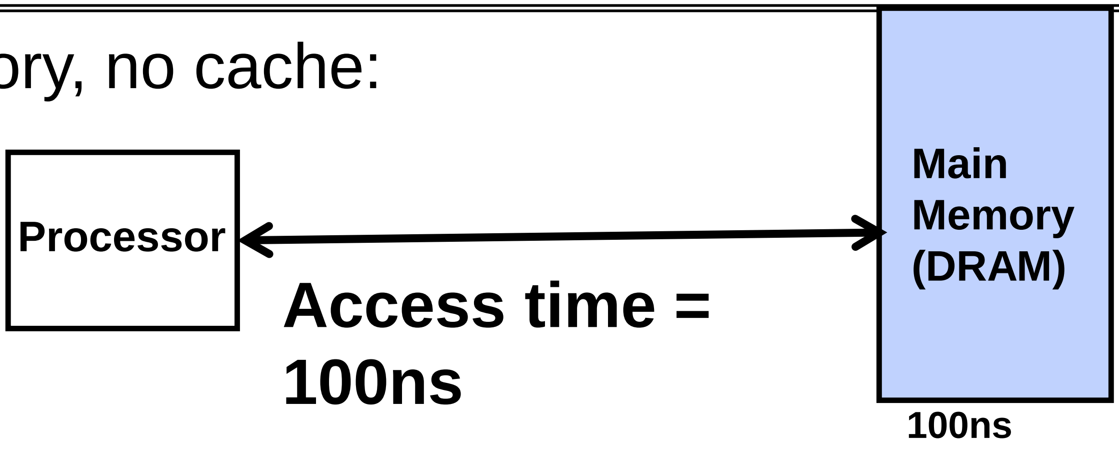
CSE150
Operating Systems
Lecture 16

Page Allocation and Replacement

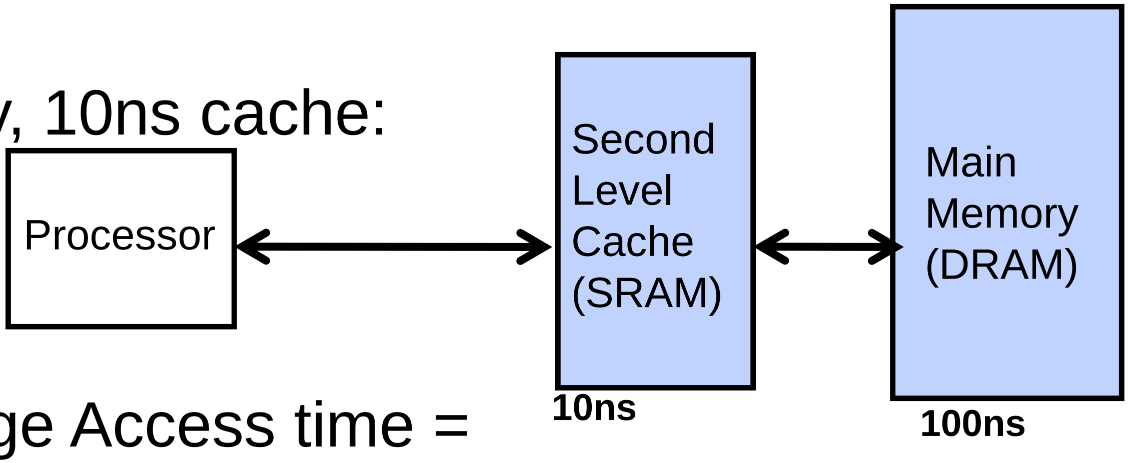
Review: Cache Example

- Data in memory, no cache:



Access time = 100ns

- Data in memory, 10ns cache:

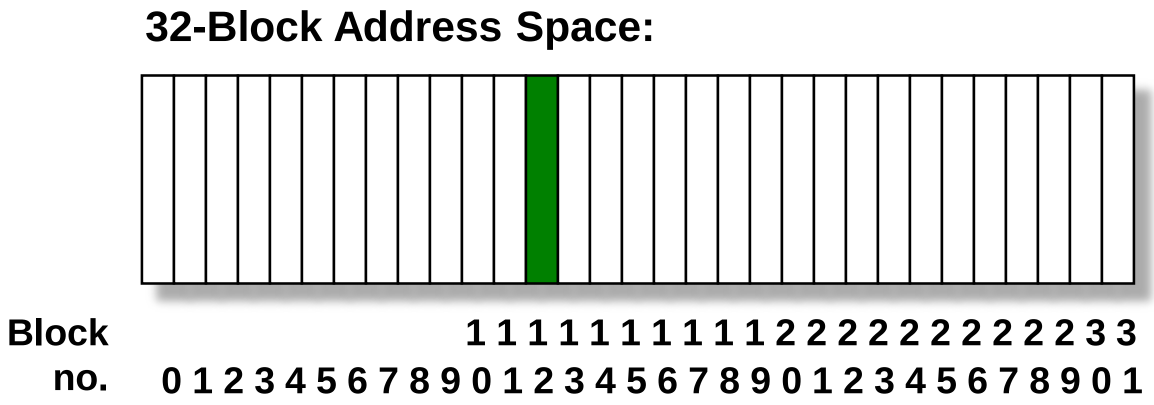


Average Access time =
(Hit Rate x HitTime) + (Miss Rate x MissTime)

- HitRate + MissRate = 1

Review: Where does a Block Get Placed in a Cache?

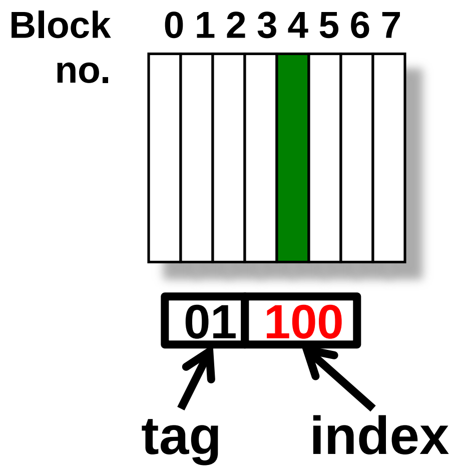
- Example: Block 12 placed in 8 block cache

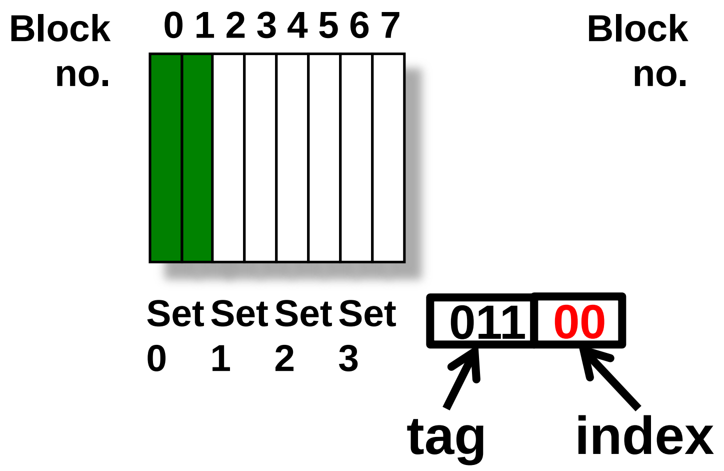


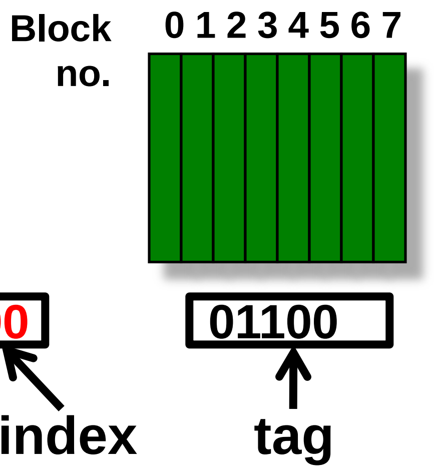
Direct mapped:
block 12 (01100) can go only into block 4 (12 mod 8)

Set associative:
block 12 can go anywhere in set 0

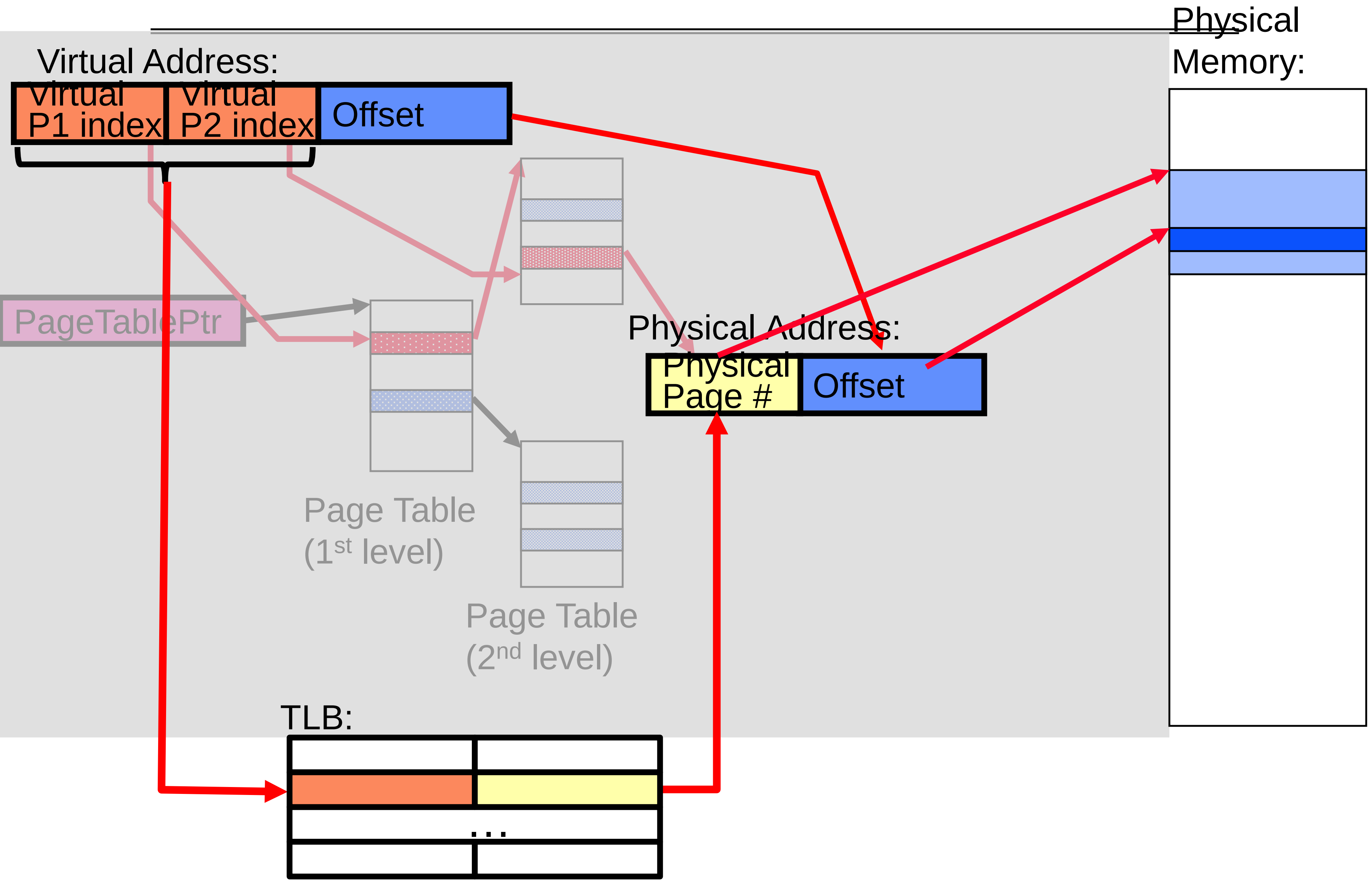
Fully associative:
block 12 can go anywhere



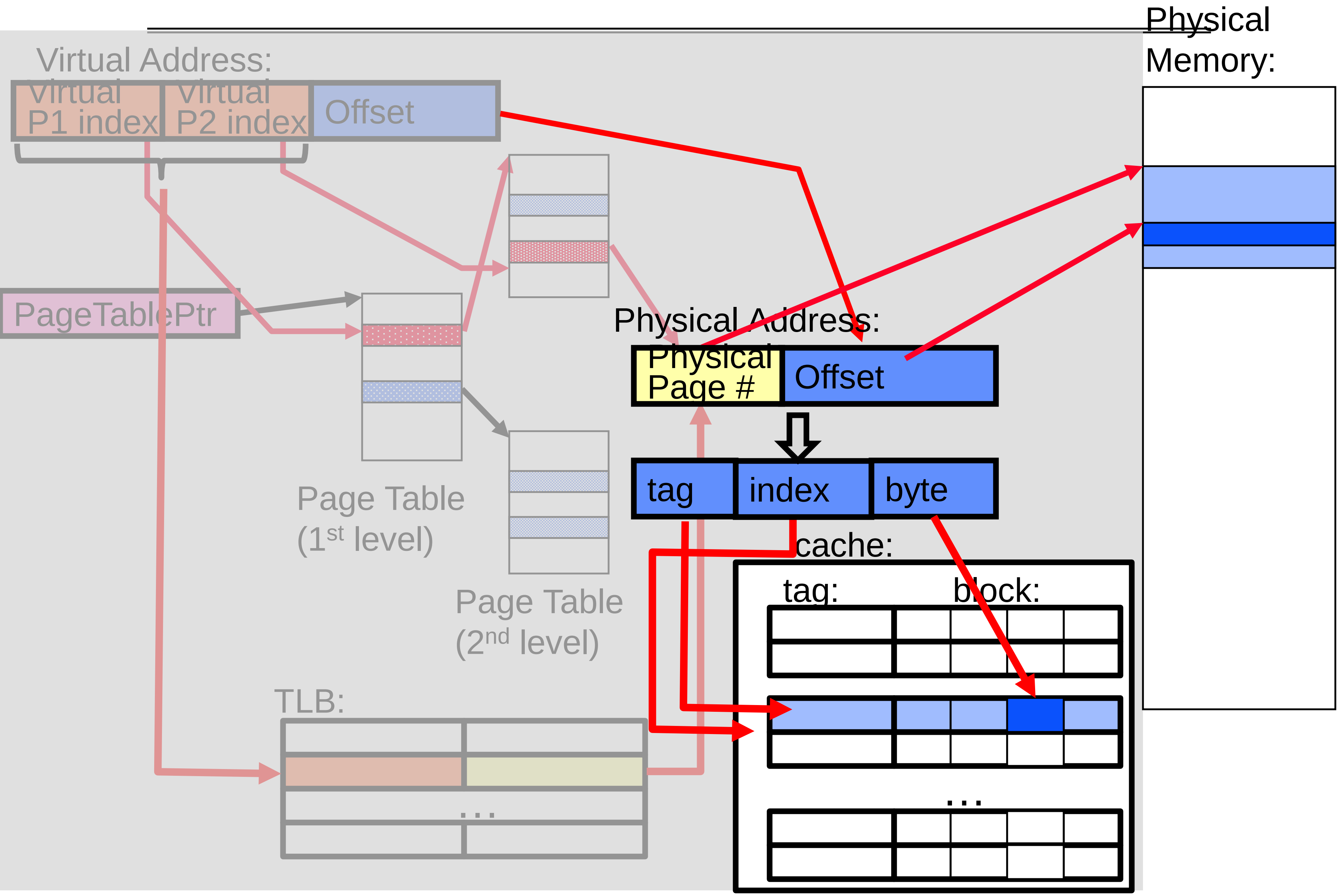




Review: Translation Lookaside Buffer



Review: Cache



What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - » If PTE valid, fills TLB and returns from fault
    - » If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!

- Options?
  - Invalidate entire TLB (flush): simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware

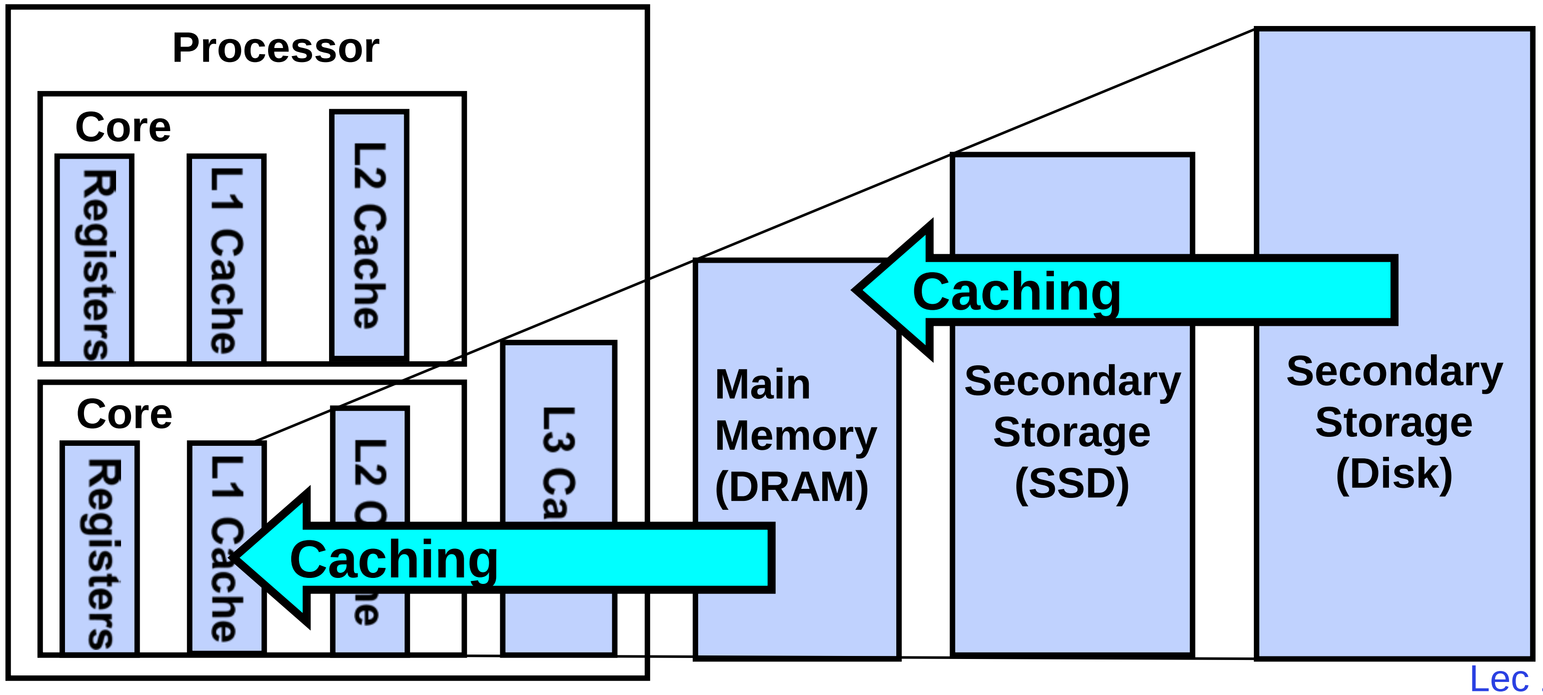
Today

- Concept of paging to disk
- TLB Faults
- Page Replacement Policies
  – FIFO, MIN, Random, LRU

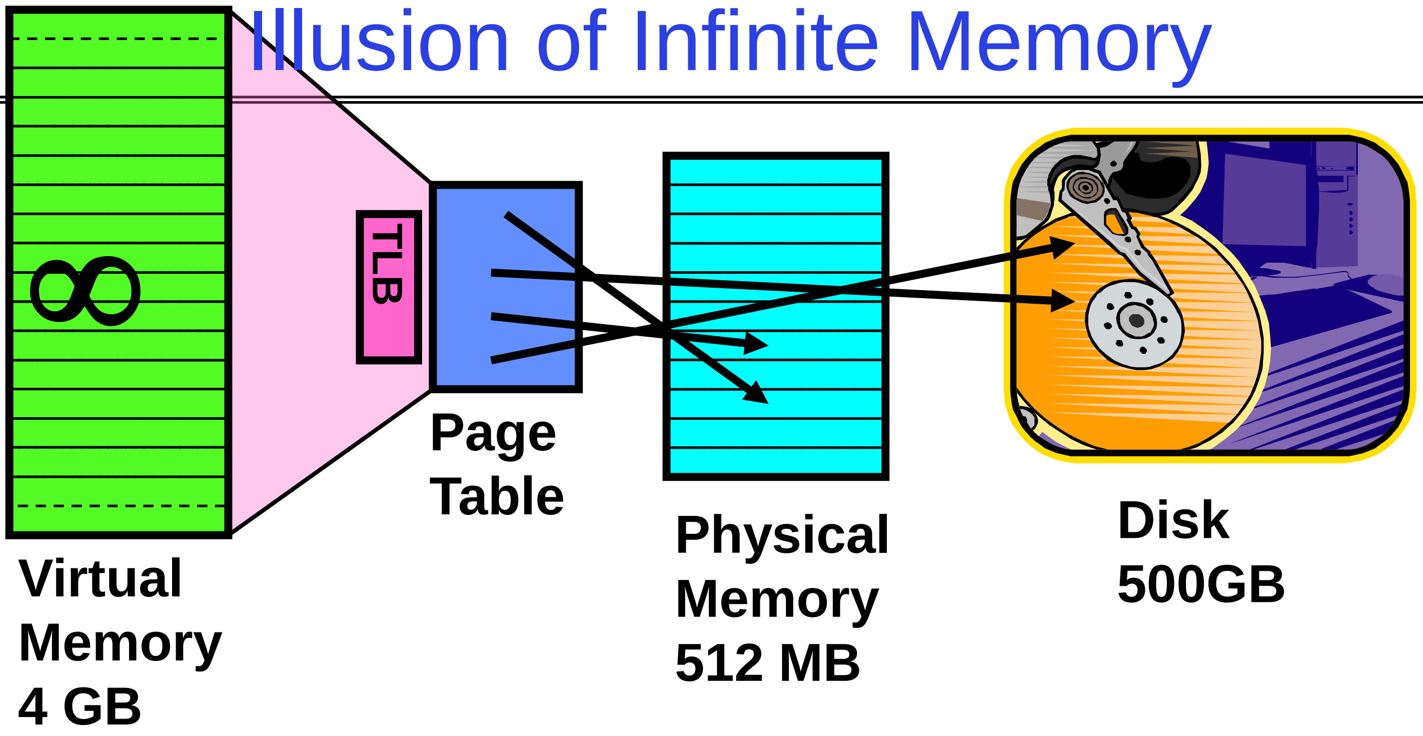
Slides adapted from CS162 Course Material at UC Berkeley

Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as cache for disk/SSD



Illusion of Infinite Memory



- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - » Performance issue, not correctness issue

Demand Paging is Caching

- Since Demand Paging is Caching, we must ask:

Question	Choice
What is the block size?	
What is the organization of this cache (i.e., direct-mapped, set-associative, fully-associative)?	
How do we find a page in the cache?	
What is page replacement policy? (i.e., LRU, Random, …)	
What happens on a miss?	
What happens on a write? (i.e., write-through, write-back)	

Recall: What is in a PTE?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

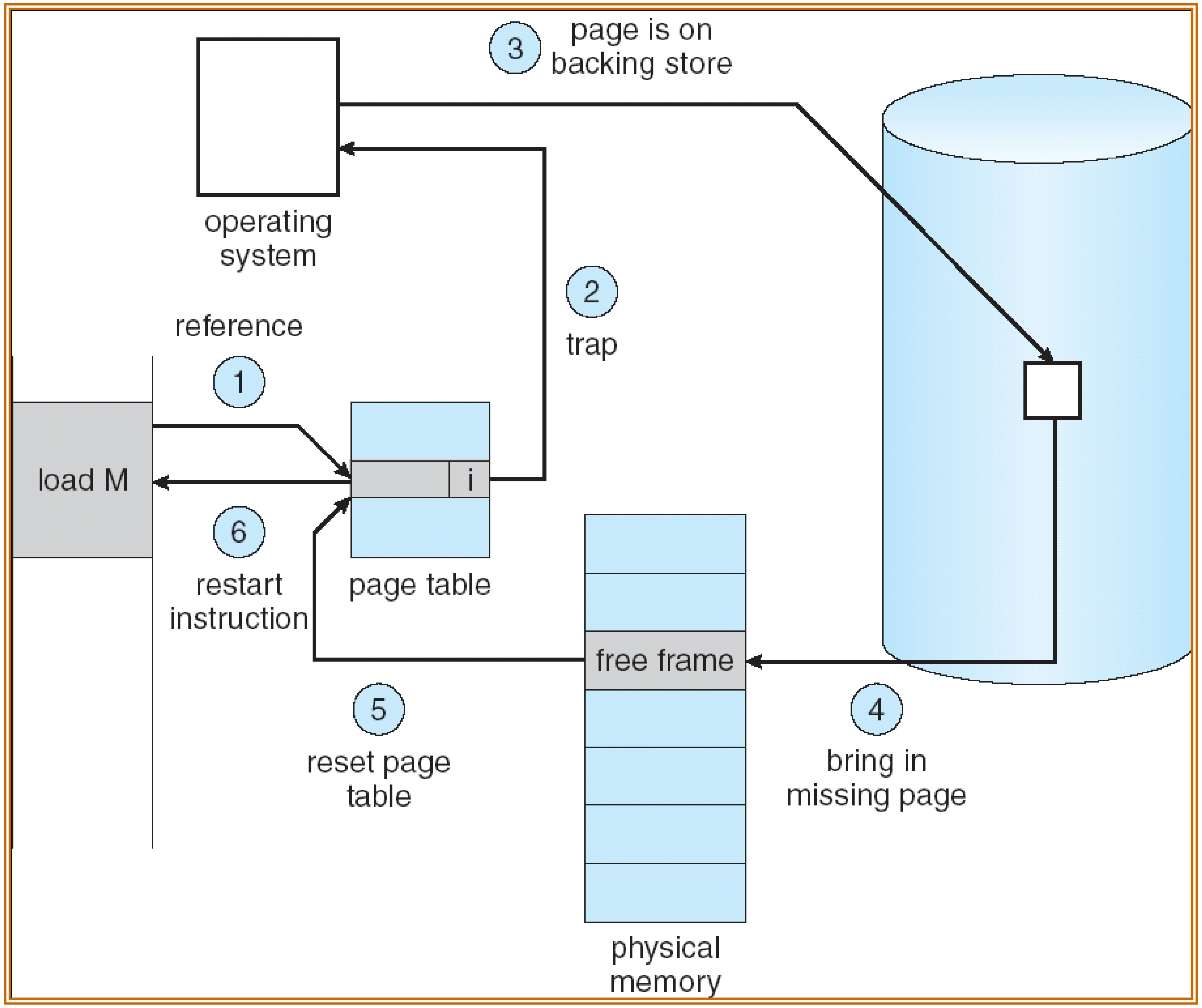
Page Frame Number (Physical Page Number)	**Free (OS)**	**0**	**L**	**D**	**A**	**PCD**	**PWT**	**U**	**W**	**P**
31-12	**11-9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

P:	Present (same as “valid” bit in other architectures)
W:	Writeable
U:	User accessible
PWT:	Page write transparent: external cache write-through
PCD:	Page cache disabled (page cannot be cached)
A:	Accessed: page has been accessed recently
D:	Dirty (PTE only): page has been modified recently
L:	L=1$\Rightarrow$4MB page (directory only). Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid $\Rightarrow$ Page in memory, PTE points at physical page
  - Not Valid $\Rightarrow$ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE to be invalid
    - » Load new page into memory from disk
    - » Update page table entry
    - » Continue thread from original faulting location
  - While pulling pages off disk for one process, OS runs another process from ready queue

Steps in Handling a Page Fault



Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

$$\begin{aligned} \text{EAT} &= (1 - p) \times 200\text{ns} + p \times 8 \text{ ms} \\ &= (1 - p) \times 200\text{ns} + p \times 8{,}000{,}000\text{ns} \\ &= 200\text{ns} + p \times 7{,}999{,}800\text{ns} \end{aligned}$$

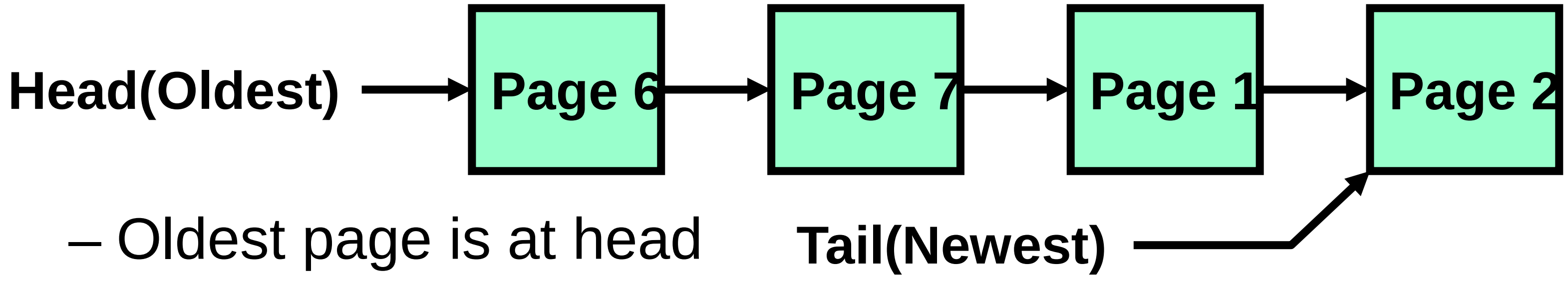
- If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - EAT < 200ns x 1.1 $\Rightarrow$ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400,000 !

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Unpredictable

Replacement Policies (Con't)

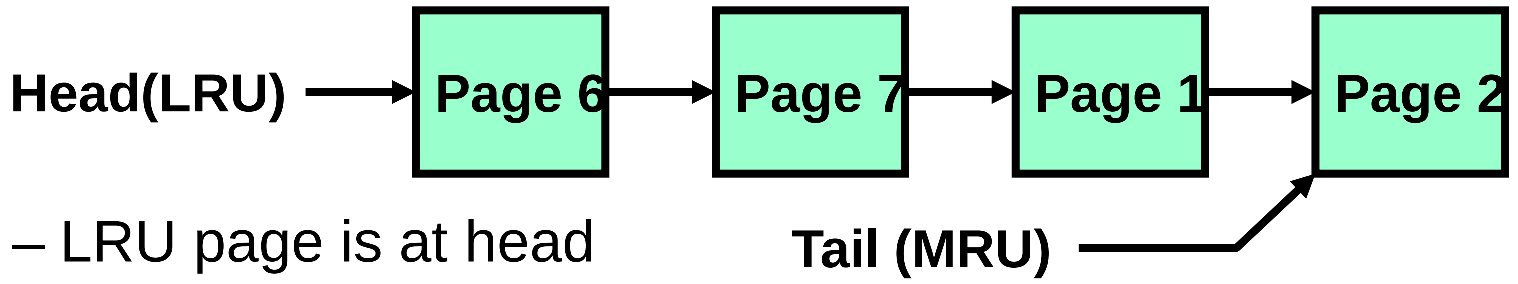
- FIFO:
  - Replace page that has been in for the longest time.
  - Be "fair" to pages and give them equal time.
  - Bad idea because page use is not even. We want to give more time to heavily used pages.
- How to implement FIFO? It's a queue (can use a linked list)



  - Oldest page is at head
  - When a page is brought in, add it to tail.
  - Eject head if list longer than capacity

Replacement Policies (Con't)

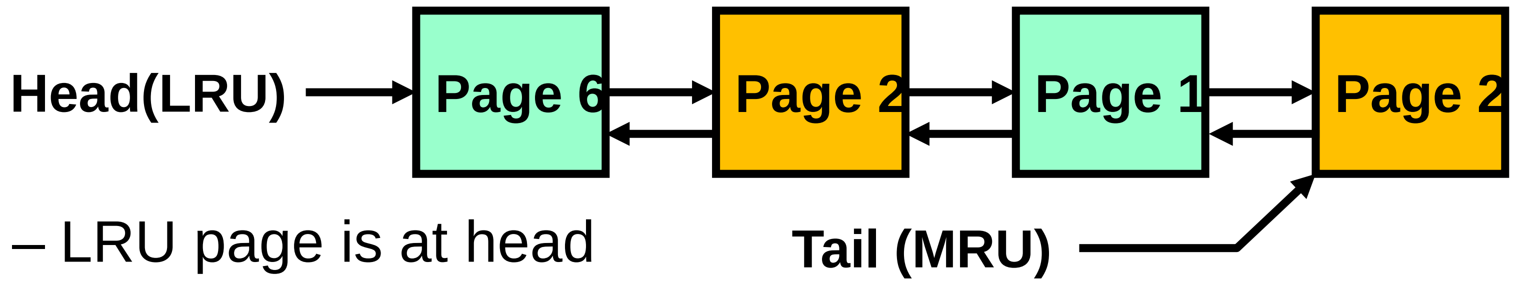
- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list?



  - LRU page is at head
  - When a page is used for the first time, add it to tail.
  - Eject head if list longer than capacity

Replacement Policies (Con't)

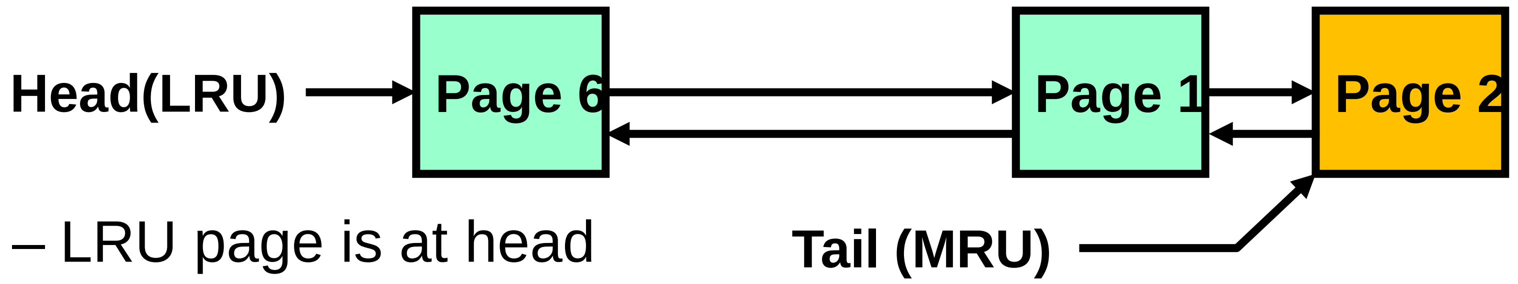
- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- Different if we access a page that is already loaded:



  - LRU page is at head
  - When a page is used again, **remove from list**, add it to tail.
  - Eject head if list longer than capacity

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- Different if we access a page that is already loaded:



  - LRU page is at head
  - When a page is used again, **remove from list**, add it to tail.
  - Eject head if list longer than capacity
- Problems with this scheme for paging? Too Expensive
  - Updates are happening on page **use**, not just swapping
  - List structure requires extra pointers compared to FIFO, more updates
- In practice, people approximate LRU.

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- Suppose we have the same reference stream:
  – A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  – MIN: 5 faults
  – Look for page not referenced farthest in future.
- What will LRU do?
  – Same decisions as MIN here, but won't always be true!

When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  – Every reference is a page fault!
- MIN Does much better:

Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A									B		
2		B					C					
3			C	D								

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady's anomaly)

Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

Summary

- Demand paging
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past