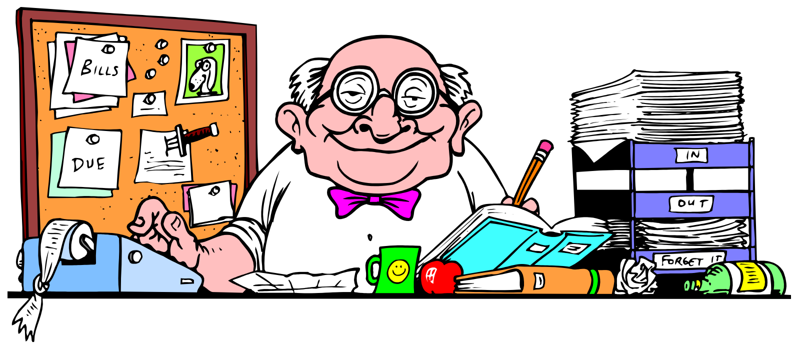
Lecture 9:
Caching & Demand page

Bo Tang @ 2021, Spring

Caching

Caching Concept

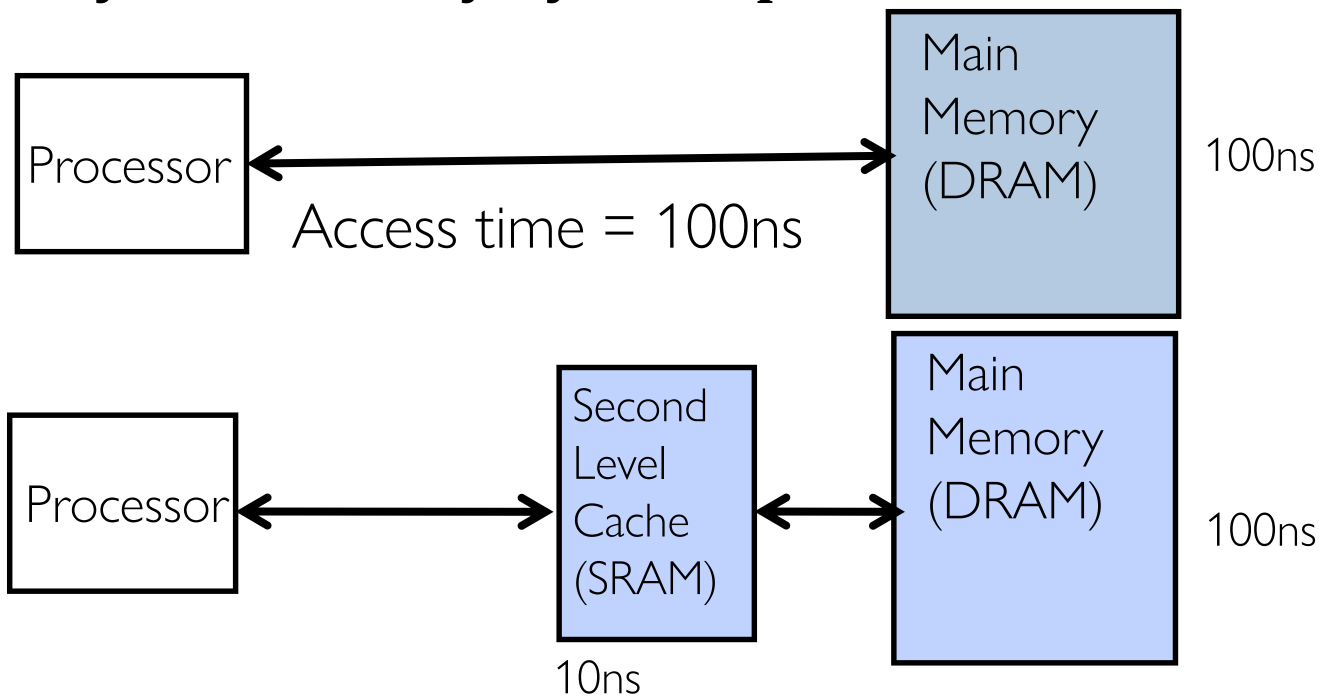


- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time =

  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Recall: In Machine Structures

- Caching is the key to memory system performance



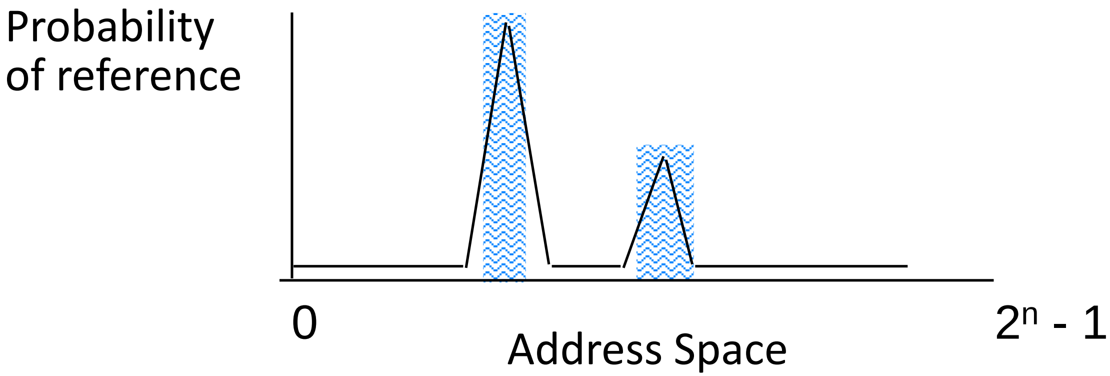
Average Access time=(Hit Rate x HitTime) + (Miss Rate x MissTime)

HitRate + MissRate = 1

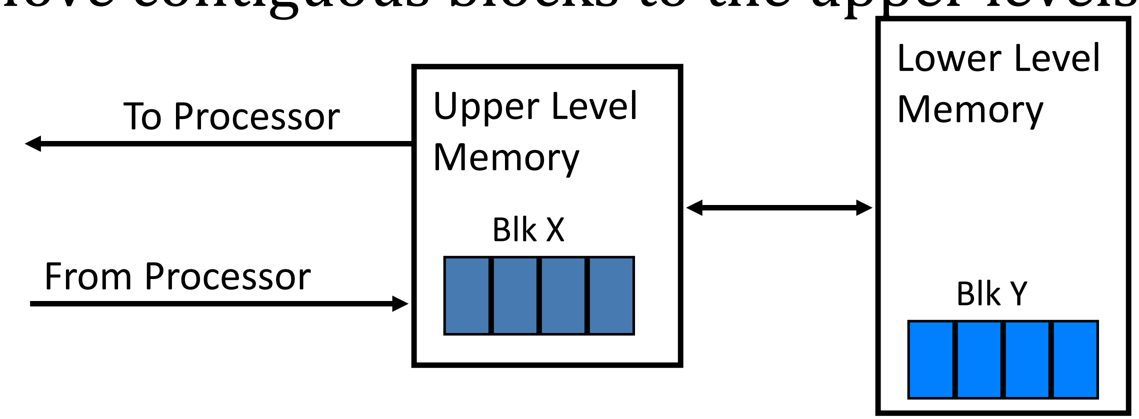
HitRate = 90% => Avg. Access Time=(0.9 x 10) + (0.1 x 100)=19ns

HitRate = 99% => Avg. Access Time=(0.99 x 10) + (0.01 x 100)=10.9 ns

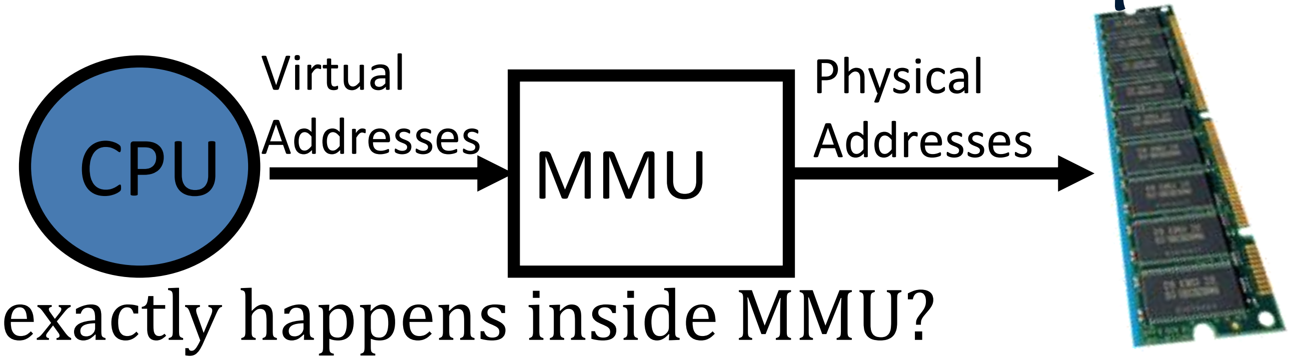
Why Does Caching Help? Locality!



- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels

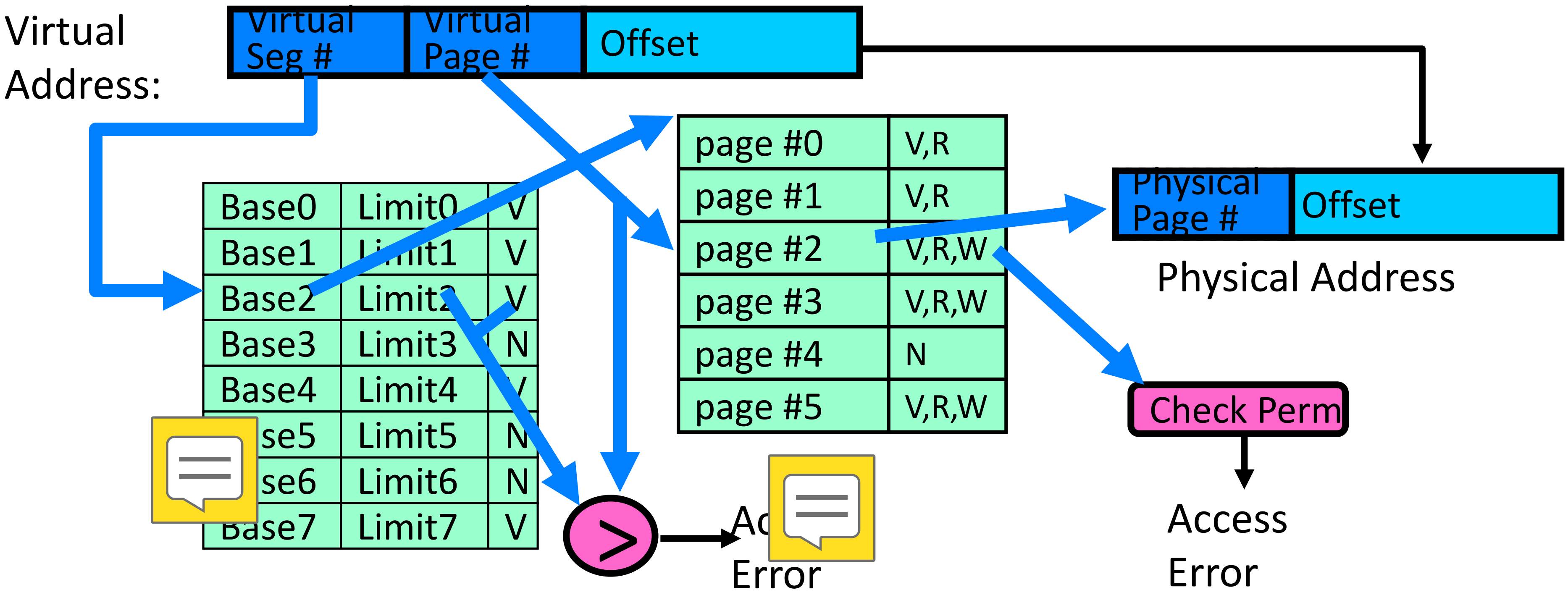


How is the Translation Accomplished?



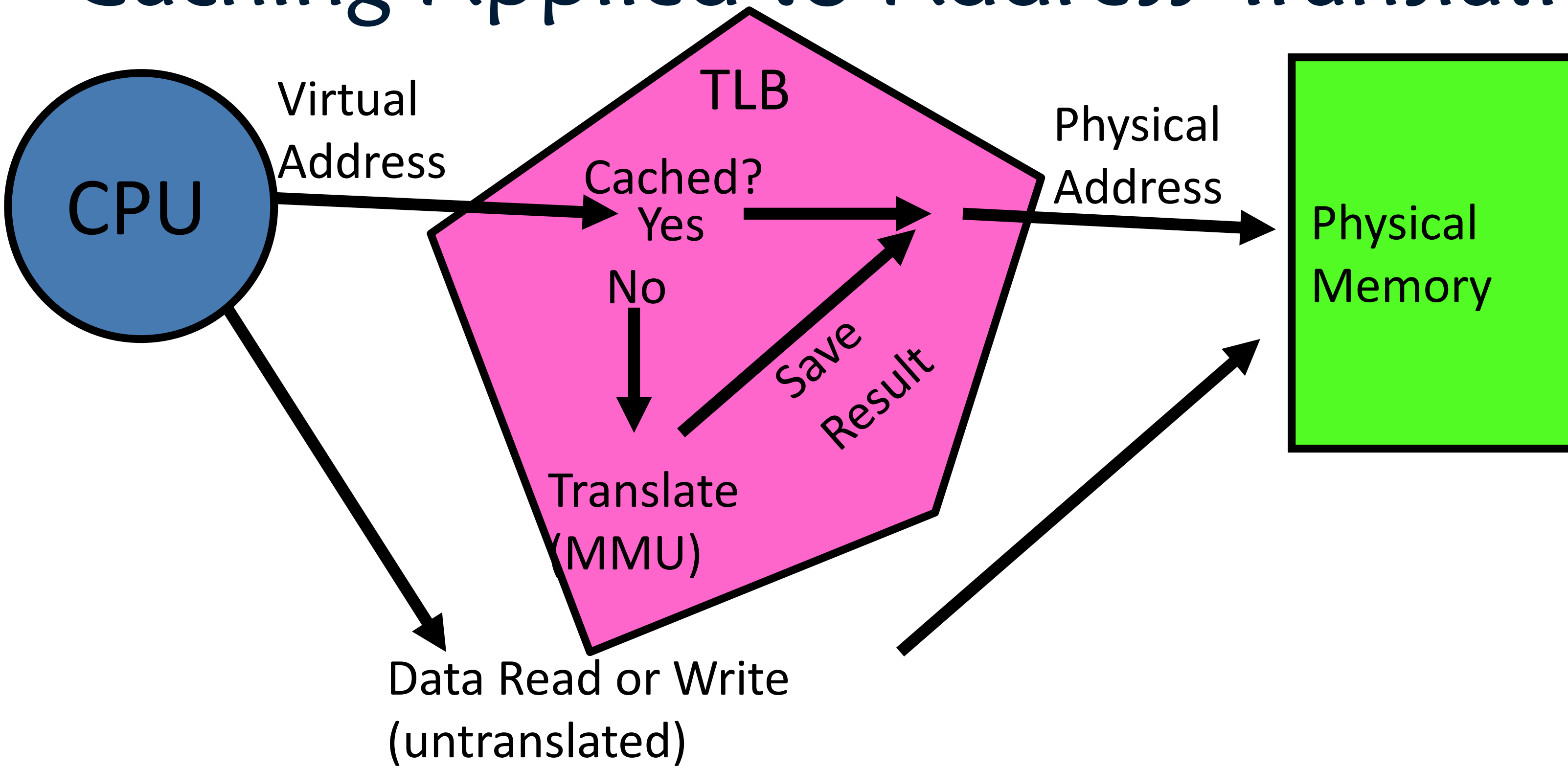
- What, exactly happens inside MMU?
- One possibility: Hardware Tree Traversal
  - For each virtual address traverses the page table in hardware
  - Generates a "Page Fault" if it encounters invalid PTE
    - Fault handler will decide what to do
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
- Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Every translation must invoke Fault!
- In fact, need way to *cache* translations for either case!

Another Major Reason to Deal with Caching



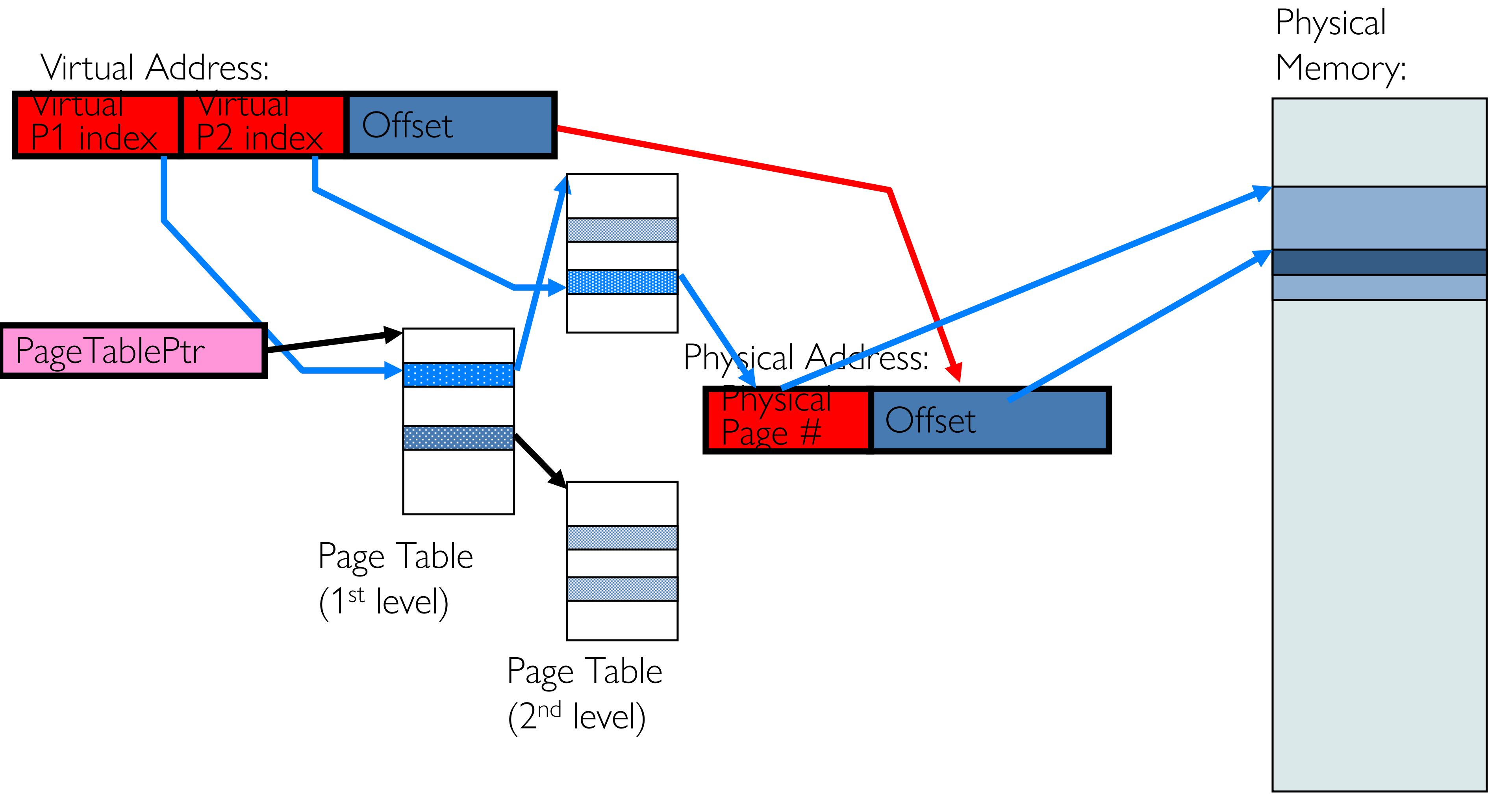
- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")

Caching Applied to Address Translation

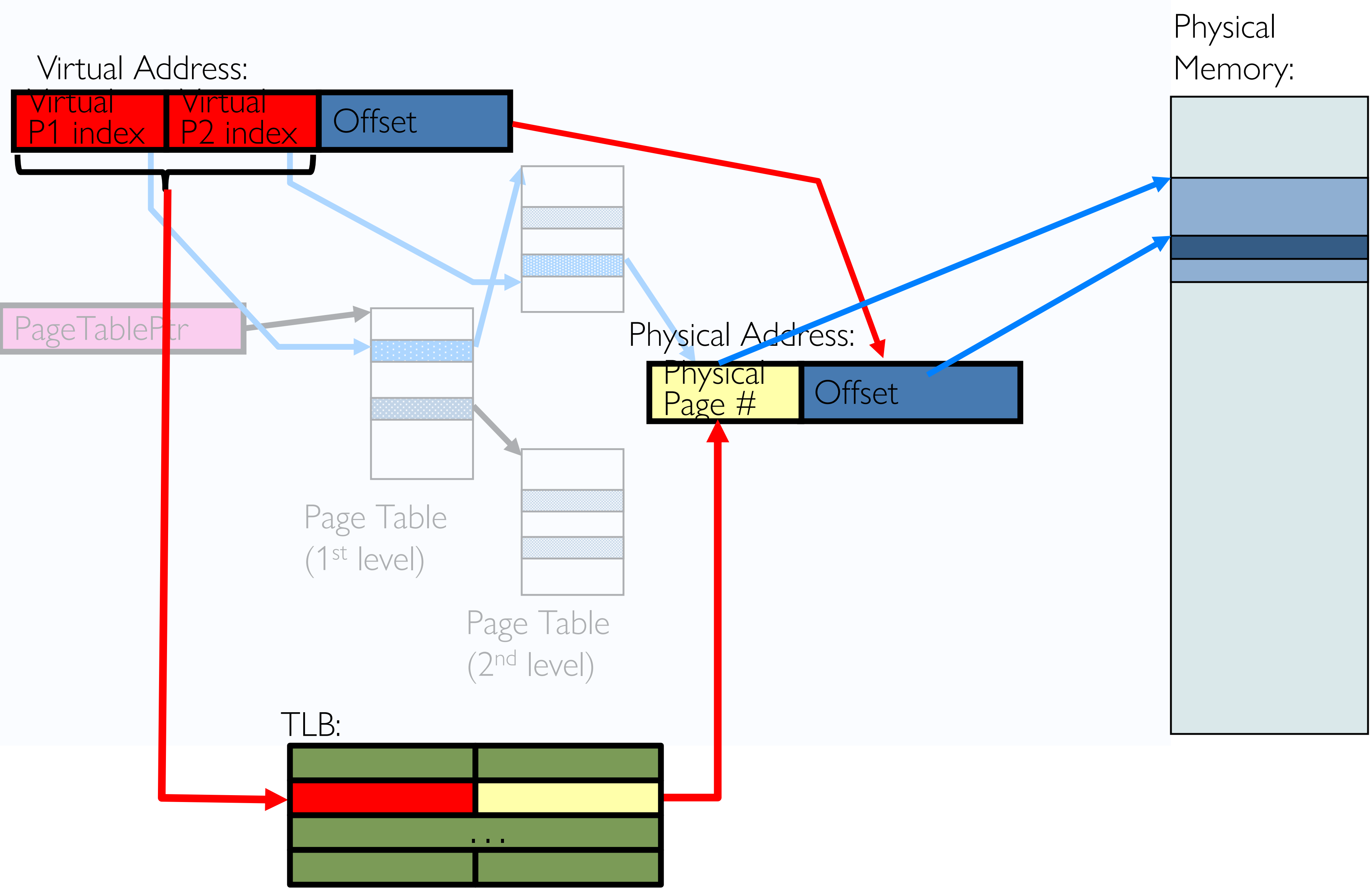


- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds

Putting All Together: Address Translation



Putting Everything Together: TLB



Where are all places that caching arises in OSes?

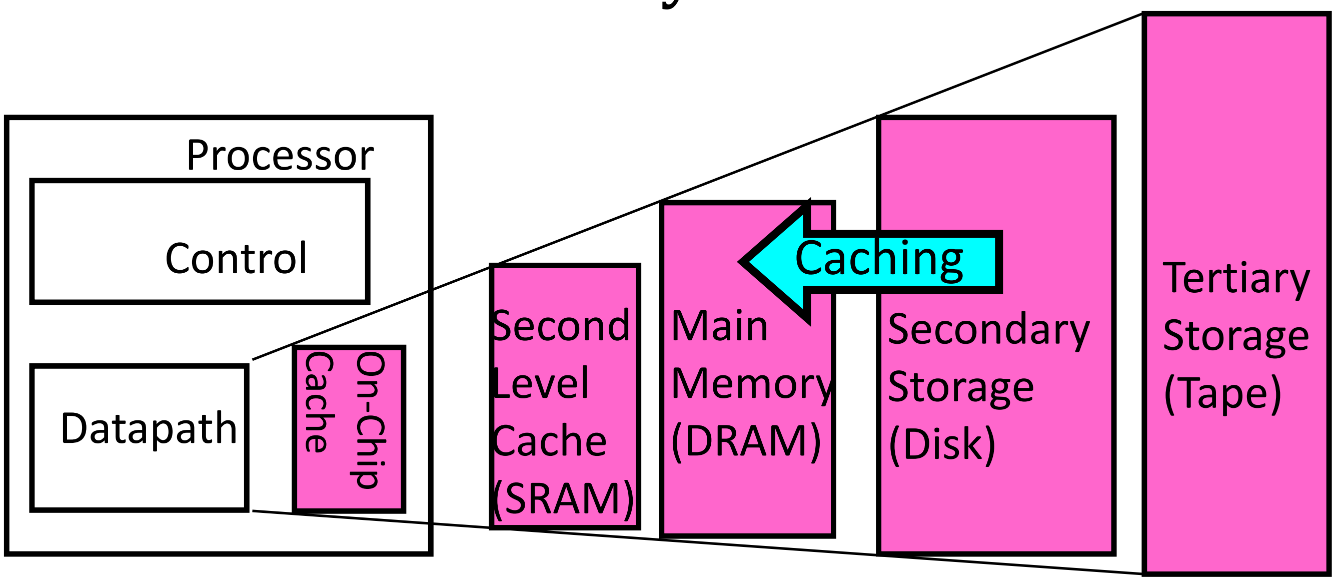
- Direct use of caching techniques
  - TLB (cache of PTEs)
  - Paged virtual memory (memory as cache for disk)
  - File systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)

- Which pages to keep in memory?
  - All-important “Policy” aspect of virtual memory
  - Will spend a bit more time on this in a moment

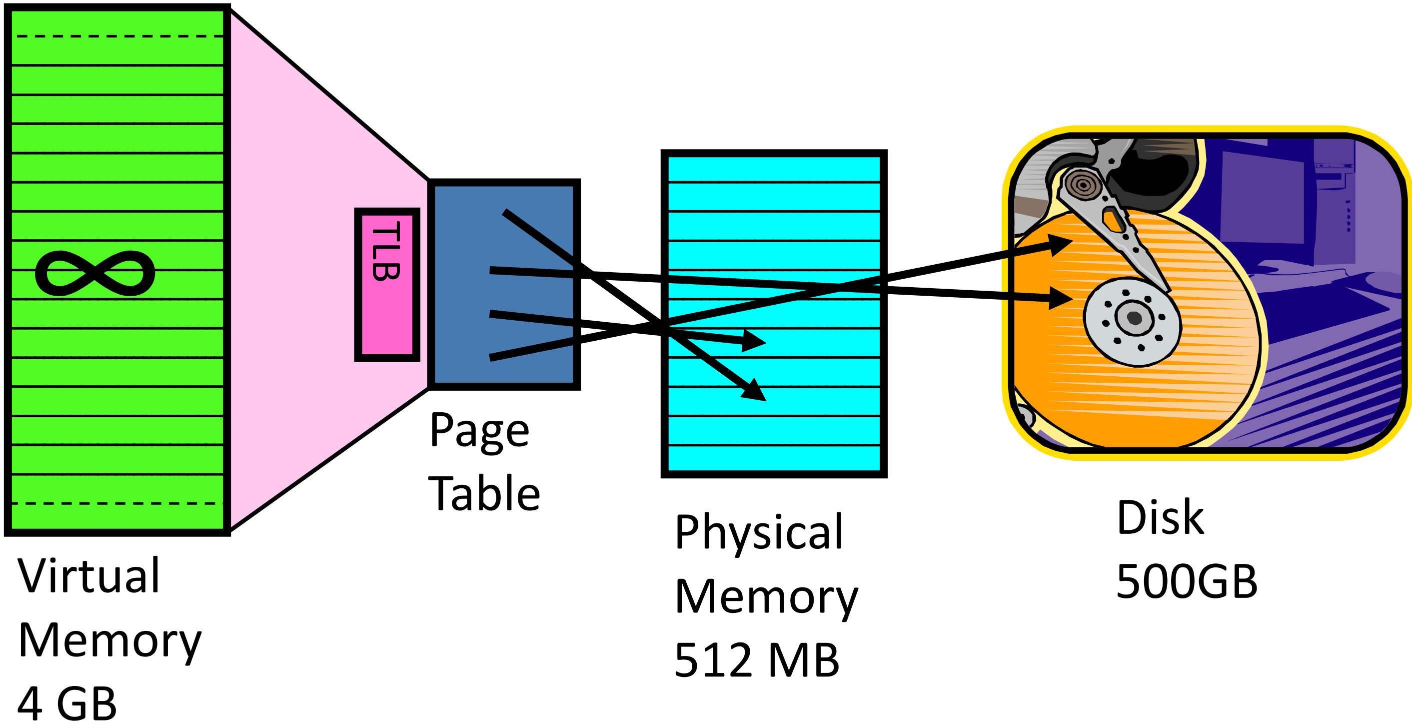
Demand Paging

Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they do not 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

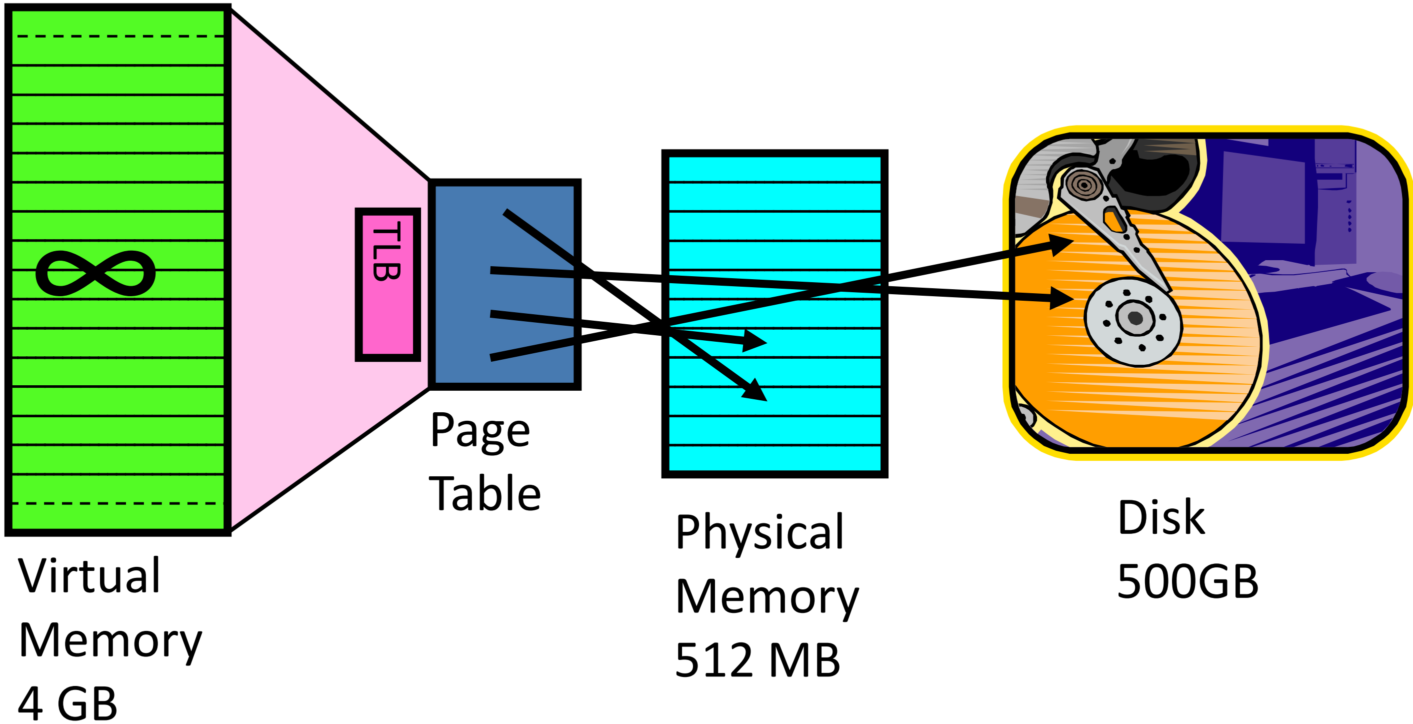


Illusion of Infinite Memory (1/2)



- Disk is larger than physical memory $\Rightarrow$
  - 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

Illusion of Infinite Memory (2/2)



- 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

Since Demand Paging is Caching, Must Ask…

- What is block size?
  - 1 page
- What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
  - Fully associative: arbitrary virtual $\rightarrow$ physical mapping
- How do we find a page in the cache when look for it?
  - First check TLB, then page-table traversal
- What is page replacement policy? (i.e. LRU, Random…)
  - This requires more explanation
- What happens on a miss?
  - Go to lower level to fill miss (i.e. disk)
- What happens on a write? (write-through, write back)
  - Definitely write-back – need dirty bit!

Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ 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"

Cache

  - 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 and any cached TLB to be invalid
    - Load new page into memory from disk
    - Update page table entry, invalidate TLB for new entry
    - Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - Suspended process sits on wait queue

Recall: Some following questions

- During a page fault, where does the OS get a free frame?
  - Keeps a free list
  - Unix runs a “reaper” if memory gets too full
    - Schedule dirty pages to be written back on disk
    - Zero (clean) pages which have not been accessed in a while
  - As a last resort, evict a dirty page first

- How can we organize these mechanisms?
  - Work on the replacement policy

- How many page frames/process?
  - Like thread scheduling, need to “schedule” memory resources:
    - Utilization? fairness? priority?
  - Allocation of disk paging bandwidth

Demand Paging Cost Model

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - EAT = Hit Time + Miss Rate x Miss Penalty
- 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:

    EAT = 200ns + p x 8 ms

    = 200ns + p x 8,000,000ns
- 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 we want slowdown by less than 10%?
  - 200ns x 1.1 < EAT $\Rightarrow$ p < 2.5×10^{-6}
  - This is about 1 page fault in 400,000!

What Factors Lead to Misses?

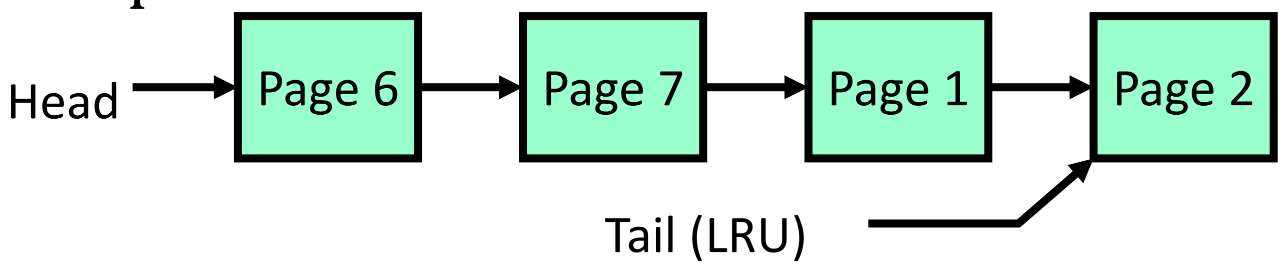
- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
    - Need to predict future somehow! More later
- Capacity Misses:
  - Not enough memory. Must somehow increase available memory size.
  - Can we do this?
    - One option: Increase amount of DRAM (not quick fix!)
    - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- Conflict Misses:
  - Technically, conflict misses don’t exist in virtual memory, since it is a “fully-associative” cache
- Policy Misses:
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

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 – throws out heavily used pages instead of infrequently used
- MIN (Minimum):
  - Replace page that will not be used for the longest time
  - Great, but cannot really know future…
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that has not 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!



  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

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
  - Where will D be brought in? Look for page not referenced farthest in future
- What will LRU do?
  - Same decisions as MIN here, but will not 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: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!

When will LRU perform badly?

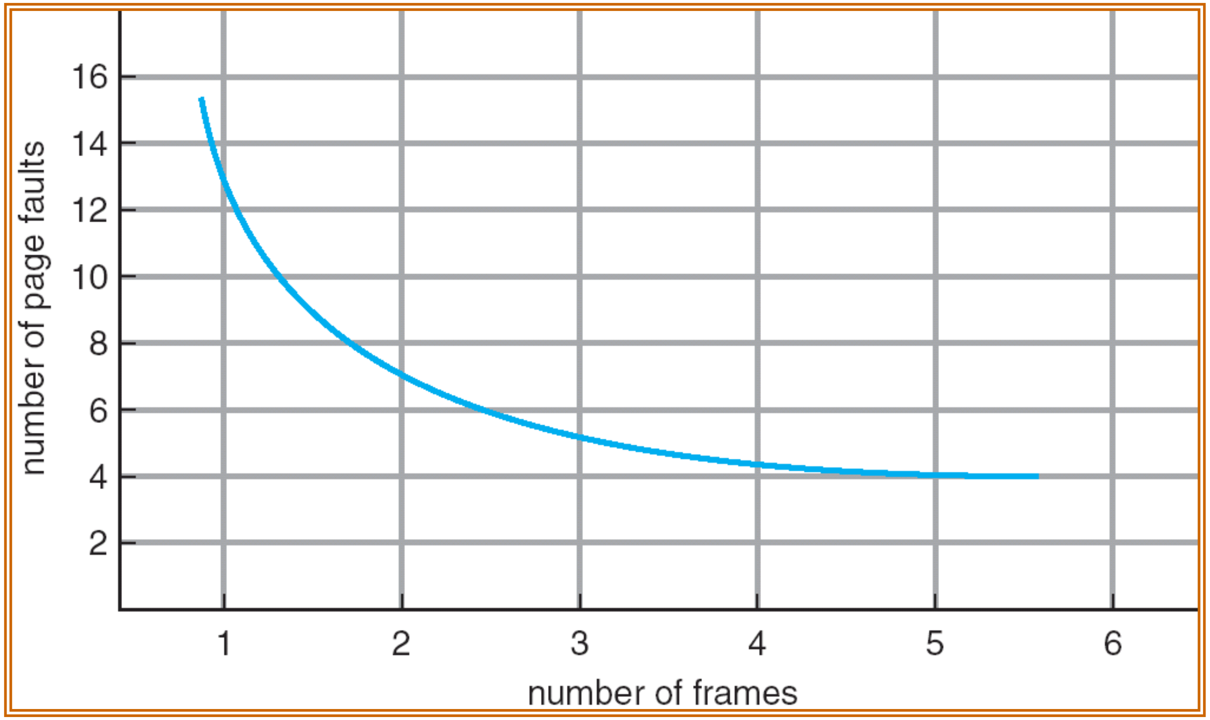
- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

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

Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate drops
  - Does this always happen?
  - Seems like it should, right?
- No: Bélády's anomaly
  - Certain replacement algorithms (FIFO) don't have this obvious property!

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 Bélády’s anomaly)

Ref: 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	

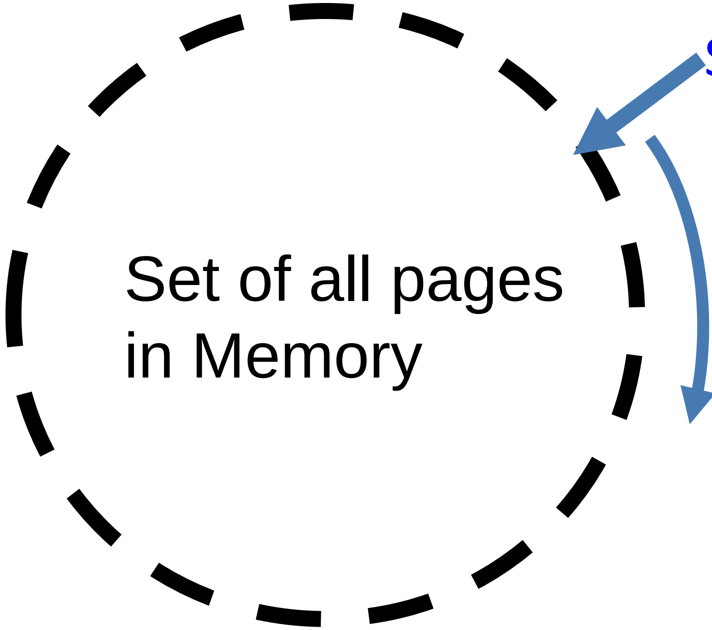
Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- 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

Implementing LRU

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (*approximation to approximation to MIN*)
  - Replace an old page, not the oldest page
- Details:
  - Hardware "use" bit per physical page:
    - Hardware sets use bit on each reference
    - If use bit is not set, means not referenced in a long time
  - On page fault:
    - Advance clock hand (not real time)
    - Check use bit: $1 \rightarrow$ used recently; clear and leave alone
$0 \rightarrow$ selected candidate for replacement
  - Will always find a page or loop forever?
    - Even if all use bits set, will eventually loop around $\Rightarrow$ FIFO

Clock Algorithm: Not Recently Used



Single Clock Hand:

Advances only on page fault!

Check for pages not used recently

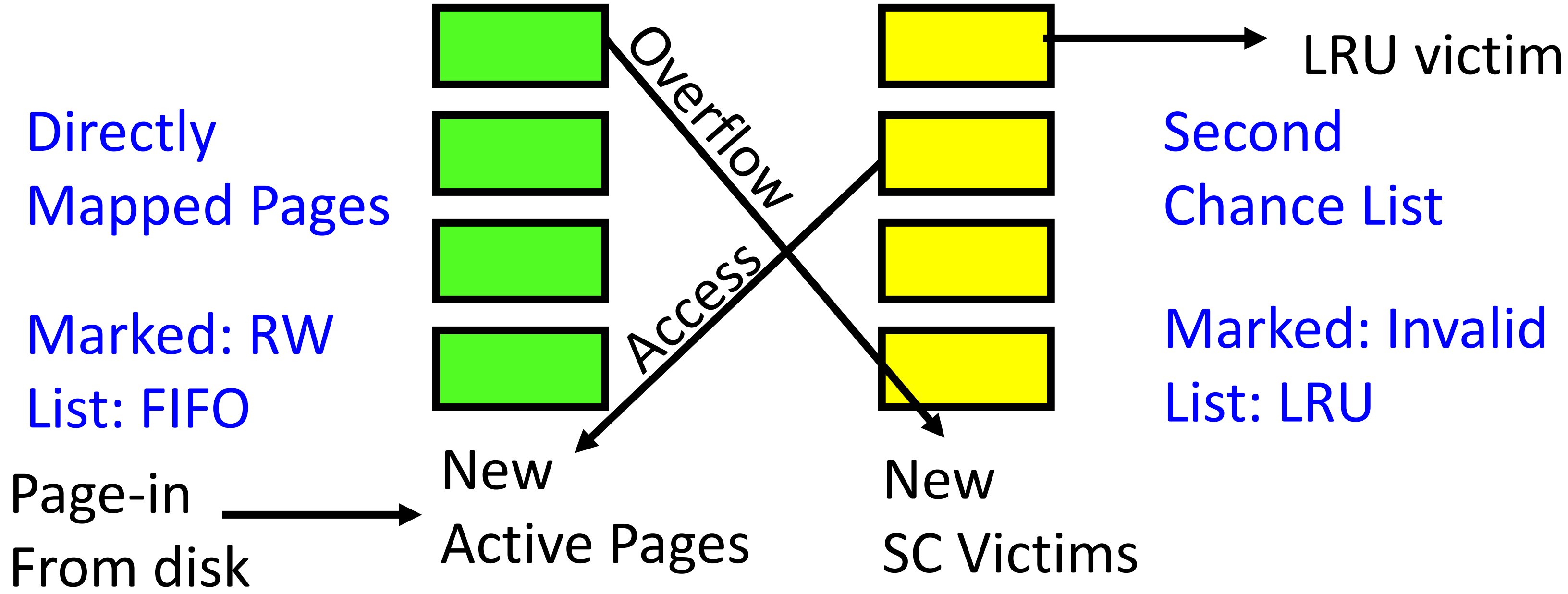
Mark pages as not used recently

- What if hand moving slowly?
  - Good sign or bad sign?
    - Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1 → clear use and also clear counter (used in last sweep)
    - 0 → increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approximation to LRU
    - If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - Clean pages, use N=1
    - Dirty pages, use N=2 (and write back to disk when N=1)

Second-Chance List Algorithm



- Split memory in two: Active list, SC list
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Demand Paging (more details)

- Does software-loaded TLB need use bit?
  Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    - Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    - Can't push page out to disk without invalidating all PTEs

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes $\rightarrow$ process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

    s_i = size of process p_i and $S = \Sigma s_i$

    m = total number of frames

    $$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$
- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

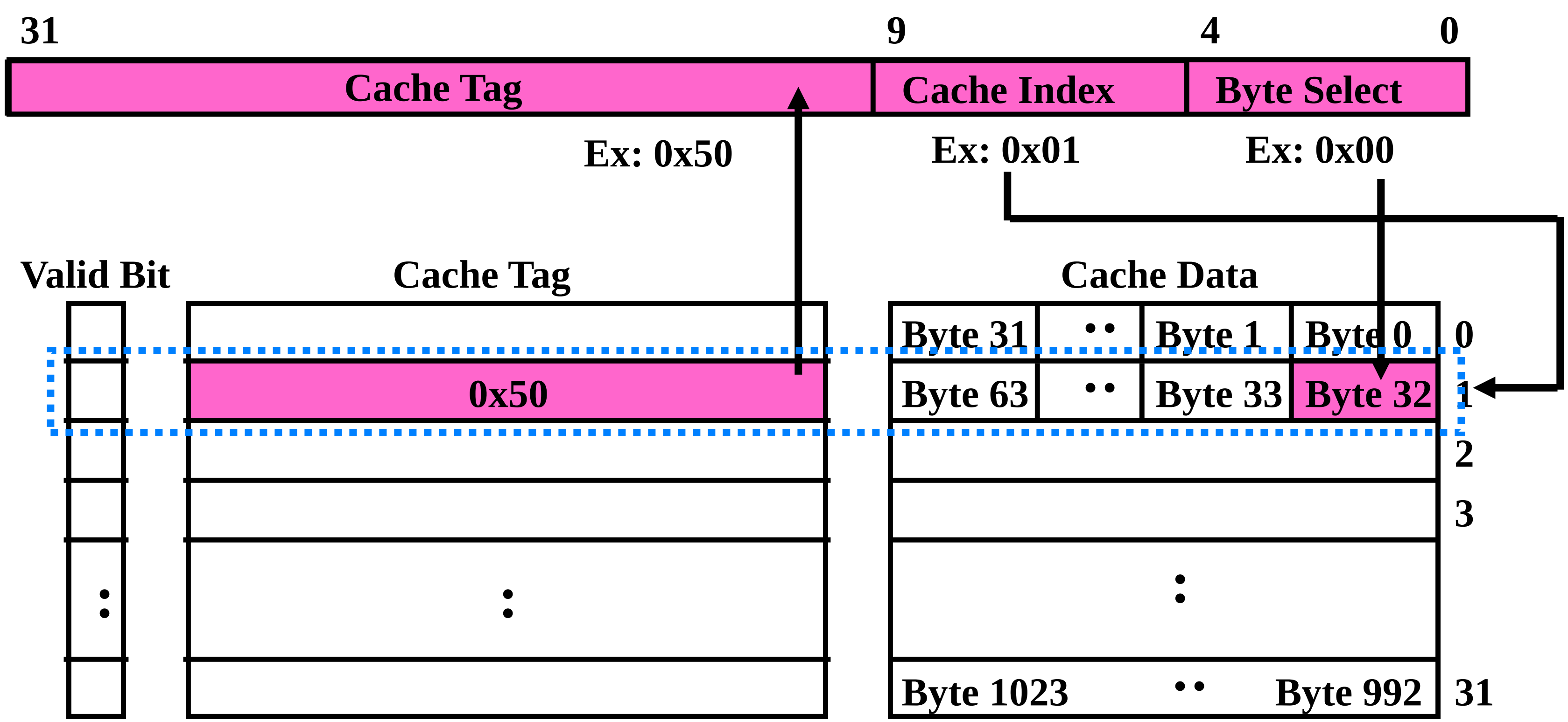
Summary

- 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
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace
- N^{th}-chance clock algorithm: Another approximate LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approximate LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.

Thank You!

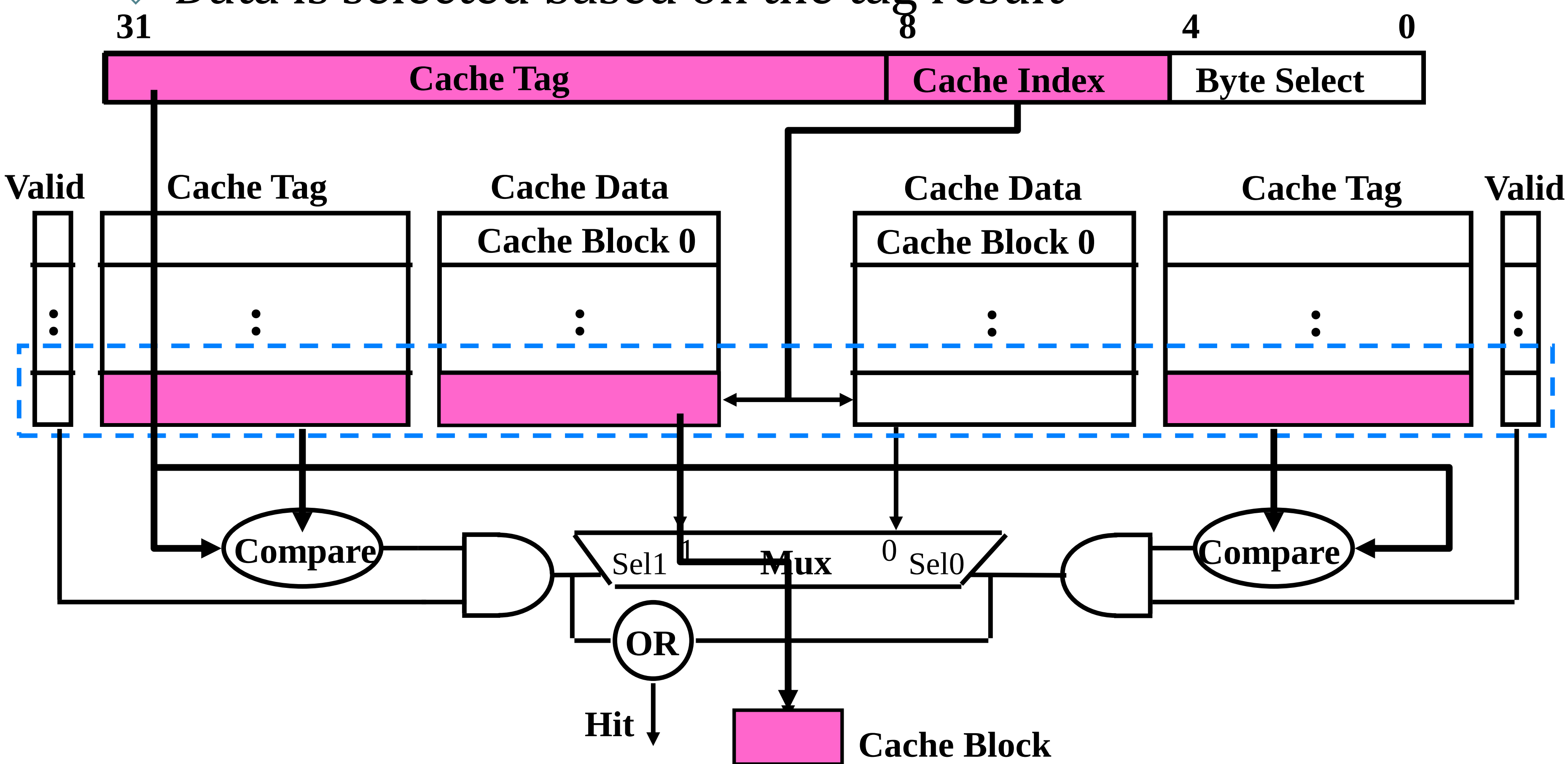
Review: Direct Mapped Cache

- Direct Mapped 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block



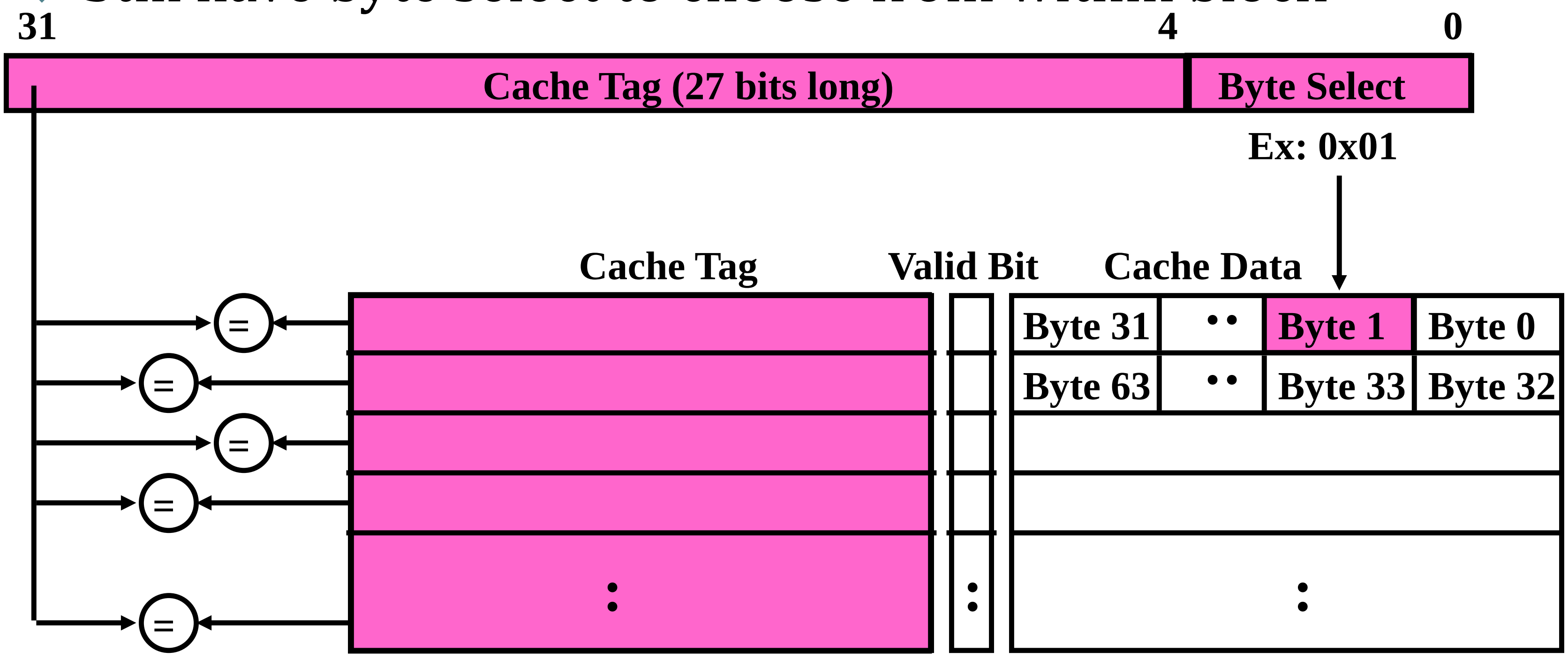
Review: Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



Review: Fully Associative Cache

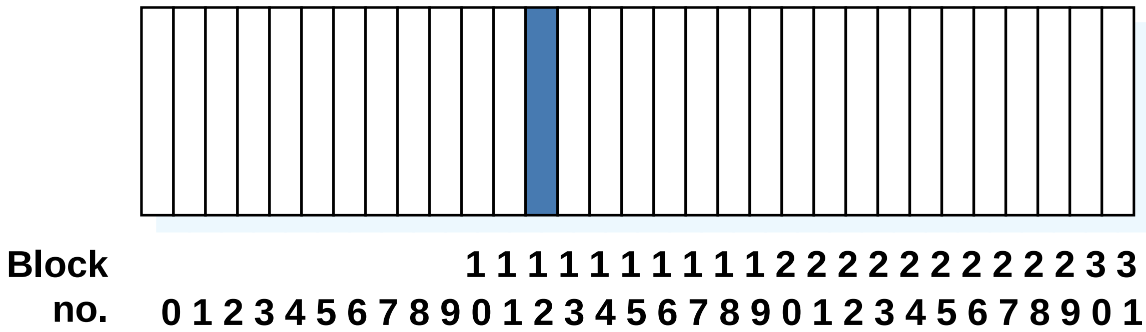
- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



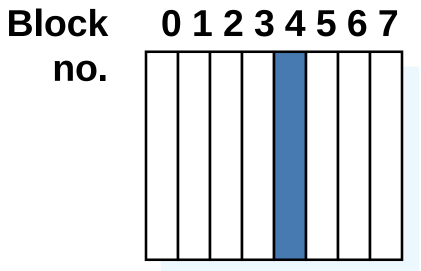
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

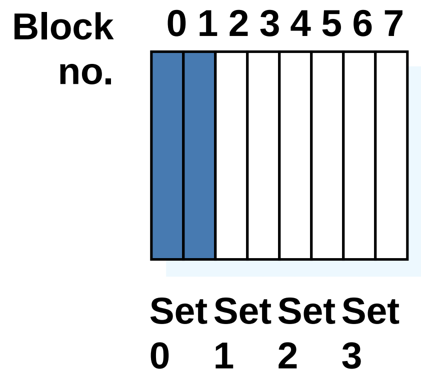
32-Block Address Space:



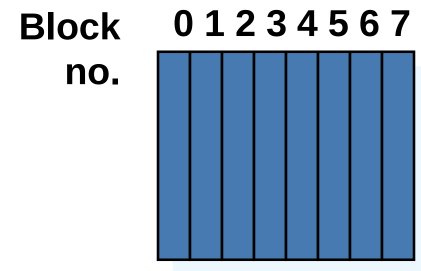
Direct mapped:
block 12 can go only into block 4 (12 mod 8)



Set associative:
block 12 can go anywhere in set 0 (12 mod 4)



Fully associative:
block 12 can go anywhere



Review: Which block should be replaced on a miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

- Miss rates for a workload:

	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%