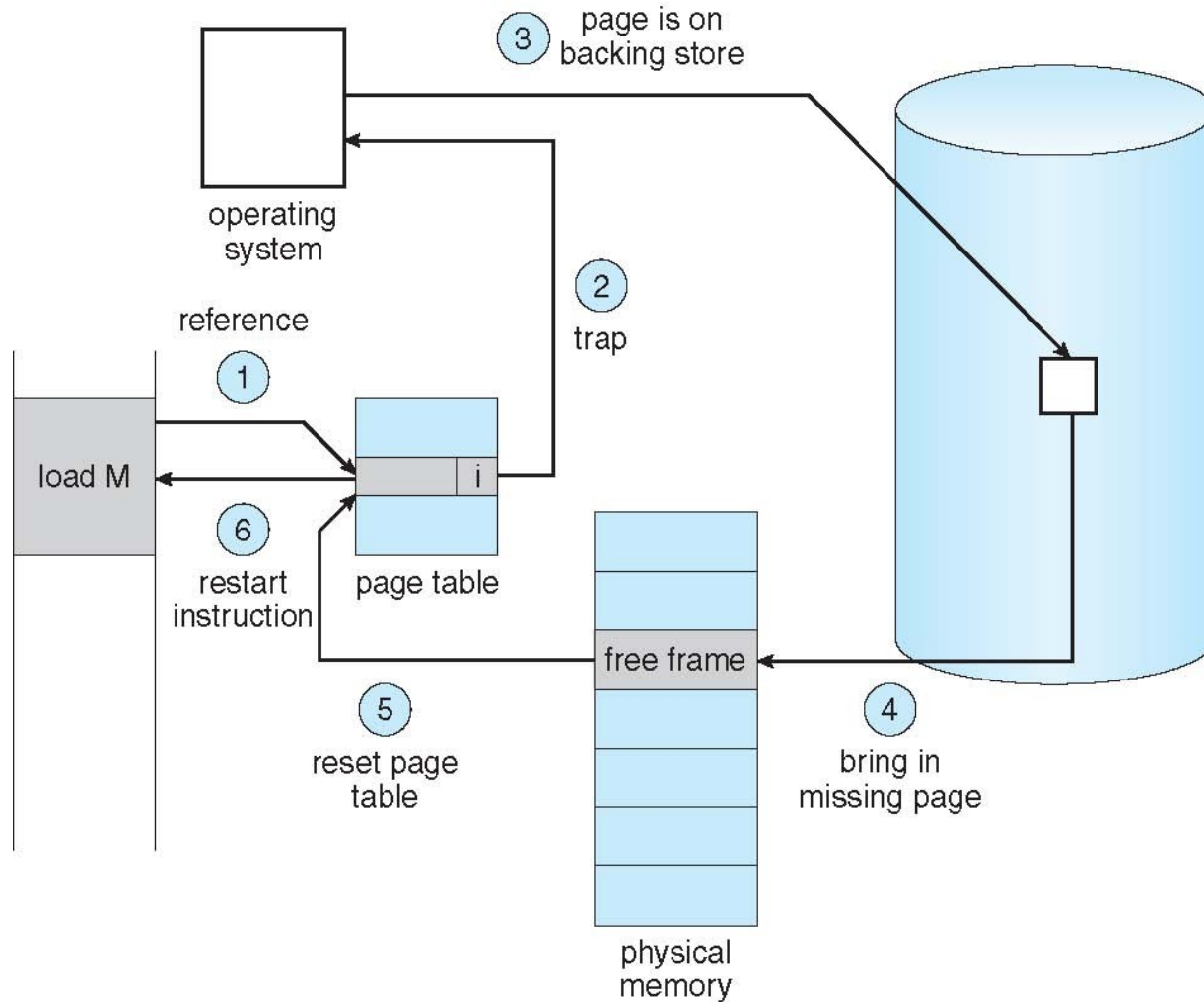
1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
Set validation bit = **v**
5. Restart the instruction that caused the page fault



و تازه وقتی که میخوایم اون page بیاریمش توی مموری باید یک frame ازادی رو پیدا کنیم توی مموری و اونو قرار بدیم توی مموری و ممکنه همه frame ها پر باشن با توجه به اینکه گفتیم داریم این کارو می کنیم که فیزیکال مموری بزرگتری نسبت به ویرچوال مموری داشته باشیم پس ممکنه وقتی که داریم یک page رو از توی Swap میاریم توی مموری اصلا فضای ازادی نداشته باشیم توی مموری حالا باید اینجا چی کار بکنیم؟ جلوتر میگیریم



Steps in Handling a Page Fault



یک برنامه ای داشته اجرا میشده و یک قسمتیش رفرنس میخوره به یک page که بیتش i است پس
cpu اینجا وقتی که مراجعه میکنه به paga table و می بینه بیتش i است باعث میشه یک trap
ارسال بکنه به سیستم عامل و سیستم عامل باید چک بکنه این Trap چرا اتفاق افتاده و اگه ببینه
علتش اینه که page توی مموری نبوده باید بره از توی Swap اون page مورد نظر رو پیدا کنه
و یک frame ازاد توی فیزیکیال مموری پیدا کنه و اون page رو قرار بده توی frame ازاد و
page table رو هم اپدیت کنه ینی شماره frame توی page table قرار بگیره و بیتش هم v
بشه و ریستارت میکنه اون دستور رو که دوباره اجرا بشه



Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
 - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
 - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with **swap space**)
 - Instruction restart



اگر این روش Demand Paging رو به کار ببریم مسائل زیر را خواهیم داشت:

یک روش pure که برای Demand Paging تعریف میشه اینه که ببینیم اصلا ما اون برنامه رو که می‌خوایم اجراش بکنیم اون پروسس‌مون رو از اون ابتدا ببینیم کدوم دستور می‌خواد اجرا بشه و بعد همون page که دستور داخلش هست رو همون page رو لود میکنیم

اگر برای اجرای اون دستور خوردیم به یک دیتایی یا یک کدی که الان توی این page مورد

نظری که الان داریم نیست می‌ریم و اون page بعدی رو لود می‌کنیم توی حافظه پس توی حالت

Pure demand paging از صفر شروع میکنیم --> اول پروسس استارت میشه و هیچ page

توی مموری نداره و بعد شروع میکنه به اجرا کردن و اولین page که نیازه اجرا بشه رو میاره

توی مین مموری و اجرا میکنه و اگر برای اجرای دستورهایش نیاز به page های دیگه ای بود اون

ها رو یکی یکی میاره توی مموری قرار میده حالا این باعث میشه چه اتفاقی بیوفته؟ توی همون

ابتدای شروع اجرای برنامه یه عالمه page fault اتفاق می‌افته و این خیلی بده چون استارت اون

پروسس خیلی باتاخیر انجام میشه چون امکان این page اولیه که لود شده و دستور یا فرمان داره

ازش اجرا میشه ادرس‌هایی توش باشه که توی همین page نباشه وجود داره و چون ما از کل اون

پروسس همین یه page رو آوردیم باعث میشه که نیاز باشه که یک page دیگه بیاد یکی page

fault اتفاق می‌افته و برای page بعدی هم ممکنه همینطوری بشه باز

پس یک مشکلی که Pure demand paging داره اینه که در ابتدای اجرای برنامه ما دچار

page fault های زیادی میشیم باعث کندی اجرای برنامه میشه و حتی ممکنه احساس بشه این

موضوع

وقتی که page fault اتفاق می‌افته بعد از اینکه سیستم عامل page مورد نظر رو آورد توی

حافظه دوباره باید cpu کار رو ریستارت بکنه و از اول اجرا بکنه و ممکنه حتی توی اجرای

دوباره هم باز به page fault بخوره --> جلوتر با یک بحثی به نام locality of reference

اشنا میشیم و می‌فهمیم خیلی هم وضع ناجور نیست

برای اجرای این Demand Paging یکسری پیاده‌سازی‌های سخت افزاری نیاز است:

توی اسلاید هایلایت کردم



Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- `vfork()` variation on `fork()` system call has parent suspend and child using address space of parent
 - Designed to have child call `exec()`
 - Very efficient
 - Ex. for programming a shell



بخاطر روش Demand Paging می توانیم از Copy-on-Write استفاده بکنیم:
توی بحث Demand Paging یه موقعی ها مجبور میشیم که یک page رو از حافظه خارج
کنیم یا اصلا ازا ابتدا نیاوردیمش توی حافظه و بعد ممکنه توی یه زمان دیگه ای اون page رو بیاریم
توی حافظه قرار بدیم

اگر توی اون لحظه ای که میخوایم اون page رو بیاریم توی حافظه قرار بدیم حافظه فیزیکیال ما
پر باشه و جا نداشته باشیم در این حالت باید یکی از page هایی که توی حافظه قرار دارد رو باید ببریم
توی swap قرار بدیم --> ادامه صفحه 20...

Copy-on-Write: ینی فقط موقعی page رو کپی بکنیم توی swap که رایتی توش انجام شده باشه
نسبت به قبل در غیر اینصورت نیاز نیست که ما رایتش بکنیم در این حالت کارایی رو خیلی می بریم بالا
ورژن های جدید fork از همین Copy-on-Write استفاده میکنه --> ادامه صفحه 16...
vfork یکی از ورژن های fork است که عملکرد متفاوتی از fork داره در واقع از Copy-on-Write
استفاده نمیکنه پس چی کار میکنه؟ vfork دقیقا از همون page های والد استفاده میکنه ینی وقتی که ما
vfork می زنیم فرزندی ایجاد میشه که دقیقا از page های والد استفاده میکنه وحتی Copy-on-Write
هم نمیکنه ینی اگر فرزند خواست یک دیتایی رو توی C تغییر بده میتونه تغییر بده و براش این page
جداگانه کپی نمیشه

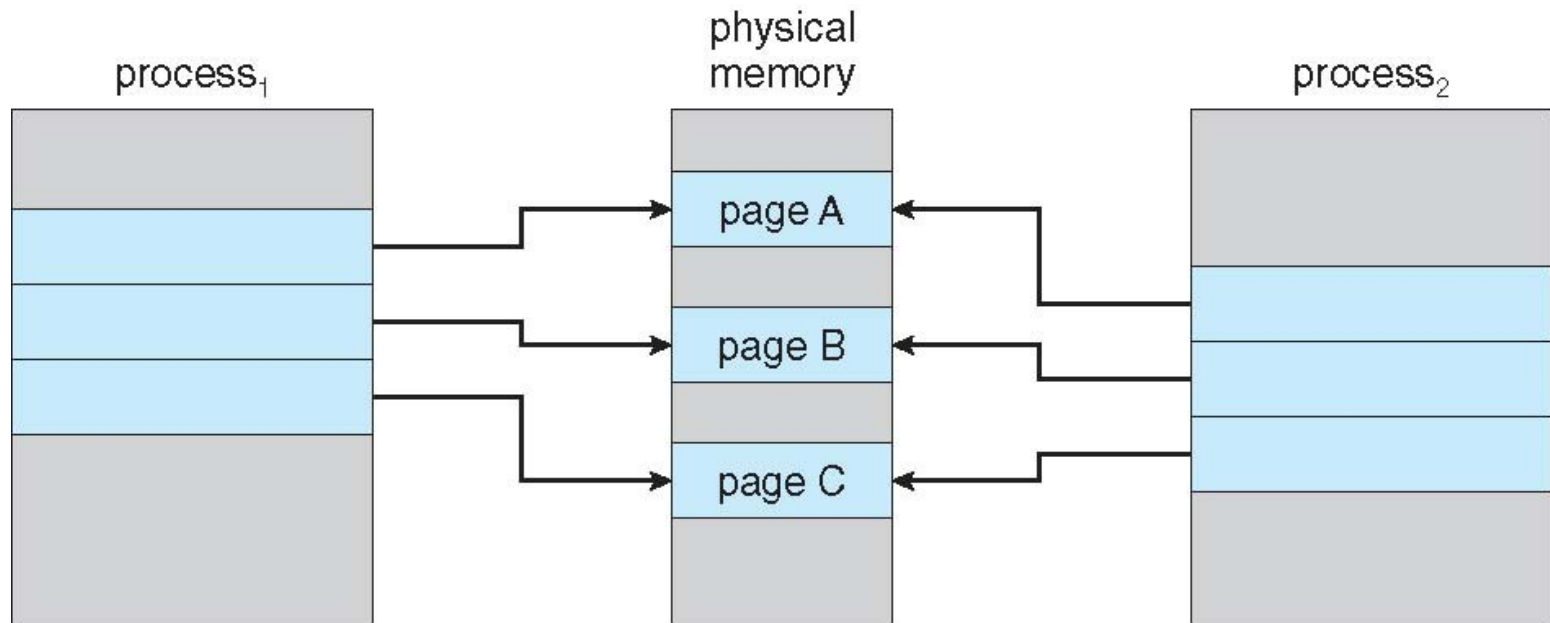
ولی vfork به این صورت عمل میکنه که به محض این که پروسس فرزند ایجاد شد و رفت توی حالت
اجرا پروسس والد رو suspend اش میکنه پس پروسس والد در آن واحد از این page ها در حال استفاده
نیست ولی نکته اش اینه که وقتی فرزند از حالت اجرا برگشت و تمام شد و بعد پروسس والد بازگشت
اینجا اگر اون دیتای قبلی مهم بوده باشه اینجا دیگه دیتا از دست رفت مثلا توی C و دیتاها توسط دست
کاری شده

پس برای استفاده از vfork باید خیلی دقت بکنیم از vfork جاهای محدودی استفاده میشه و برای این بوده
که ما سرعت ایجاد این پروسس فرزند رو بالا ببریم که نیاز به کپی کردن page هاش نباشه و از مموری
هم بهتر استفاده بشه ولی خب این مشکل رو هم داره

مثلا از vfork جاهایی استفاده میشه که قراره پروسس فرزند exec بزنه ینی پروسس فرزند بعد از اینکه
ساخته شد یک پروسس دیگه ای رو لود بکنه - معمولا vfork توی پیاده سازی های برنامه های شل
استفاده میشه



Before Process 1 Modifies Page C



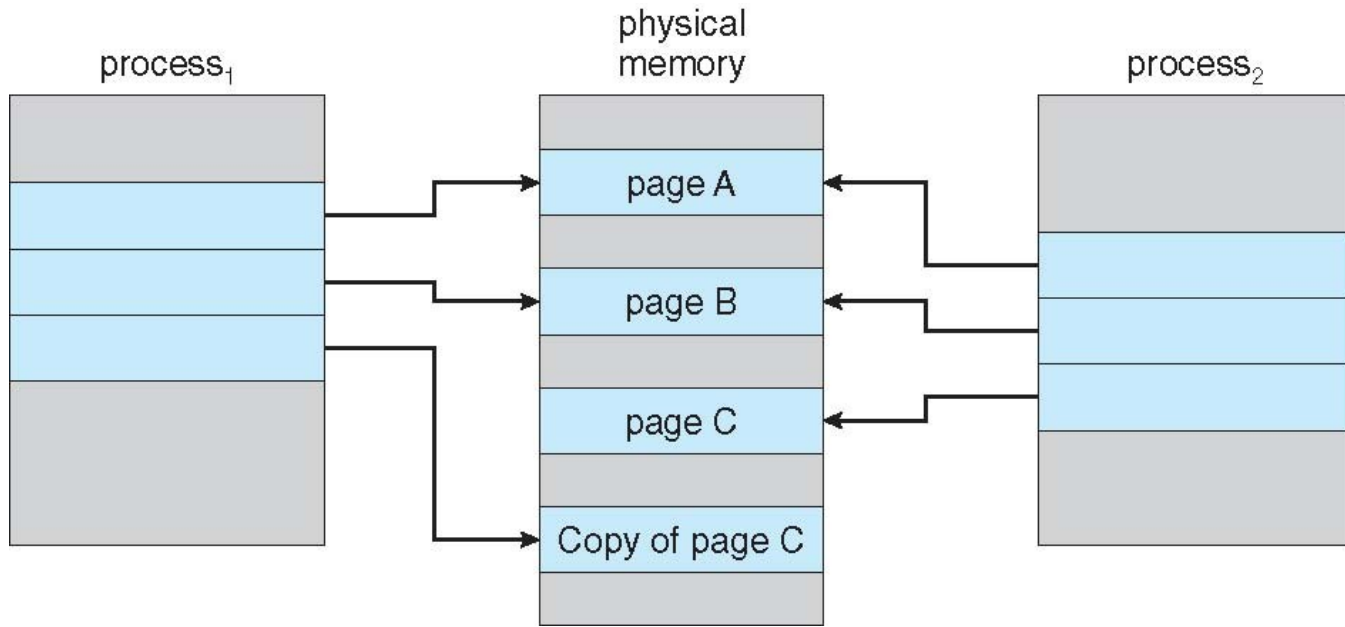
مثلا این پروسس 1 اگر پروسس والد باشه پروسس فرزند وقتی که ایجاد میشه دقیقا یک کپی از پروسس والده و page های A,B,C اگر از ابتدا مربوط به پروسس والد بوده باشه کافیه که page table فرزند به همین page ها اشاره بکنه

حالا اگر جایی پروسس فرزند دیتایی رو عوض کرد که توی این page ها قرار داره مثلا دیتایی که توی page C بود رو اگر تغییرش داد اونجا نیاز میشه که ما یک کپی از page C تازه بگیریم برای پروسس والد یا فرزند --> مثل صفحه بعدی..

پس ما تا وقتی که توی فرزند هیچ کدوم از دیتا و کدهای والد رو تغییر ندادیم از همون page های والد استفاده میکنیم ولی به محض اینکه اولین تغییری رو خواستیم انجام بدیم تازه اصلا برای اون page والد کپی می گیریم توی فیزیکال مموری --> پس از مموری به صورت بهینه تر می تونیم استفاده بکنیم



After Process 1 Modifies Page C





What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
 - Algorithm – terminate? swap out? replace the page?
 - Performance – **want an algorithm which will result in minimum number of page faults**
- Same page may be brought into memory several times



-
به هر حال یک موقعی پیش میاد که ما بخوایم یک page رو بیاریم توی حافظه ولی frame ازاد براش نداشته باشیم؟ در این حالت باید چی کار کنیم؟
ممکنه که اگر ما page مناسبی رو برای جایگزینی انتخاب نکنیم منجر به این بشه که ما مدام page faults های دنباله هم داشته باشیم و یک page دفعات زیادی بیاد توی حافظه اصلی و دوباره بره توی swap و دوباره برگرده
پس الگوریتمی که داریم استفاده میکنیم باید مانع این کار بشه



Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

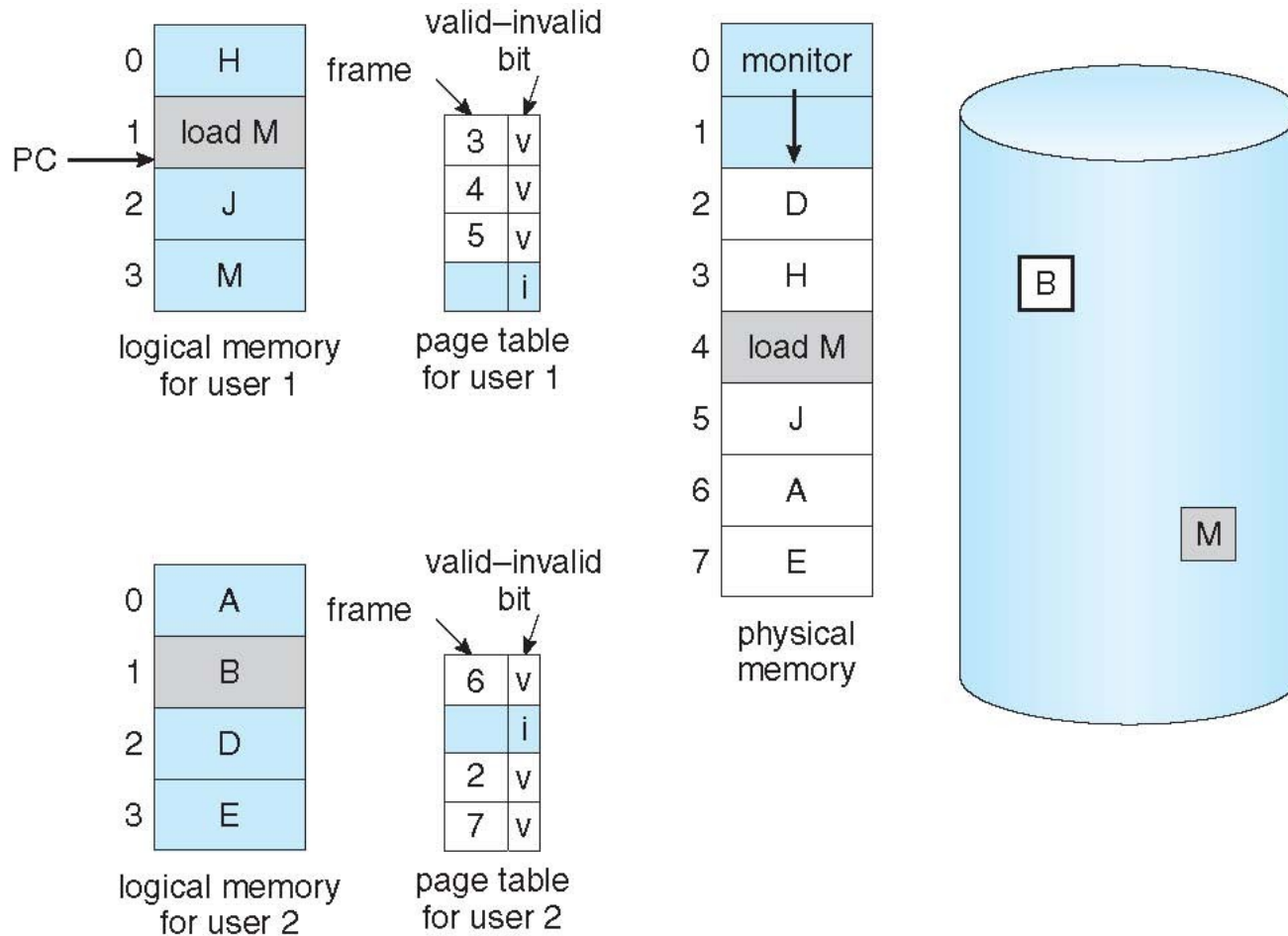


Page Replacement:

اینکه ما چه page رو باید Replac کنیم
نکته: برای قضیه Copy-on-Write ما می‌تونیم اینجا یک بیت دیگه ای هم اضافه بکنیم به نام dirty که از این استفاده میکنیم برای اینکه بفهمیم که ایا page تغییری پیدا کرده یا نه و اگر تغییر پیدا کرد کپیش بکنیم روی دیسک وگرنه Replac اش نیاز به کپی اون page اولیه توی Swap هم نخواهد داشت



Need For Page Replacement



اینجا دو تا پروسس وجود داره
فرض میکنیم قبل از اینکه M لود شده باشه توی حافظه توی اون مکان B لود شده بود
وقتی که نیاز میشه M لود بشه مجبور می شیم که این B رو ببریم توی swap و به جاش M لود
بشه جای B

اینجا نیازه که دوتا I/O انجام بشه <-- یک page بره توی Swap و یک page از Swap بیاد
توی حافظه <-- این کار زمان بری است

ممکنه این page B که میخوایم جایگزینش بکنیم با M اصلا نسبت به قبل این page B تغییری
نکرده چون این page B توی Swap هم بوده یه زمانی و الان مثلا می بینیم که نسبت به قبل
اصلا این B تغییری نکرده در این حالت دیگه لزومی نداره که B رو کپی بکنیم توی swap و بعد
M بیاریم جاش فقط کافیه اینجا M رو بیاریمش جای B و بعد page table ها رو اپدیت بکنیم
اینجا باعث میشه که از یکسری I/O ها جلوگیری بشه به این کار Copy-on-Write
ادامش صفحه 15...



Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault



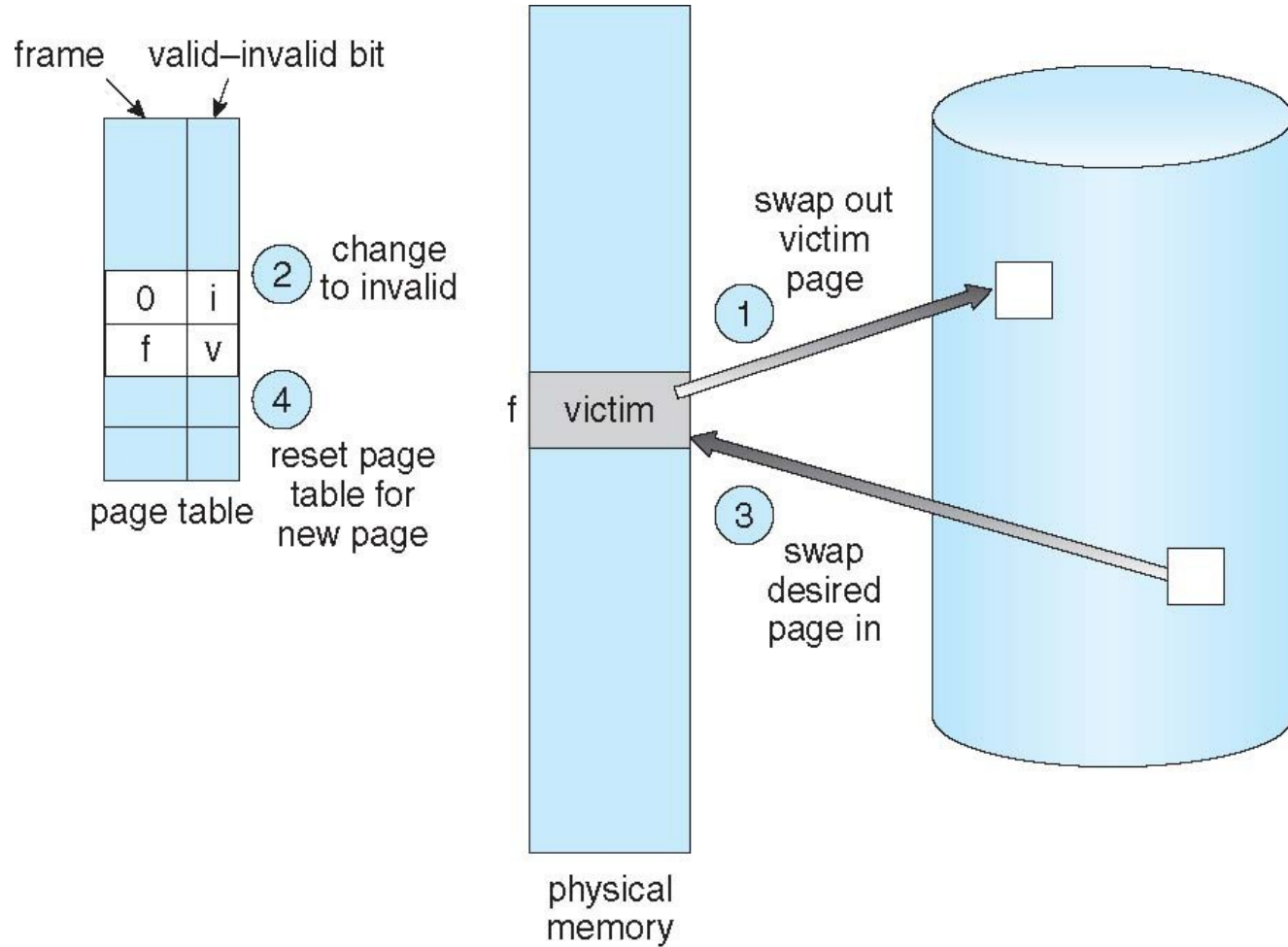
پس به طور کلی سناریویی که اتفاق می افتد اینه که:

page که میخوایم توی حافظه نیست و ما جایی هم توی حافظه نداریم برای اینکه اون page رو بیاریم

اینجا باید یک page قبلی قربانی بشه و از توی حافظه دربیاد و اگر بیت dirty اش هم یکه ینی رایت روش انجام شده باید روی Swap هم ذخیره بشه و بعد اون page جدید از روی swap برداشته باشه و بیاد روی اون frame خالی ما قرار بگیره و بعد پروسس ریستارت بشه تا اون دستور قبلیه که در حال اجرا بود بتونه اجرا بشه



Page Replacement



مثال:



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
 - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



الان بحث اینه که این قربانی کدوم page باشه؟

الگوریتم Frame-allocation: ینی از اون ابتدا که مثلا می خوایم برنامه رو اجرا بکنیم چندتا frame بهش اختصاص بدیم یا اینکه کدوم page رو قرار بدیم..

الگوریتم Page-replacement: اگر ما frame ازادی نداشتیم و page رو خواستیم بیاریم توی حافظه کدوم page قبلی رو replace کنیم؟

توی این بحث Page-replacement چه مسئله ای مهمه؟ اینه که ما این الگوریتم رو به نوعی انتخاب بکنیم که کمترین تعداد page fault رو داشته باشیم ینی با اجرای این الگوریتم با کمترین تعداد page fault مواجه بشیم هم توی دسترسی اولیه و هم توی دسترسی های بعدی

این الگوریتم خودش هم می تونه مسائل دیگه ای داشته باشه: مثلا پرفرمنس اجرای خود این الگوریتم به چه صورت است؟ و چقدر سریع میشه اون page قربانی رو پیدا کرد؟

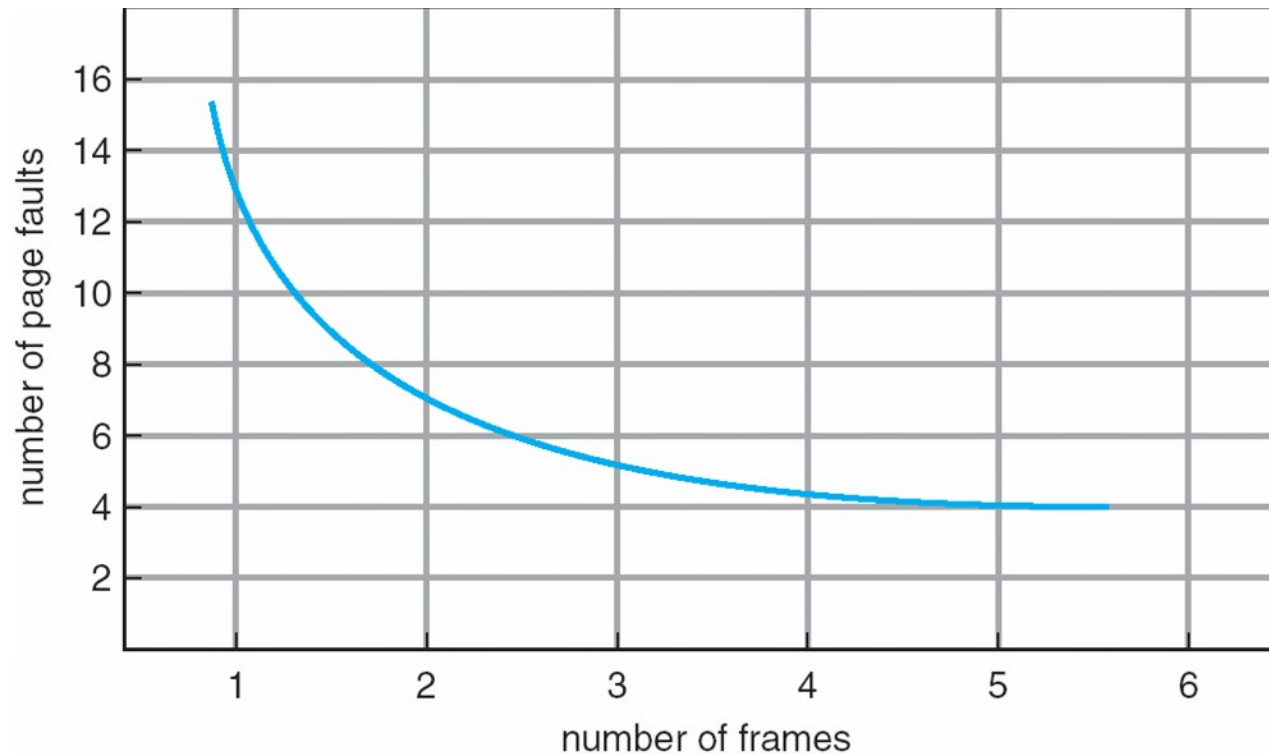
پس دوتا چیز مهم میشه: یکی این که ما توی الگوریتممون دنبال این هستیم که کمترین page fault اتفاق بیوفته و دوم هم اینه که خود الگوریتم با over head کمی اجرا بشه

توی این روش هایی که میخوایم بگیم از یک reference string استفاده میکنیم و reference string چی هست؟ منظورمون یک رشته ای از شماره page ها هست که توی اجرای یک پروسس بهشون دسترسی پیدا میشه

مثلا: من برای اجرای یک پروسس page شماره 7 رو نیاز داریم پس 7 باید بیاد توی حافظه و page بعدی که بهش نیاز میشه page شماره 0 است پس 0 هم باید بیاد توی حافظه و به همین ترتیب... پس در کل این عددهای قرمز شماره page هایی است که دسترسی بهش داده میشه توی اجرای دستورها



Graph of Page Faults Versus The Number of Frames



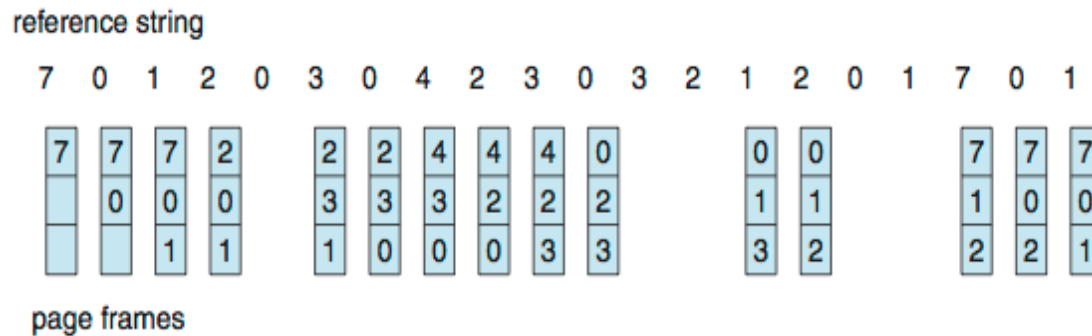
انتظار داریم که اگر تعداد frame های موجود اون فیزیکیال مموری رو ببریم بالا تعداد page fault هایی که اتفاق می افته باید کمتر باشه

نکته: جلوتر می بینیم این تعداد page fault ها وابسته به الگوریتممون هم داره ینی ممکنه ما یک الگوریتم Replacement رو خوب انتخاب نکنیم و باعث بشه page fault های بعدیمون بیشتر بشه ینی ما با وجود محدودیت حافظه فیزیکیمون دنبال الگوریتم های Replacement هستیم که ما رو به page fault کمتری برسونه



First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - ▶ **Belady's Anomaly**
- How to track ages of pages?
 - Just use a FIFO queue



-

اولین الگوریتمی که بررسی میکنیم از الگوریتم Replacement الگوریتم FIFO است:

فرض میکنیم 3 تا frame بیشتر توی حافظه فیزیکی نداریم

همین اول اگر با Demand Paging جلو رفته باشیم یعنی اول بار همه frame ها خالی تصور

کرده باشیم پس توی اون لحظه اول به یک page fault خوردیم که منجر به این میشه که اون

7 page بیاریمش توی frame های حافظمون --> توی عکس رنگ های نارنجی میشه که در کل

میشه 15 تا page fault

وقتی که 3 تا frame پر شد مجبوریم یکی از frame ها رو خالی بکنیم و برای خالی کردن این از

روش FIFO میریم

برای پیاده سازی به چه نیاز داریم؟ برای بحث over head اش باید بریم سراغ این که اینو

چطوری باید پیاده سازی بکنیم؟ کافیه که یک صف FIFO اینجا داشته باشیم که از لحاظ پیاده سازی

این خیلی خوبه --> با اردر یک می تونیم این ها رو مدیریت بکنیم توی صف و با اردر یک هم از

توی صف برداریم پس over head این روش خیلی کم است

اما مسئله ای که برای این روش وجود داره که یک نکته منفی ای است Belady's Anomaly

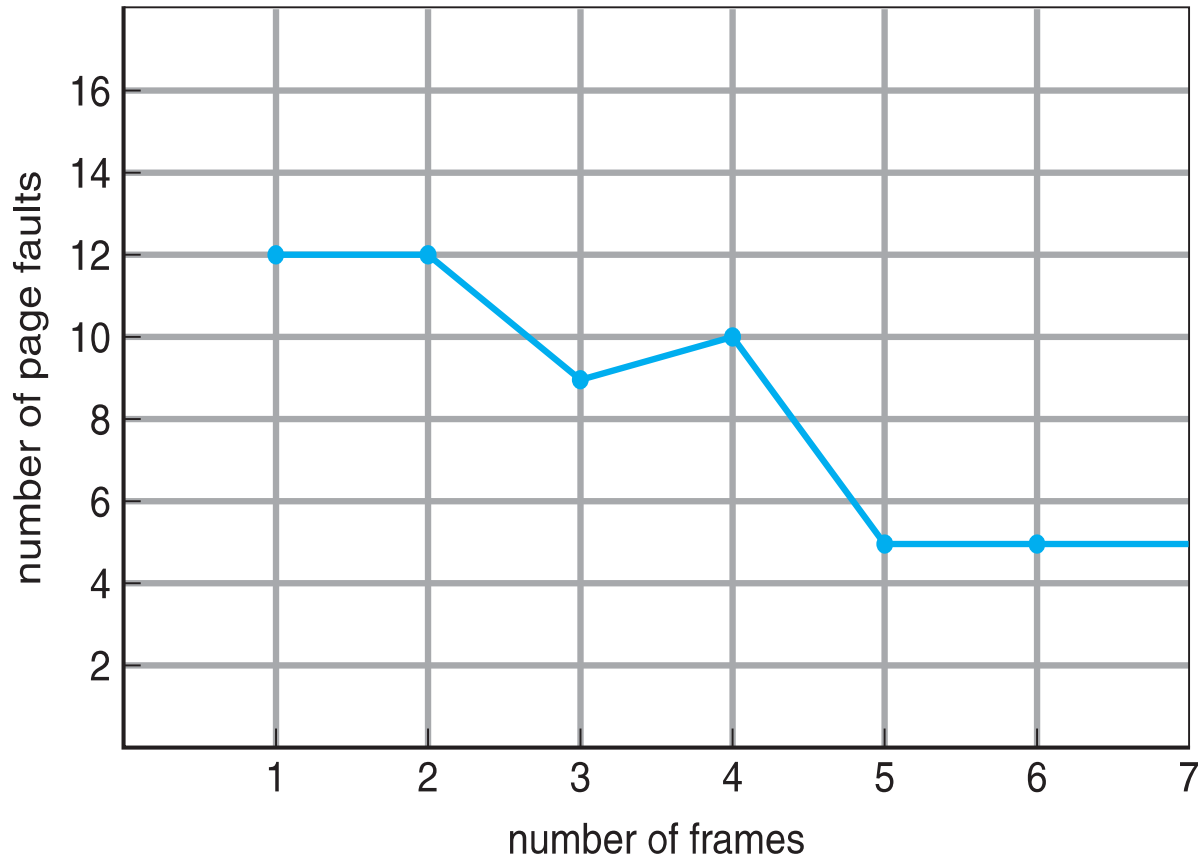
است



■ Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



FIFO Illustrating Belady's Anomaly



:Belady's Anomaly

انتظار داریم که به طور کلی هر وقت تعداد frame ها رو می بریم بالا تعداد page fault کمتر بشه اما اگر با همون Reference string صفحه قبلی بیایم تعداد page fault ها رو بشماریم با تعداد frame های متفاوت و روش هم FIFO رو اجرا بکنیم می بینیم یه جایی است که اگر تعداد frame ها از 3 به 4 تغییر بدیم تعداد page fault ها هم یکی بیشتر میشه به این اتفاق Belady's Anomaly می گیم که دوست نداریم این اتفاق بیوفته اینجا می بینیم FIFO خیلی بد عمل میکنه و حتی باعث میشه که گاهی اگر ما تعداد frame ها هم بیشتر بکنیم ولی با تعداد page fault بیشتری مواجه بشیم



Optimal Algorithm

- Replace page that will not be used for longest period of time
 - 9 is optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		2		2		2						7		
	0	0	0		0		4		0		0						0		
		1	1		3		3		3		1						1		

page frames



از لحاظ پرفرمنس خیلی روش قبلی خوب نیست و اینکه ما دوست داشتیم کمترین تعداد

page fault داشته باشیم الگوریتم قبلی اینو برای ما فراهم نمی کرد

اگر بخوایم یک الگوریتم Optimal داشته باشیم که منجر به کمترین تعداد page fault بشه این الگوریتم باید چه page رو جایگزین کنه؟

برای کمترین تعداد page fault باید ببینیم که کدوم page ها در آینده کمترین جایگزینی رو

خواهند داشت و اون page رو انتخاب بکنیم برای جایگزینی --< اینجوری تعداد page fault ها مینیمم میشه

توی این مثال:

مثلا اینجا وقتی که می خواد 2 رو وارد کنه نگاه می کنه توی reference string کدوم یکی از

این page های 7 , 0 , 1 دیرتر بهشون ارجاع شده که میشه 7 پس جای 7 میایم 2 رو می داریم

اینطوری page fault ما مینیمم میشه

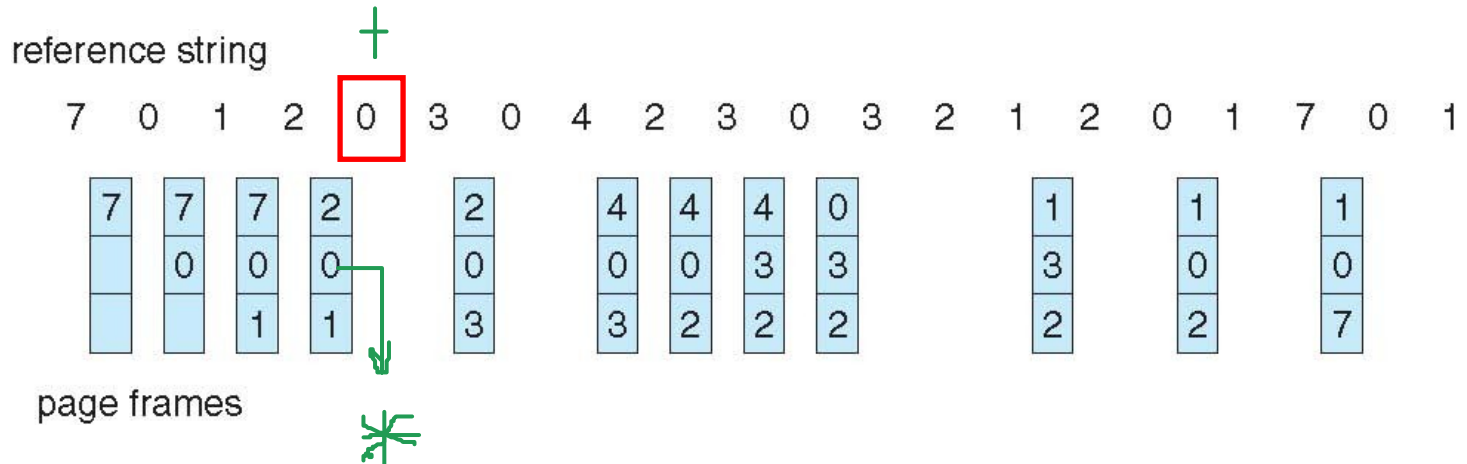
اما این الگوریتم مشکلاتی داره

ادامه صفحه بعدی...



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?



مشکلات:

پیاده سازی سخت است ما از کجا بدونیم کدام page دیرتر از page های دیگر بهش رفرنس می خوره ما اینجا برای مثال reference string آوردیم و می تونیم کل reference string رو ببینیم ولی در واقعیت براساس Demanding پیج رو میاریم توی حافظه و ما نمی دونیم در آینده چه page های Demanding میشن پس ما نمی تونیم اندازه گیری بکنیم که کدام page دیرتر از page های دیگر بهش رفرنس میخوره

پس این الگوریتم برخلاف الگوریتم FIFO که از لحاظ page fault وضعیت خوبی نداشت ولی برعکس FIFO که با اردر یک می تونست کار رو انجام بده اینجا اصلا نمی دونیم چطوری باید این کارو انجام بدیم یه اطلاعاتمون کافی نیست

برای همین کاری که می کنن اینه که یک تخمین هایی از الگوریتم Optimal رو استفاده می کنن یکی از این تخمین ها LRU است --> که بسیاری از جاها برای جایگزینی از این الگوریتم استفاده میشه

الگوریتم LRU: می خوایم ببینیم کدام پیج Recently کمتر استفاده شده --> کلمه Recently مهمه اینجا که در کل میگه که کدام page اخیرا کمتر استفاده شده

مثلا: توی این reference string این 1 0 7 مثل قبل پر میشه و الان اینجا که میخوایم 2 رو جایگزین بکنیم با یکی از این page ها نگاه میکنیم کدام page که الان توی حافظه است اخیرا بهش ریکوست نشده یا توی زمان گذشته که نگاه کنیم به اون عدهای اولی اخیرا ارجاع داده نشده --> پس 1 0 که اخیرا ارجاع شده پس اینجا باید 7 رو انتخاب بکنیم برای جایگزینی

اینجا برای این الگوریتم تعداد کل page fault ها شده 12 تا که کمتر از الگوریتم FIFO شده --> این الگوریتم از حالت Optimal بدتر است و از حالت FIFO بهتر است
توی حالت Optimal کل page fault ها شده بود 9 تا



LRU Algorithm (Cont.)

- Counter implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to find smallest value
 - ▶ Search through table needed
- Stack implementation
 - Keep a stack of page numbers in a double link form:
 - Page referenced:
 - ▶ move it to the top
 - ▶ requires 6 pointers to be changed (for 6 virtual page)
 - But each update more expensive
 - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly



چجوری این الگوریتم رو پیاده سازی بکنیم؟

1- یکی از کارها اینه که ما یک کانتری رو قرار بدیم ینی هر page یک کانتری داشته باشه این کانتر شمارنده نیست بلکه یک time page است ینی مثلا فرض کنیم کلاک هر موقعی که به page رفرنس می خوره زمانی رو اینجا برای اون page مون ذخیره میکنیم مثلا توی صفحه قبل اونجا که این page ها رفرنس خورده اونجا که 7 رفرنس خورد تایمر رو نگاه می کنیم و یک جا براش تایم رو ذخیره میکنیم و همینطور برای بقیشون که توی حافظه قرار میگیرن مثلا 1 و 0 و به همین ترتیب ینی هر کدومشون یک تایم دارند و زمانی که می خوایم جایگزینی رو انجام بدیم نگاه میکنیم که تایم کدوم از همه کوچکتر است و این نشون میده که کی اخیرا بهش رفرنس خورده مثلا توی صفحه قبلی که دوباره صفر + رفرنس می خوره تایم این صفر * رو که الان توی حافظه فیزیکی موجود است الان باید اپدیت بشه با تایم فعلی این + پس هر بار که هر کدوم از page وارد مموری میشن یا رفرنس می خورن تایمشون اپدیت میشه پس اگر این کارو همیشه بکنیم موقعی که میخوایم LRU رو انتخاب بکنیم کافیه اون page توی مموری انتخاب بشه که تایم کوچکتری داره و این همون LRU رو نشون میده

مشکل این حالت چیه که از یک تایم استفاده بکنیم؟ اینه که موقعی که میخوایم page رو جایگزین کنیم باید بگردیم که کدوم page تایم کمتری داره ینی به یک سرچ نیاز داریم موقع جایگزین کردن پس به اندازه اردری که سرچ کردن داره ما باید اینجا تاخیر داشته باشیم و over head اش به اندازه O سرچ کردن است

وقتی که میخوایم اپدیت بکنیم تایم ها رو کافیه همون page که الان رفرنس خورده رو یک فیلد تایم براش در نظر بگیریم و اپدیتش بکنیم پس اینجا این روش اول مسئله اش این قسمت سرچش میشه که شاید اردرش برای ما مناسب نباشه و ما دوست داشته باشیم که over head کمتری داشته باشیم

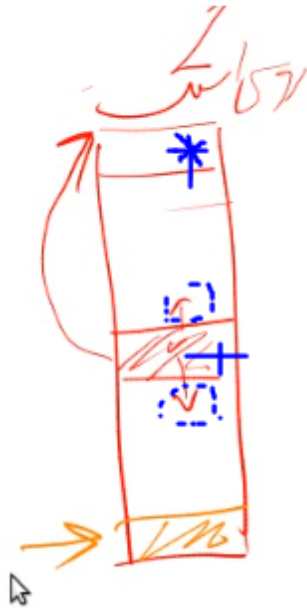
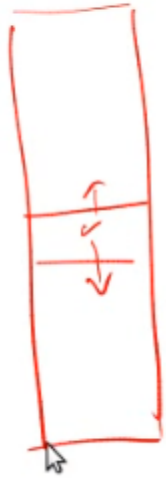
پس توی روش بعدی این سعی میکنیم که این over head که برای قسمت انتخاب page داریم رو کمتر بکنیم این روش از استک استفاده میکنه و این استک رو دو طرفه نگهداری می کنه ینی همه نودهاش دوطرفه هستن که هم سر استک رو داشته باشه و هم ته استک رو --> عکس توی این استک page هایی که الان توی مموری قرار دارن رو نگه می داریم هر وقت یک page رفرنس می خوره اون رو میاریمش روی استک قرار میدیم با این فرض اگر ما بخوایم یک page رو برای جایگزین کردن کدوم رو باید انتخاب بکنیم؟ مسلما اونایی که روی استک قرار دارن اخیرا بهشون رفرنس خورده و اونی که ته استک افتاده زمان زیادی که بهش رفرنس نخورده که نیومده بالا پس این page که ته استک هست رو می تونیم انتخاب بکنیم برای جایگزینی کردن

حالا over head کارهایی که باید انجام بدیم به چه صورت است؟ موقعی که میخوایم یک page رو انتخاب بکنیم که برای جایگزینی کافیه که ته استک رو برداریم و وقتی که لینکدلیست های دو طرفه داشته باشیم و یک پوینتر هم ته استک داریم اینجا کافیه با اردر یک پیداش بکنیم و میتونیم جایگزینش بکنیم این page رو ولی مشکل کجاست؟ موقعی که یک page بهش رفرنس میخوره و میخواد روی این استک اپدیتش بکنیم که باید جابه جاش کنیم اونجا نیازه که پوینترهای زیادی جابه جا بشه --> ینی این page که الان داریم + از اینجا برش داریم پس لینک های دوطرفه، دو طرفش باید تغییر بکنه و بعد این page که میاد روی استک باید پوینترهای * هم اپدیت بکنیم پس به طور کلی 6 تا پوینتر اینجا باید اپدیت بشه که اینجا یک مقداری ما over head داریم ولی به هر حال به طور کلی وضعیت یک مقدار بهتر شده نسبت به حالت قبلی که از یک تایم استمپ استفاده می کردیم ینی اینجا یک سرچ کامل نیاز نداریم موقع جایگزین کردن

توی روش قبلی این جایگزین کردن توی زمان انلاین محسوب میشد و برامون مهمه که اون زمان کمی ببره ولی توی این روش که page ها رو روی این استک اپدیتشون بکنیم این میتونه جز زمان افلاینمون باشه ینی اون لحظه ای که داریم page روی این استک جاش رو تغییر میدیم اون لحظه نباید مثلا منتظر باشیم میتونیم بقیه کارها رو انجام بدیم و جای این رو هم روی استک درست بکنیم ولی وقتایی که میخوایم page رو برای جایگزین کردن انتخاب بکنیم باید همون لحظه جواب کامل رو پیدا بکنیم که اون page کدام است و اینجا منتظره این جایگزین کردن است پس میگیم این روی زمان انلاینمون است و این خوبه که توی زمان انلاین over head کمتری داشته باشیم که تاخیر کمتری حس کنیم

به طور کلی این روش ها که از LRU گرفته شدند و LRU هم از اون روش Optimal هم گرفته شد این ها مشکل Belady's Anomaly هم ندارند و برای ما مناسب هستن برخلاف FIFO که دچار این Belady's Anomaly میشد

1



3

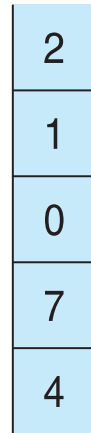




Use Of A Stack to Record Most Recent Page References

reference string

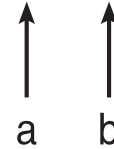
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



مثال:



LRU Approximation Algorithms

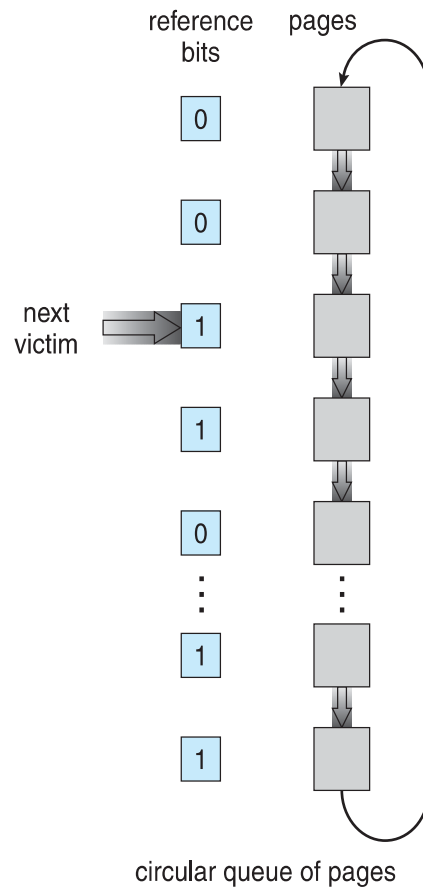
- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - ▶ We do not know the order, however
- **Second-chance algorithm**
 - Generally FIFO, plus hardware-provided reference bit
 - **Clock** replacement
 - If page to be replaced has
 - ▶ Reference bit = 0 -> replace it
 - ▶ reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules



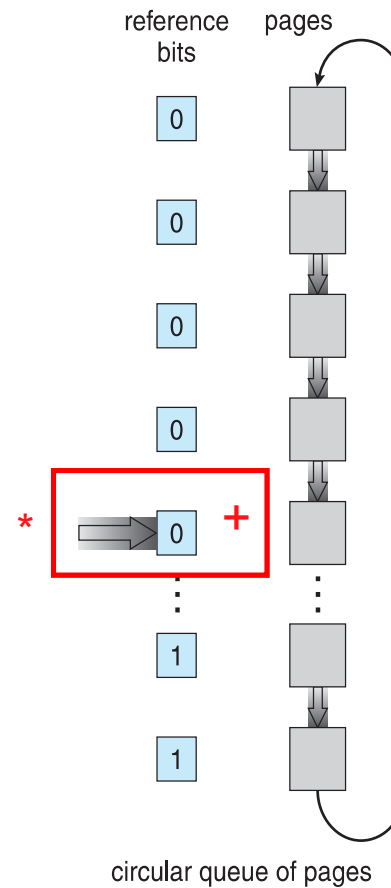
این دوتا روشی که تا اینجا گفتیم دقیقا LRU پیاده می کردن چه روش تایم و چه روش استک داره
 دقیقا LRU رو انتخاب میکنه ولی اگر ما یک مقدار بخوایم بازم over head پیدا کردن اون
 page رو کمتر بکنیم یا کلا عملیاتی که باید انجام بشه برای اپدیت page ها میتونیم
 Approximation های LRU استفاده بکنیم ینی به دقت LRU نیستن ولی میشه گفت سعی میکنن
 page رو انتخاب بکنن که اخیرا نسبت به بقیه کمتر انتخاب شده باشه
 یکی از این روش ها روش رفرنس بیت است: این روش میاد برای هر کدوم از page هایی که توی
 حافظه قرار داره یک رفرنس بیت در نظر میگیره و اول کار رفرنس بیت همه رو صفر میگیره
 حالا هر وقت یک page بهش رفرنس خورد این رفرنس بیتش رو میکنه یک و حالا اگر خواست
 یک page رو جایگزین بکنه از این page هایی که توی حافظه است میاد اونایی که صفرن رو
 انتخاب میکنه چون اونایی که رفرنس خوردن رو یک کردیم و اونایی که رفرنس نخوردن و صفرن
 ینی اخیرا کمتر رفرنس خوردن پس کافیه یکی از این صفرها رو انتخاب بکنیم برای جایگزین کردن
 اما این دقیق نیست --> یکی از روش های پیاده سازی این ایده الگوریتم Second-chance
 است



Second-Chance (clock) Page-Replacement Algorithm



(a)



(b)



الگوریتم Second-chance به چه صورت کار میکنه؟

فرض میکنیم این طوسی ها page هایی هستن که توی حافظه قرار دارن و اینو داریم به صورت یک لینک circular نگهداریم میکنیم

و همیشه یک پوینتری اینجا داریم که با این پوینتر میخوایم سرچ بکنیم که کدوم page رو جایگزینش بکنیم

هرموقعی هم که یک page رفرنس میخوره رفرنس بیتش رو یک می کنیم و مقدار دهی اولیه همه این بیت ها اول کار صفر است

اگر ما خواستیم یک page رو انتخاب بکنیم برای جایگزین کردن کافیه از این پوینتر next

victim شروع بکنیم و بریم جلو و برسیم به page که رفرنس بیتش صفر است و اونو انتخاب

بکنیم برای جایگزین کردن ولی در این بین که داریم این پوینتر رو می بریم جلو اگر page هایی

دیدیم که رفرنس بیتشون یک بودن کاری که میکنیم این است که یک شانس دیگه ای به این ها میدیم

و اینارو تبدیلس میکنیم به صفر توی این جلو رفتن --< توی شکل b نشون داده

page --< + رو جایگزین میکنه و چه اتفاقی برای اون دوتا قبلی افتاد؟ ینی انگار یک شانس دیگه

ای بهشون دادیم اگر توی این فاصله این پوینتر * داره می ره جلو و به صورت حلقه ای برمیگرده تا

اینکه برسه به این دوتا اگر توی این فاصله این دوتا رفرنس خوردن صفرشون به یک تبدیل میشه و

دوباره که پوینتر اومد اینجا برای جایگزین کردن انتخاب نمی شن اما اگر توی این فاصله رفرنس

نخوردن اگر پوینتر رسید به اینجا انتخاب میشن برای جایگزین شدن

این روش نسبت به اون روش استک یک مقداری پیاده سازیش ساده تر است کافیه که این پوینتر رو

داشته باشیم و با این بریم جلو و به اولین صفری که رسیدیم جایگزین میشه ولی خب دقیق نیست این

روش ینی دقیقا اون LRU انتخاب نمی کنه برخلاف استک که این کار رو انجام میداد



Allocation of Frames

- Each process needs ***minimum*** number of frames
 - The minimum number of frames is defined by the computer architecture
 - we must have enough frames to hold all the different pages that any single instruction can reference.
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations



یه تعدادی فریم اختصاص بدیم به هر پروسس
یکیش اینه که ثابت بگیریم

این مینیمم تعداد فریم که از اون ابتدا به پروسس داده میشه شاید سخت افزار بتونه مشخص بکنه
ماکزیمم می تونه با توجه به حجم حافظه فیزیکی ما انتخاب بشه



Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
 - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
 - Dynamic as degree of multiprogramming, process sizes change

— s_i = size of process p_i

— $S = \sum s_i$

— m = total number of frames

— a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$



اینجا فریم رو ثابت گرفته --> تعداد فریم تقسیم بر تعداد پروسس

فرموله می‌گه به هر پروسسی به نسبت خودش فریم بدیم

به هر پروسسی به نسبت خودش frame اختصاص بدیم

لازمش اینه که بدونیم یک سیستم چندتا پروسس داره و این امکان پذیر نیست که از اون ابتدا چندتا پروسس داره



Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
 - select for replacement one of its frames (**local replacement**)
 - select for replacement a frame from a process with lower priority number (**global replacement**)



توی حالتی که میخوایم به نسبت هر پروسی بهش فریم بدیم:
یکش حالتش اینه که: با توجه به Priority اش بهش frame بدیم
از پروسی که اولویتش از بقیه پروسس ها کمتره بیایم page بگیریم و جایگزین بکنیم



Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
 - But then process execution time can vary greatly
 - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
 - More consistent per-process performance
 - But possibly underutilized memory



اگر توی جایگزینی همیشه page های خود اون پروسس که نیاز به page جدید داره میگیریم لوکالی است و اگر نه مجاز باشیم که از page های پروسس های دیگه رو جایگزین بکنیم میگیریم گلوبال که این گلوبال داره توی سیستم استفاده میشه مزیت:

گلوبال: دستمون ازادتر است و احتمال به page fault های کمتری می خوریم چون اگر از خود اون پروسس page بگیریم ممکنه توی اجراهای بعدی باز به اون page نیاز داشته باشیم پس توی لوکالی زیاد ممکنه page fault داشته باشیم

مشکل گلوبال: رفتار اجرای پروسس های مختلف متفاوت توی استفاده از page ها و وابستگی به این داره که چه پروسس های در حال اجرا هستن --> منجر به این میشه که توی حالت گلوبال اگه page fault اتفاق می افته ندونیم که چقدر زمان نیازه برای جایگزینی اون page و پیدا کردنش و اجرای این پروسس ممکنه با توجه به این که پروسس های دیگه چی هستن ممکنه زمان های متفاوتی بگیره اما توی لوکال داریم از page های خودش استفاده میکنیم که خود این پروسس یک رفتاری هم داره که برای همشون مشخصه پس فرمرنس این مشخصه ینی یا خیلی over head زیادی داریم یا کم پس متغییر نخواهد نشد ینی در کل over head مون ثابتته اینجا



Reclaiming Pages

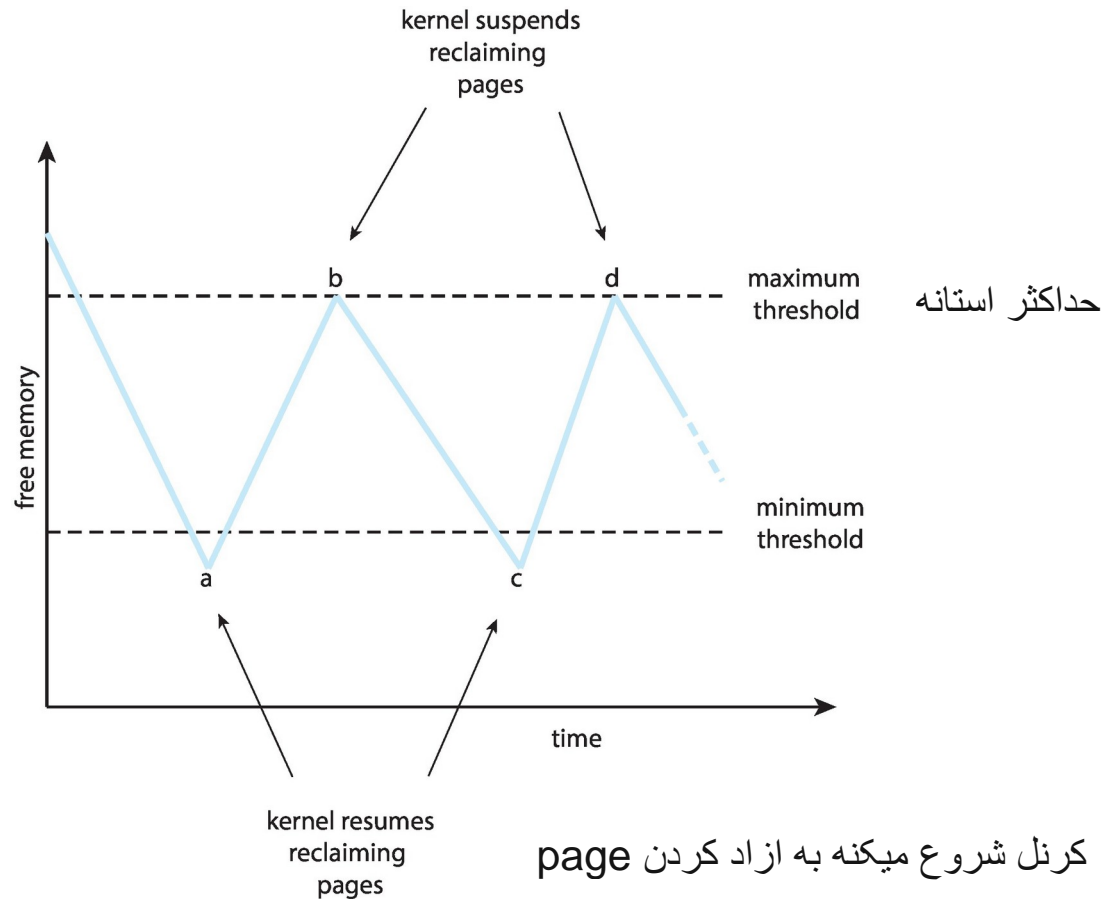
- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the **free-frame list**, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.



ینی کلا ما اجازه ندیم به حالتی برسیم که کلا هیچ پیج ازادی نداشته باشیم که مجبور به جایگزینی بشیم توی این حالت سیستم میاد یه تعداد فریم ازاد رو نگه میداره که اگر یک پروسسی نیاز به پیج بیشتری داشت بتونه از اون لیست ازاد فریم بگیره
ایده این کار میشه صفحه بعدی



Reclaiming Pages Example



میگه که اگر تعداد فریم های ازادمون رسید به این مینیمم یا کمتر شد اون وقت شروع بکنیم به ازاد کردن یکسری فریم های دیگه --> توی این لحظه که داریم فریم ازاد می کنیم ینی داریم یکسری پیج رو جایگزینی میکنیم ینی یکسری پیج داریم از پروسس های مختلف می گیریم به ماکزیمم که رسید فریم ازاد کردن رو رها میکنه و اجازه میده که سیستم با همین فریم های ازاد کار کنه و پروسس های دیگه میاد این فریم ها رو می گیرن و باز همین کار...



Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace existing frame
 - But quickly need replaced frame back
 - This leads to:
 - ▶ Low CPU utilization
 - ▶ Operating system thinking that it needs to increase the degree of multiprogramming
 - ▶ Another process added to the system
- **Thrashing** \equiv a process is busy swapping pages in and out

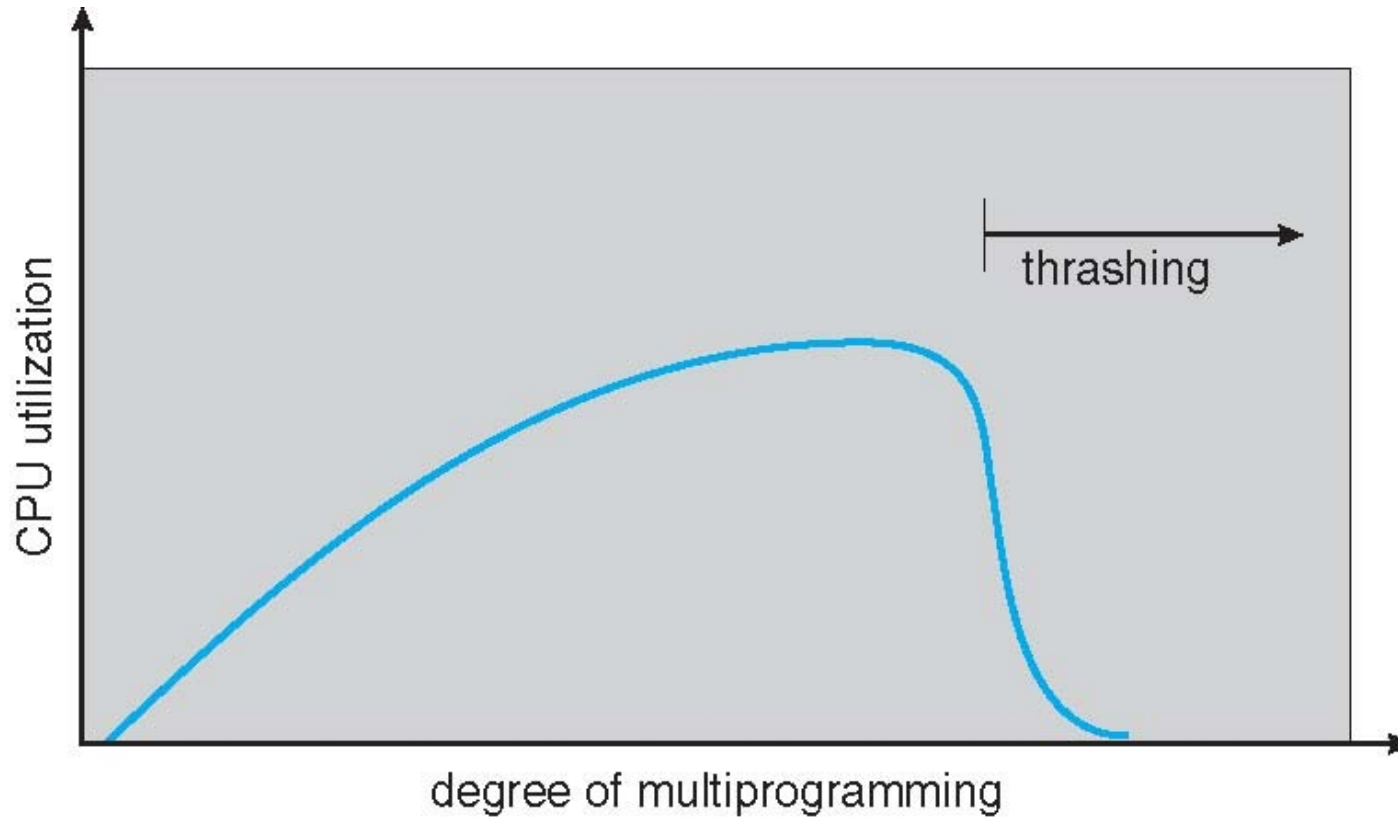


-
اگر یک پروسسی پیج هاش به اندازه کافی نباشه و page fault بخوره بهش برای اینکه page fault اش بره بالا اونوقت باعث میشه ک دائما در حال جایگزینی باشه مخصوصا حالتی که از خودش داره استفاده می کنه ینی لوکالی باعث میشه که استفاده از cpu میاد پایین و توی سیستم عامل ها یک ترفندی که می دارن اینه که به بهره گیری از cpu نگا میکنه و اگر پایین بود فک میکنه که تعداد پروسس هایی که پایین هستن پس میتونه پروسس دیگه هم جواب بده پس درجه مالتی پروگرامینگ رو می بره بالا

اگر کاهش cpu بخاطر این باشه که یک پروسسی دچار جایگزینی پیج و page fault شده در واقع سیستم وقتی پروسس جدیدی اضافه میکنه باعث میشه وضع بدتر بشه و اون پروسسی که دچار page fault شده بود وضعیتش بدتر میشه و میگیریم دچار Thrashing و اجراش کند میشه و بیشتر مواقع در حال جایگزین کردن پیج است و باید کاری بکنیم که Thrashing اتفاق نیوفته



Thrashing (Cont.)





Demand Paging and Thrashing

- Why does demand paging work?

Locality model

- Process migrates from one locality to another
- Localities may overlap

- Why does thrashing occur?

Σ size of locality > total memory size

- Limit effects by using local or priority page replacement



کنترل Thrashing چجوری انجام میشه:

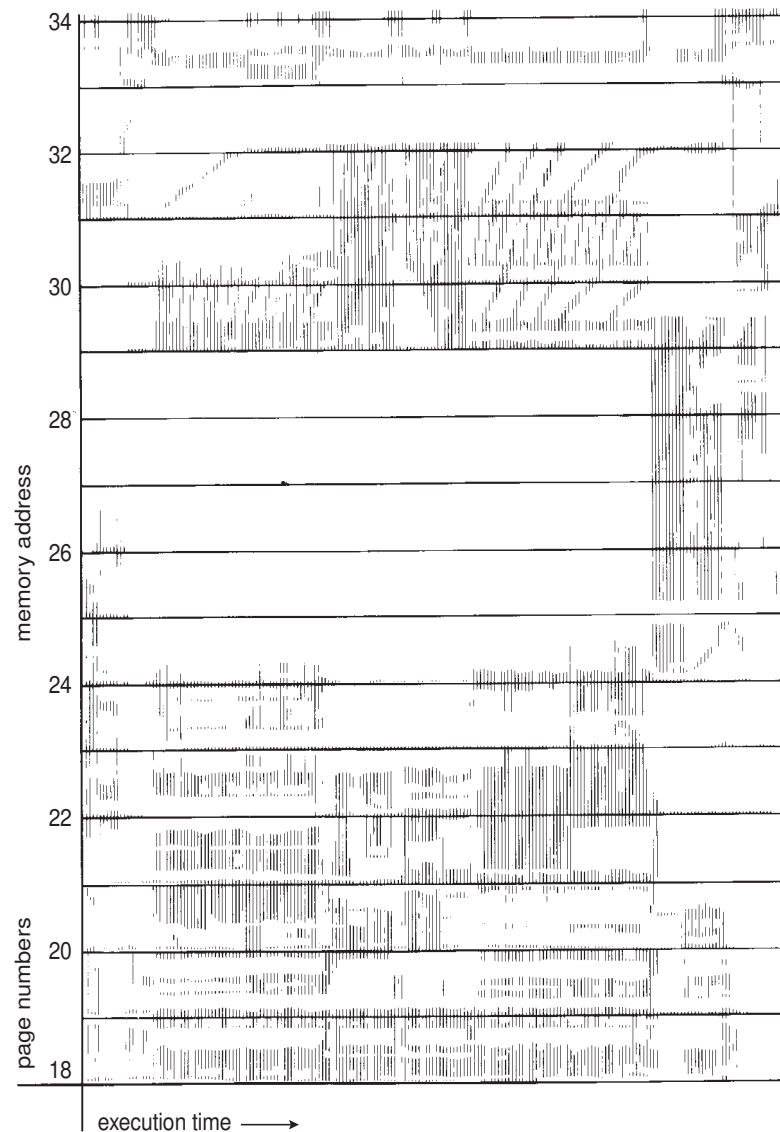
Locality model: صفحه بعدی...

اجرای پروسس ها حالت Locality model رو داره --> توی هر بازه زمانی داره با یکسری پیج های مشخصی کار میکنه پروسس

Locality model: ینی توی هر بازه زمانی پروسس با یک تعداد پیج مشخصی کار میکنه نه اینکه از کل پیج های پروسس استفاده بکنه --> این ویژگی خوبی است --> احتمال page fault کم میشه پس اگر مجموع اون size of locality از حافظه فیزیکی ما کمتر باشه مشکلی نداریم و دچار Thrashing نمیشیم ولی اگر مجموع اون از اندازه مجموع سایز مموری بیشتر باشه دچار Thrashing میشیم



Locality In A Memory-Reference Pattern





Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
 - Kernel requests memory for structures of varying sizes
 - ▶ As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation
 - Some kernel memory needs to be contiguous
 - ▶ i.e., for device I/O



End of Chapter 10

