Caching Basics and Demand Paging

25 January 2026
Lecture 13

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Disk block

- Sector

Defragment

Spinning HDD

- Cylinder
- Surface
- Sector

Logical Block Addressing

File header

- File number
- File attributes

File control block

File Allocation Table (FAT)

Free list

Inode

- Indirect pointer
- Double indirect pointer
- Triple indirect pointer

Block groups

Topics for Today

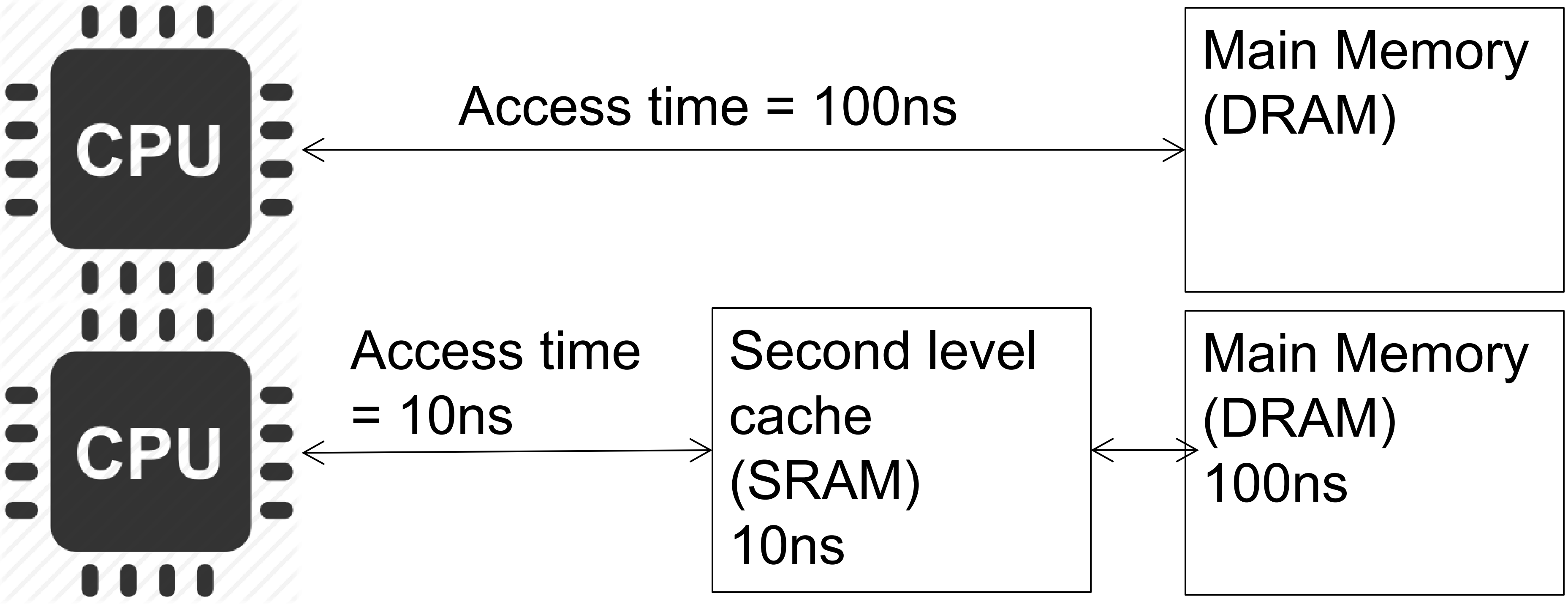
- Caching Basics
- Caching on Address Translations (TLB)
- Demand Paging

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 of the techniques that are 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)

Image source: https://www.drupal.org/files/project-images/squirrel%20cache.jpg

In Machine Structures



- Average Access time = (Hit Rate x HitTime) + (Miss Rate x MissTime)
- $HitRate + MissRate = 1$
- $HitRate = 90\%$ => Average Access Time = $19\ ns$
- $HitRate = 99\%$ => Average Access Time = $10.9ns$

Why Bother with Caching?

(Really Joy's law)

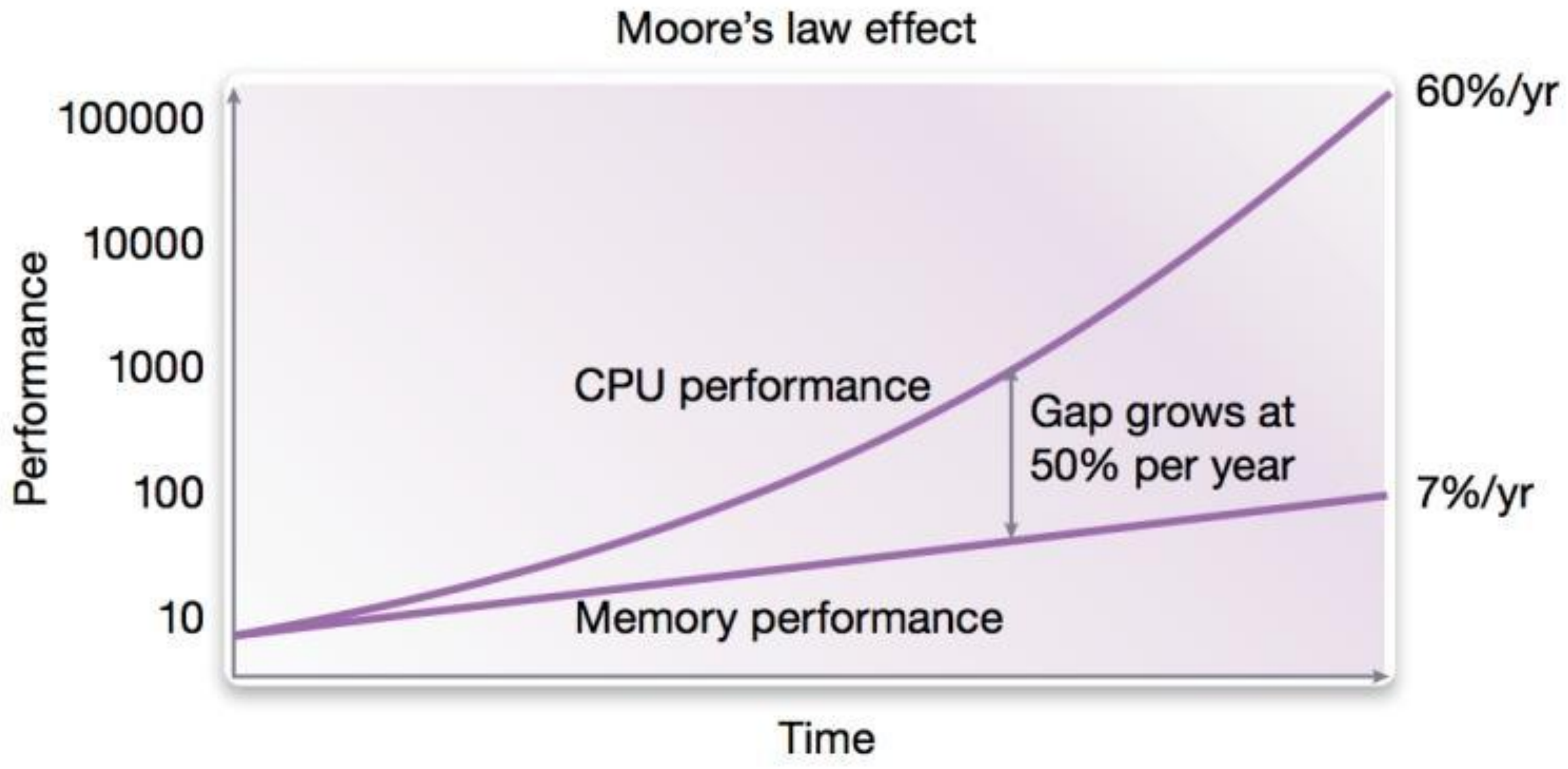
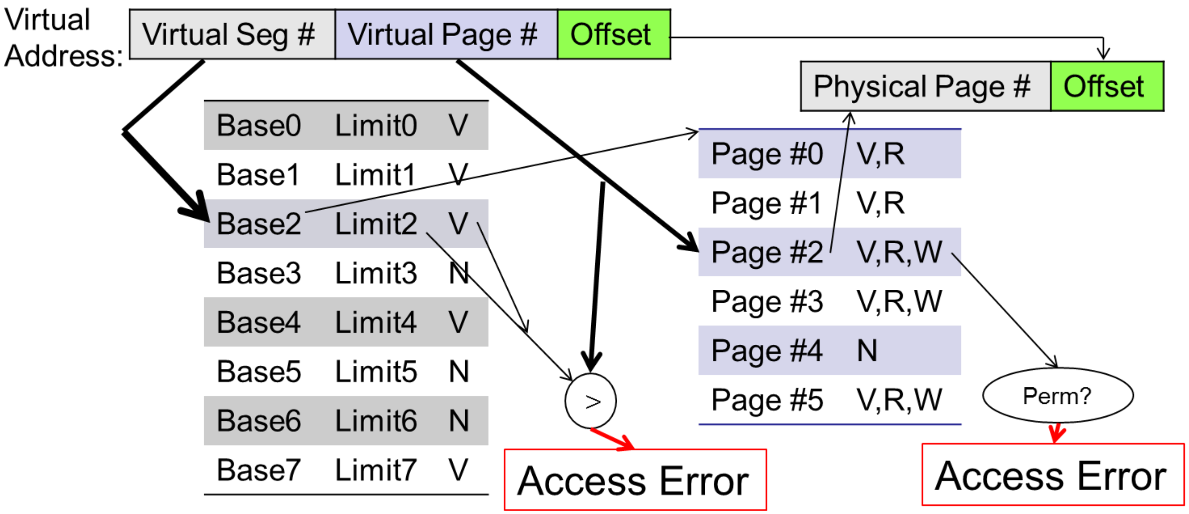


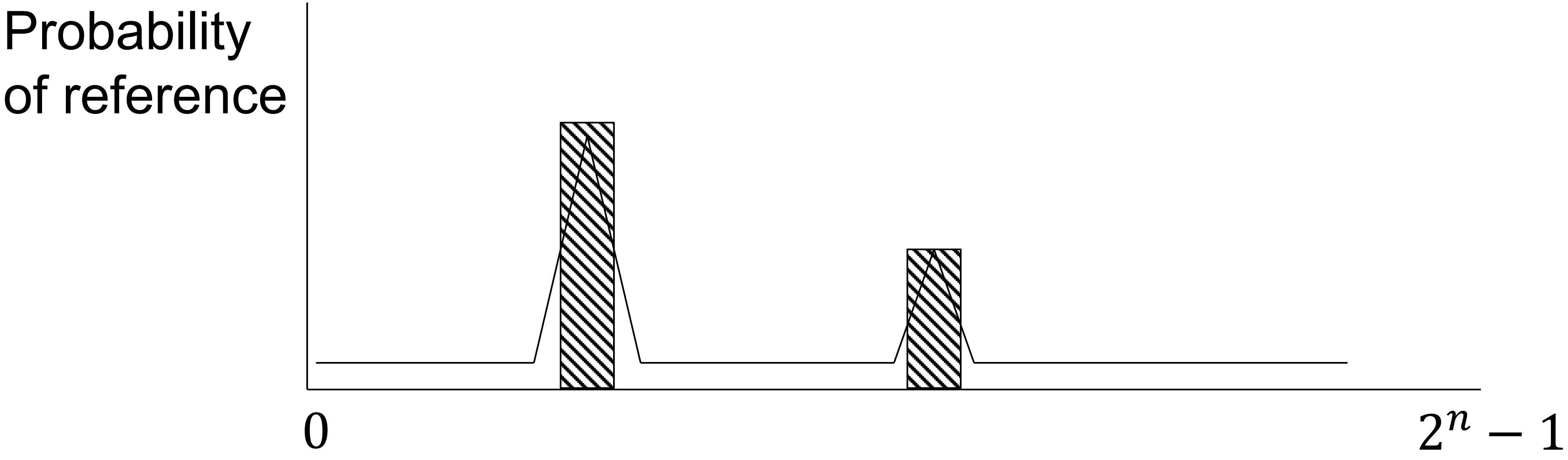
Image source: https://images.vice.com/motherboard/content-images/contentimage/no-id/140364245678419.jpg

Another Reason for Caching



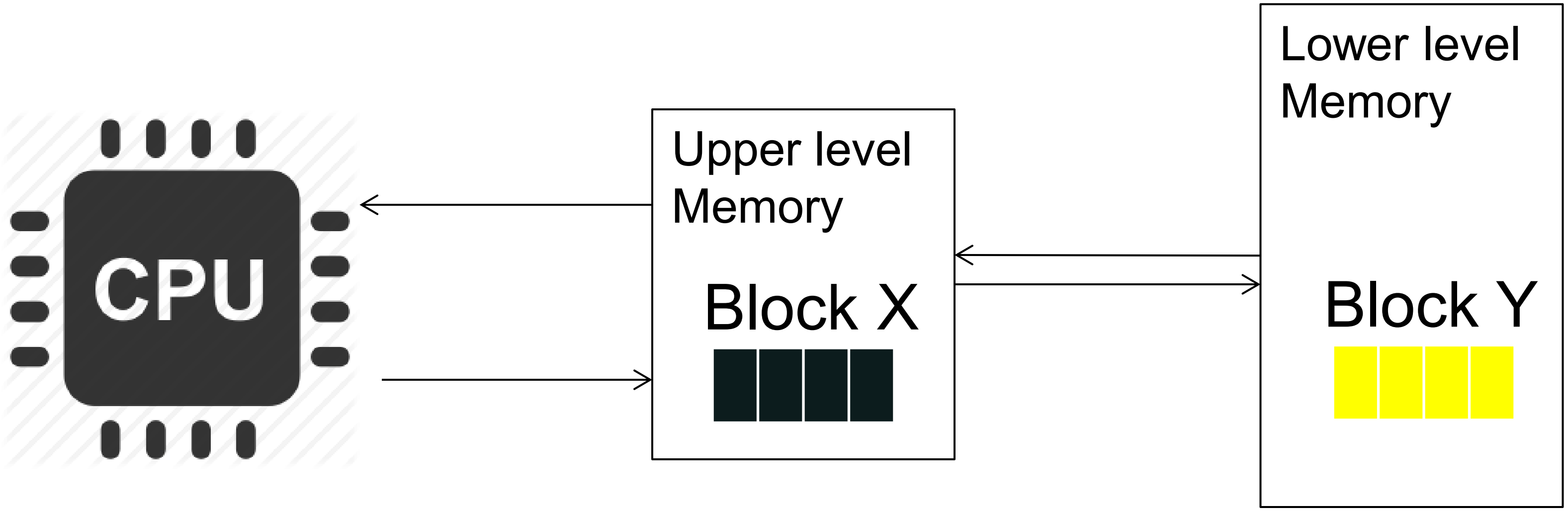
- We can’t afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - Translation Cache: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

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

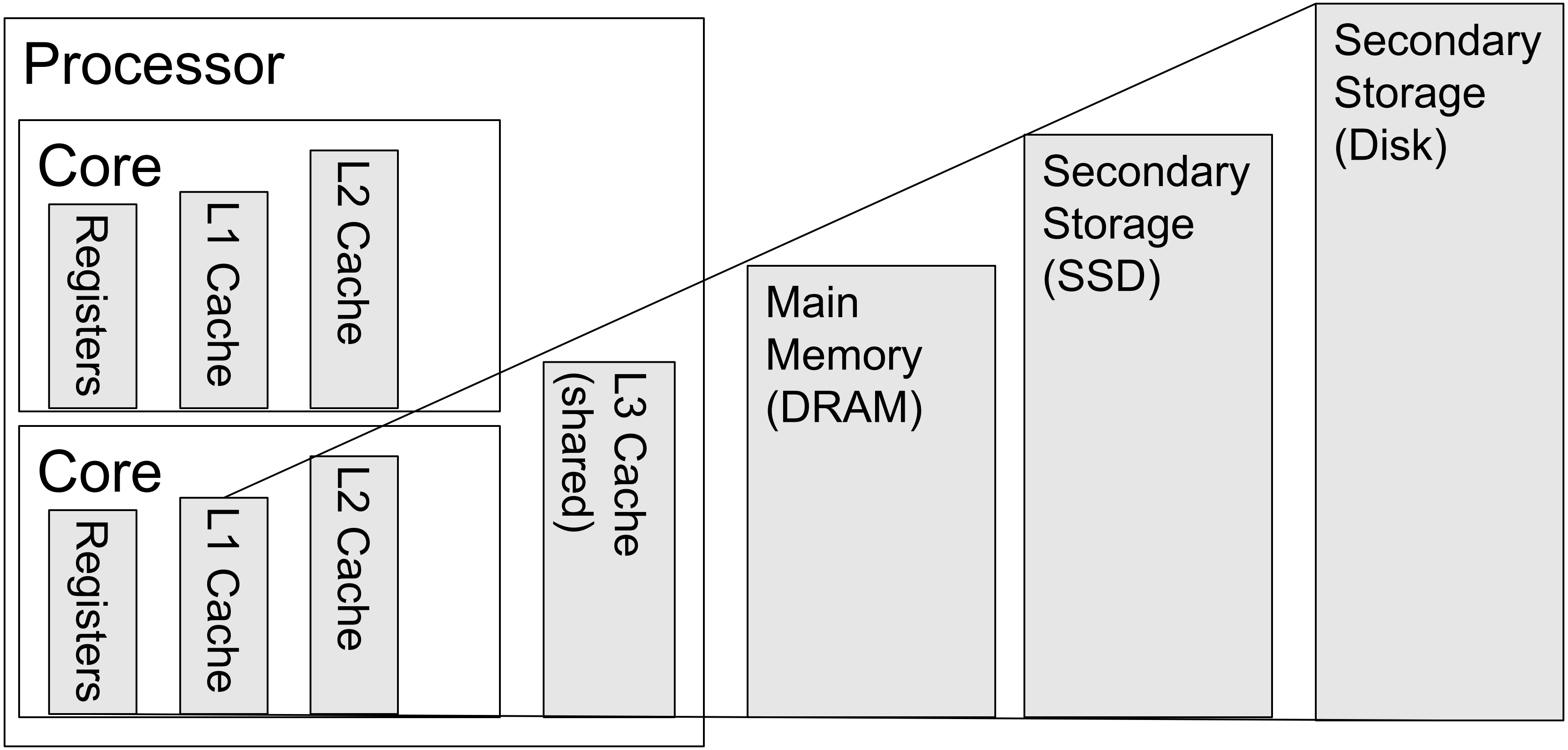
Why Does Caching Help? Locality!



- 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

Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1ms)	10,000,000
Size (B):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

Sources of Cache Misses

Compulsory

- Cold start or process migration, first reference: first access to a block
- “Cold” fact of life: not a whole lot you can do about it
- Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant

Capacity

- Cache cannot contain all blocks access by the program
- Solution: increase cache size

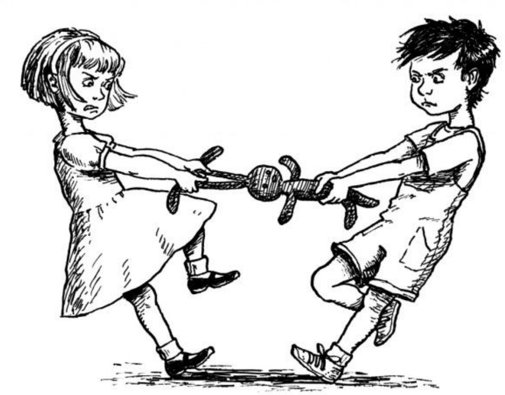
Image: http://37th.btck.co.uk/SorrywereFull

Sources of Cache Misses

Copyright © Ron Leishman * http://ToonClips.com/1388

Conflict (collision)

- Multiple memory locations mapped to the same cache location
- Solution 1: increase cache size
- Solution 2: increase associativity



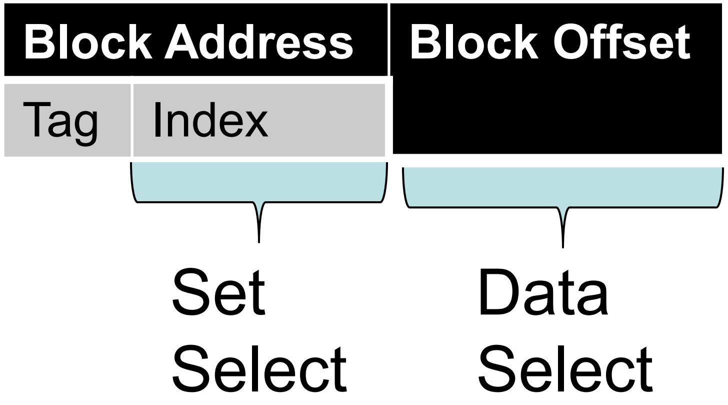
Coherence

- Invalidation
- Other process (e.g., I/O) updates memory

SOTALLY
TOBER

Image: https://www.spreadshirt.com/sotally+tober+totally+sober-A101118405

How is a Block found in a Cache?



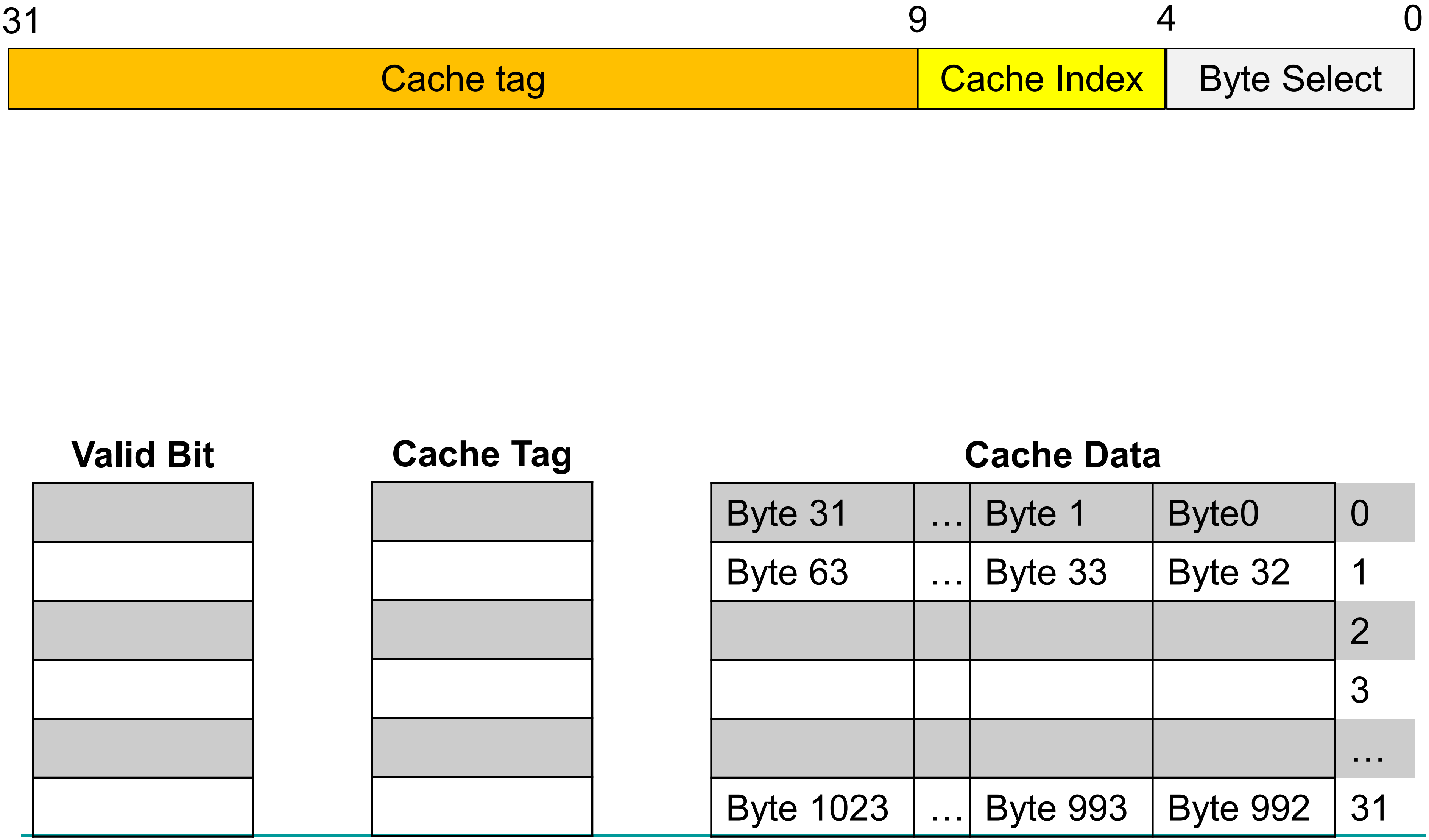
- Index Used to Lookup Candidates in Cache
  - Index identifies the set (may be >1 in an associative cache)
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss
- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don’t have data select field

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 $(Block\ Size = 2^M)$

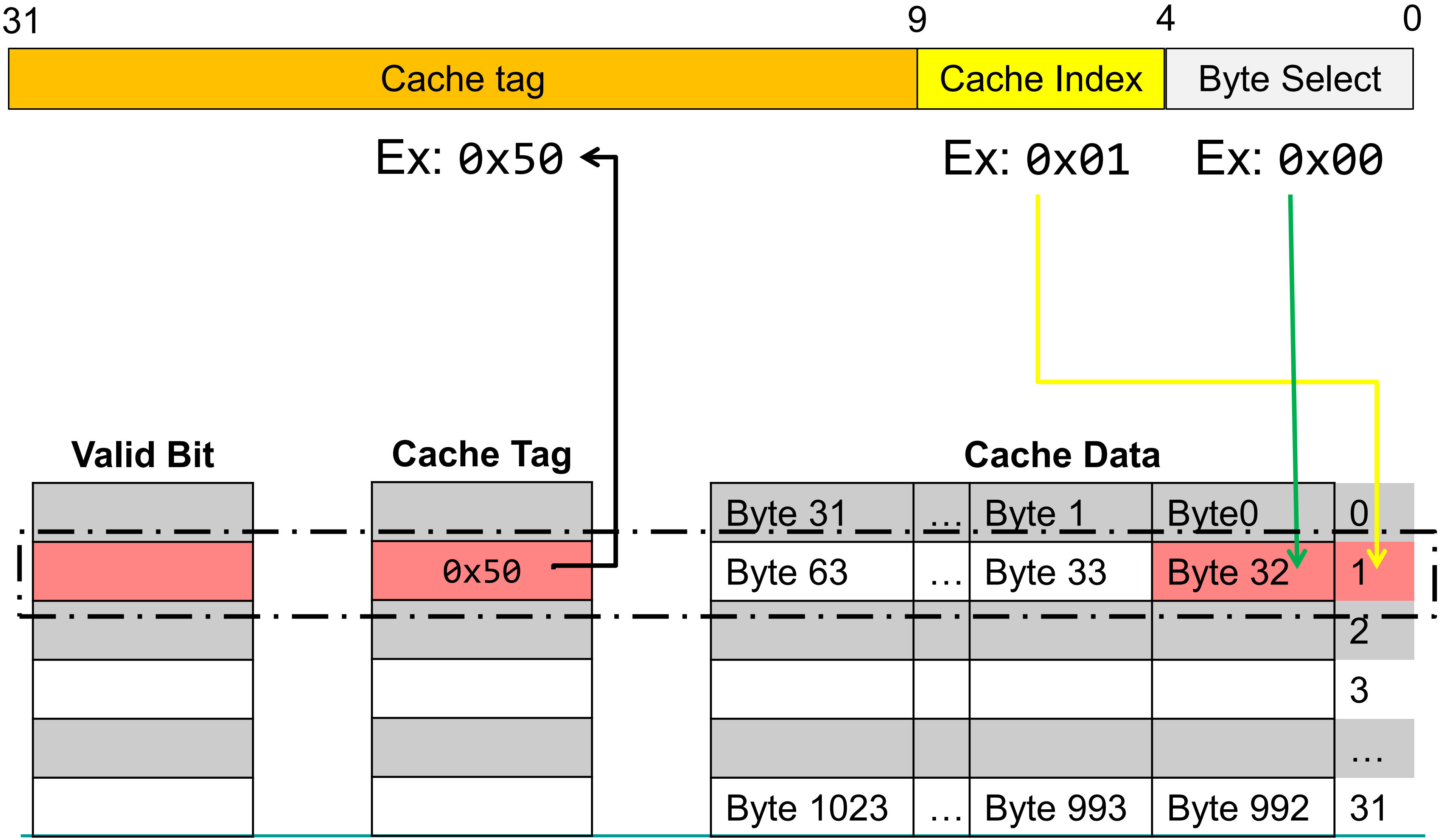
- Example: $1\ KB$ Direct Mapped Cache with $32\ B$ Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

Direct Mapped Cache



Valid Bit	Cache Tag	Cache Data				
		Byte 31	…	Byte 1	Byte0	0
		Byte 63	…	Byte 33	Byte 32	1
						2
						3
						…
		Byte 1023	…	Byte 993	Byte 992	31

Direct Mapped Cache

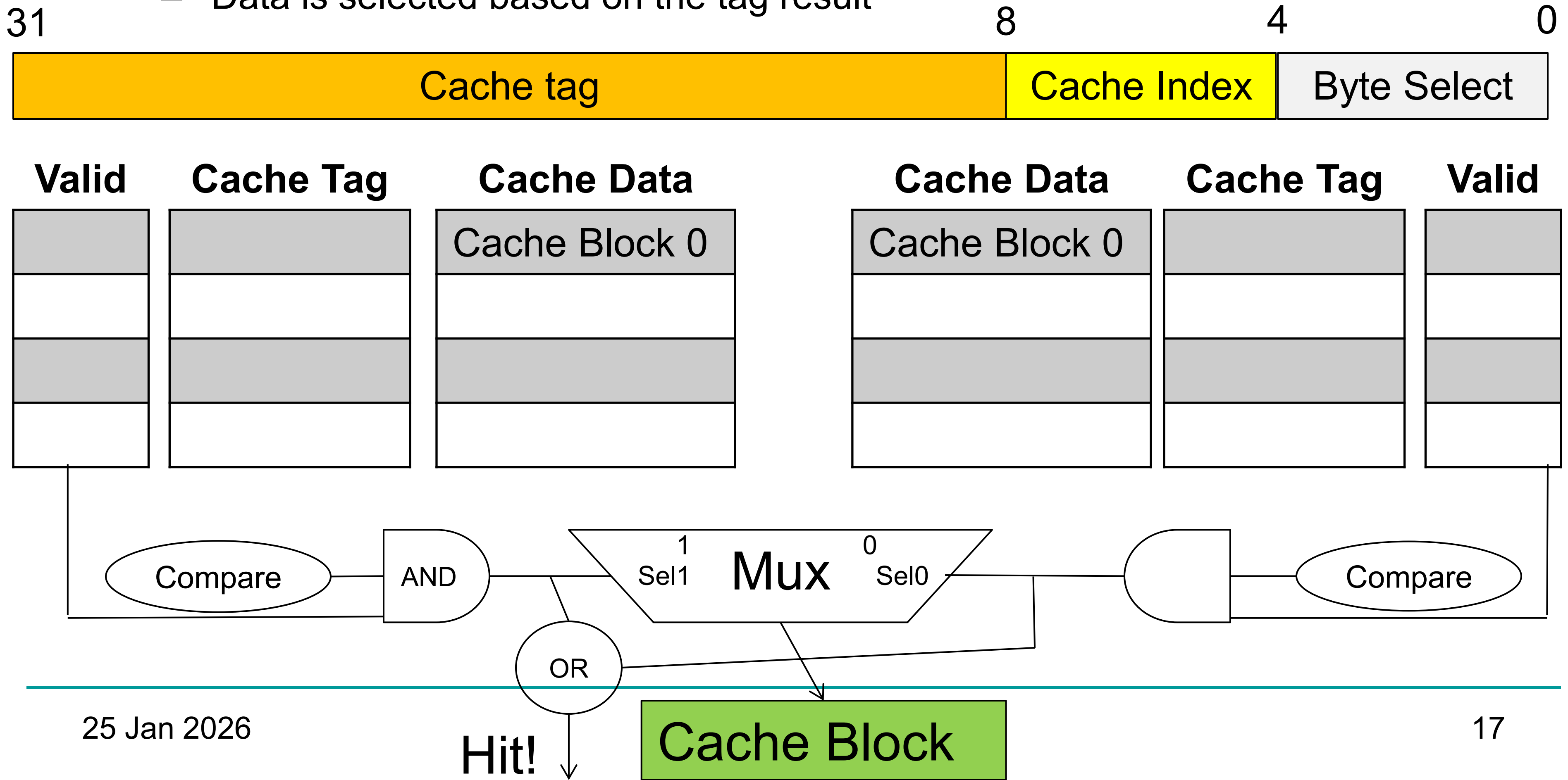


31	9	4	0
Cache tag	Cache Index	Byte Select	
Ex: `0x50`	Ex: `0x01`	Ex: `0x00`	

Valid Bit	Cache Tag	Cache Data				
		Byte 31	…	Byte 1	Byte0	0
	`0x50`	Byte 63	…	Byte 33	Byte 32	1
						2
						3
						…
		Byte 1023	…	Byte 993	Byte 992	31

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



31 8 4 0

Cache tag | Cache Index | Byte Select

Valid | Cache Tag | Cache Data | Cache Data | Cache Tag | Valid

Cache Block 0

Cache Block 0

Compare

AND

1 Sel1

Mux

0 Sel0

Compare

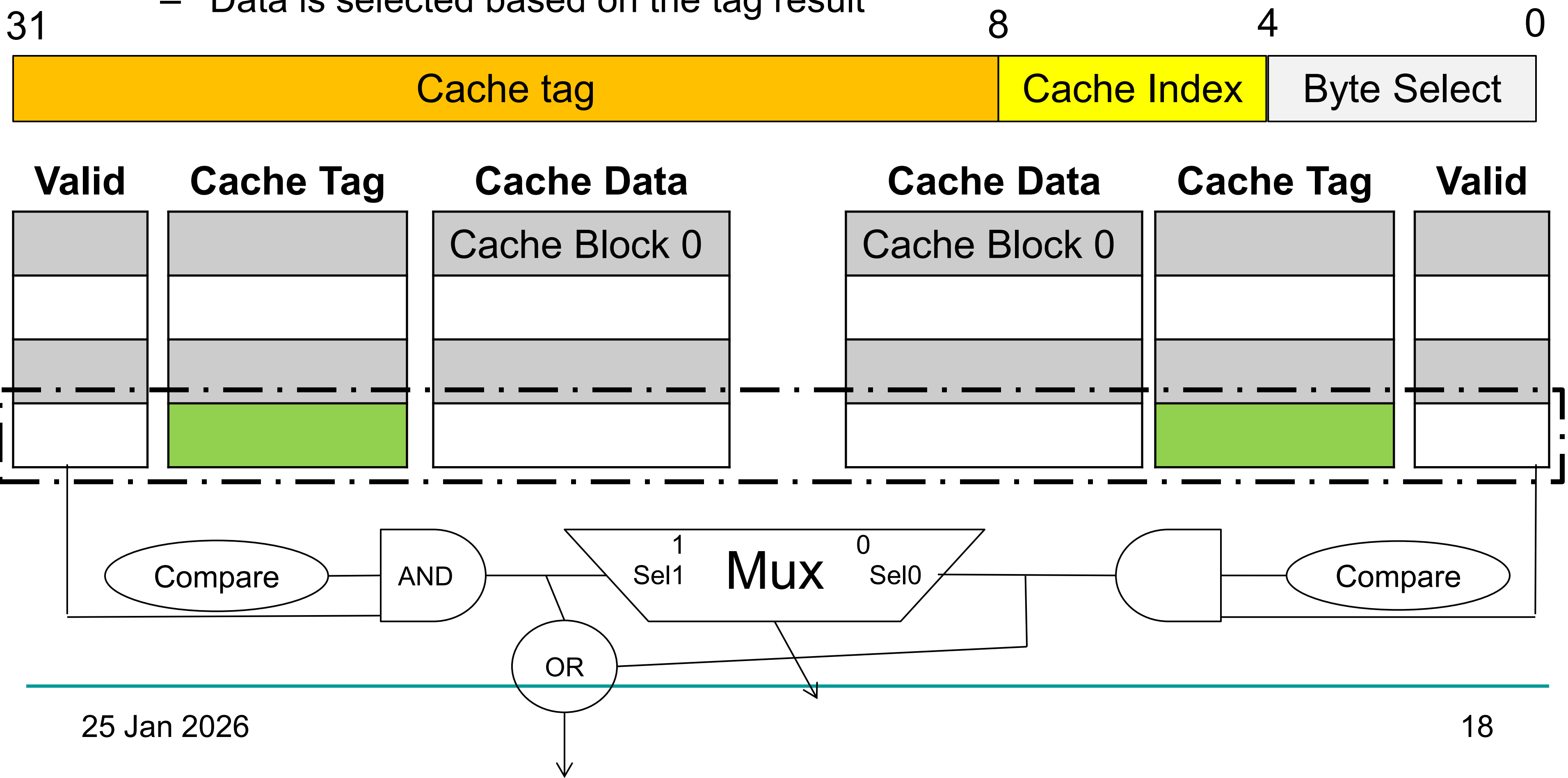
OR

Hit!

Cache Block

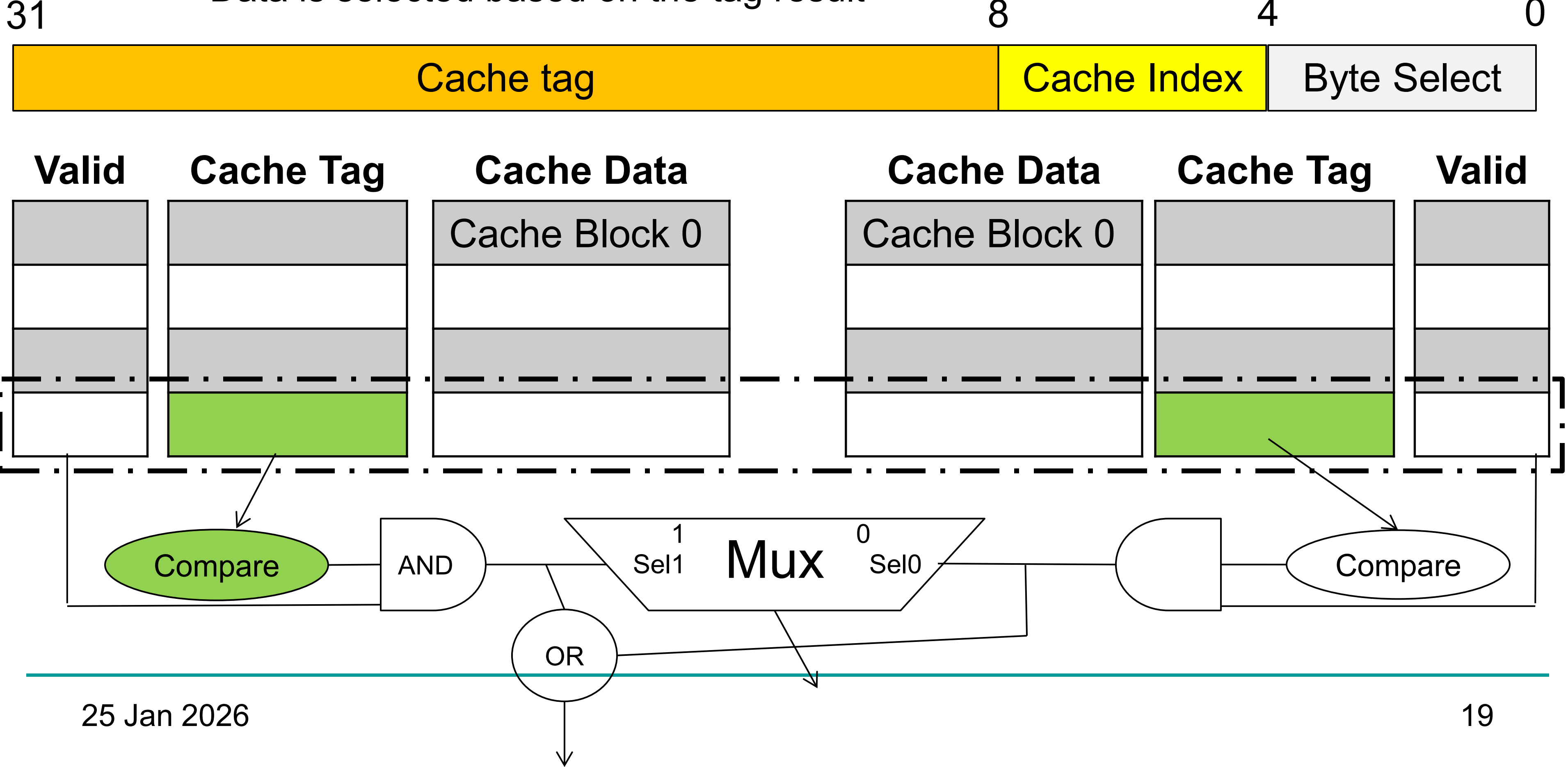
Set Associative Cache

- *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



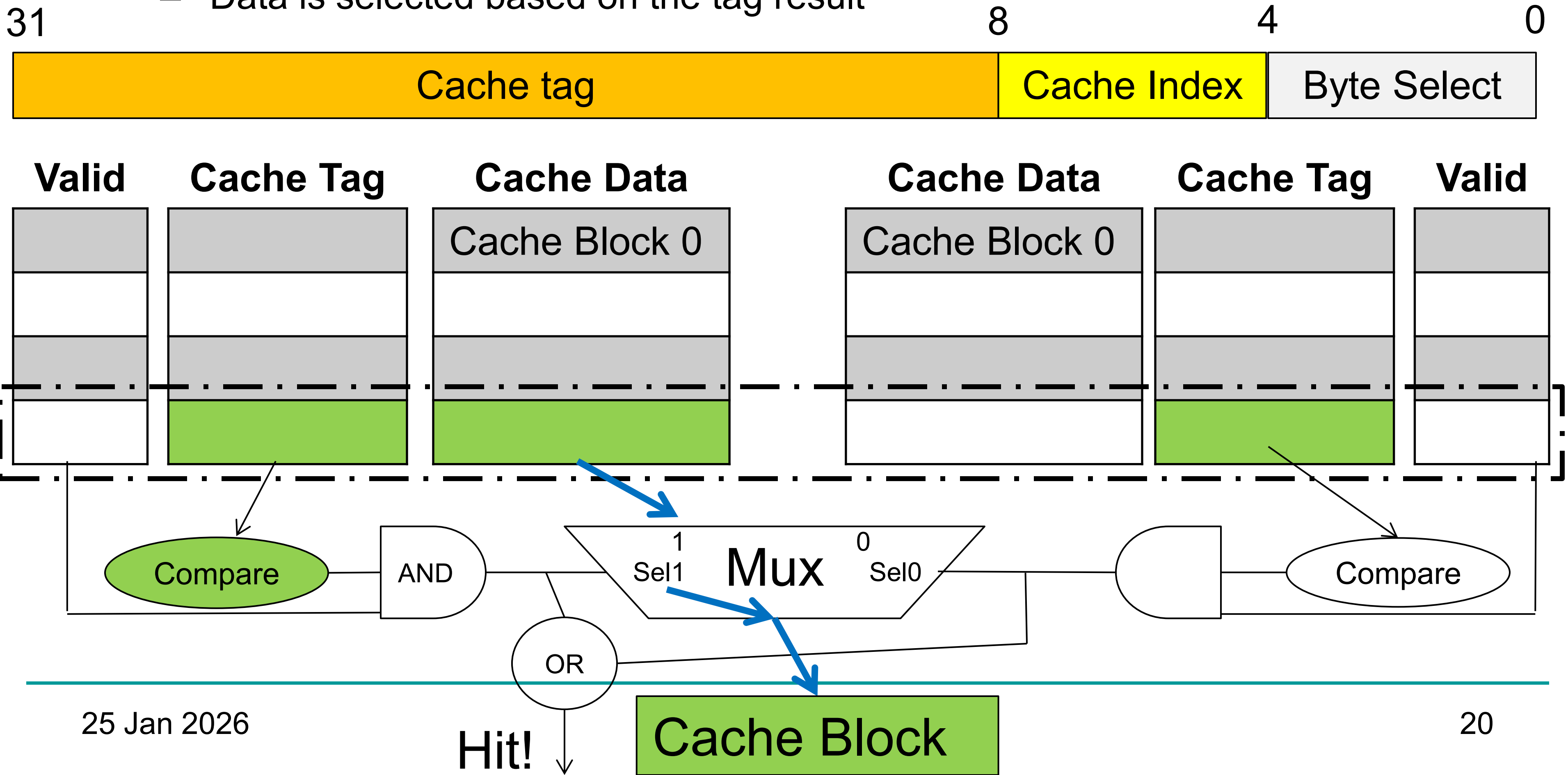
Set Associative Cache

- *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



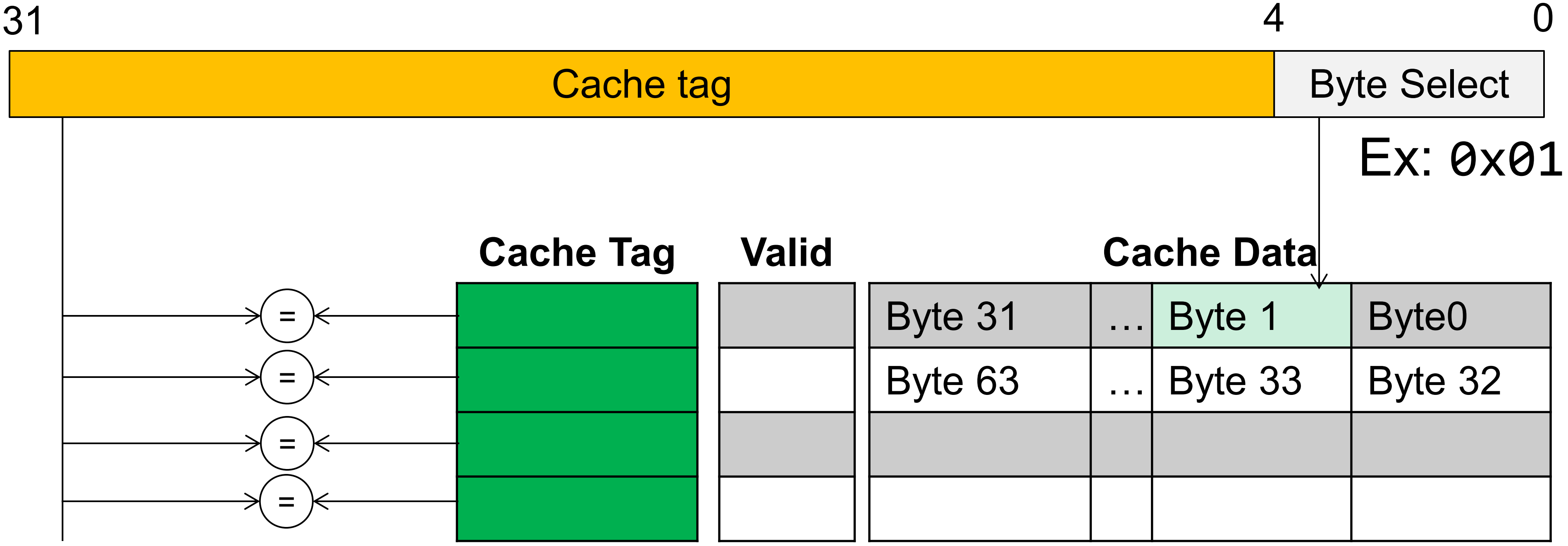
Set Associative Cache

- *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



Fully Associative Cache

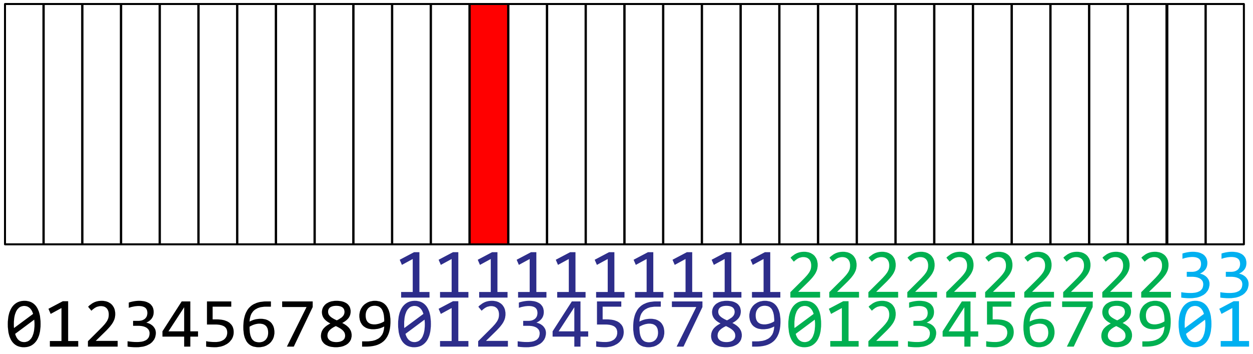
- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



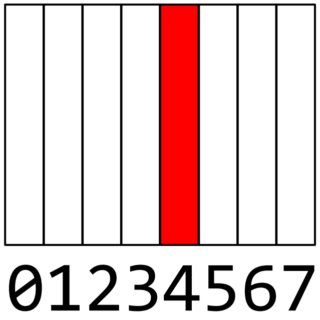
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

Block number:

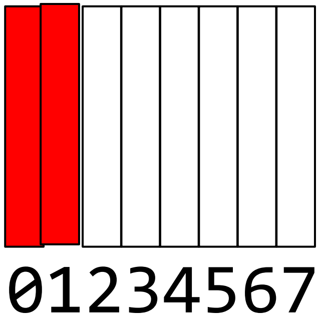


Direct mapped: Block 12 can go only into block 4 (12 mod 8)

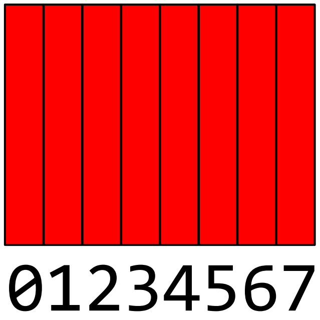


Set associative: Block 12 can go anywhere in set 0 (12 mod 4)

Set: 0 1 2 3



Fully associative: Block 12 can go anywhere



Which block to replace on a miss?

- Easy for Direct Mapped: Only one possibility

- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

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

(These are old benchmark cache miss rates)

What happens on a write?

Write Through

- The information is written to both the block in the cache and to the block in the lower-level memory
- 👍
  - Read misses cannot result in writes
- 👎
  - Processor held up on writes unless writes buffered

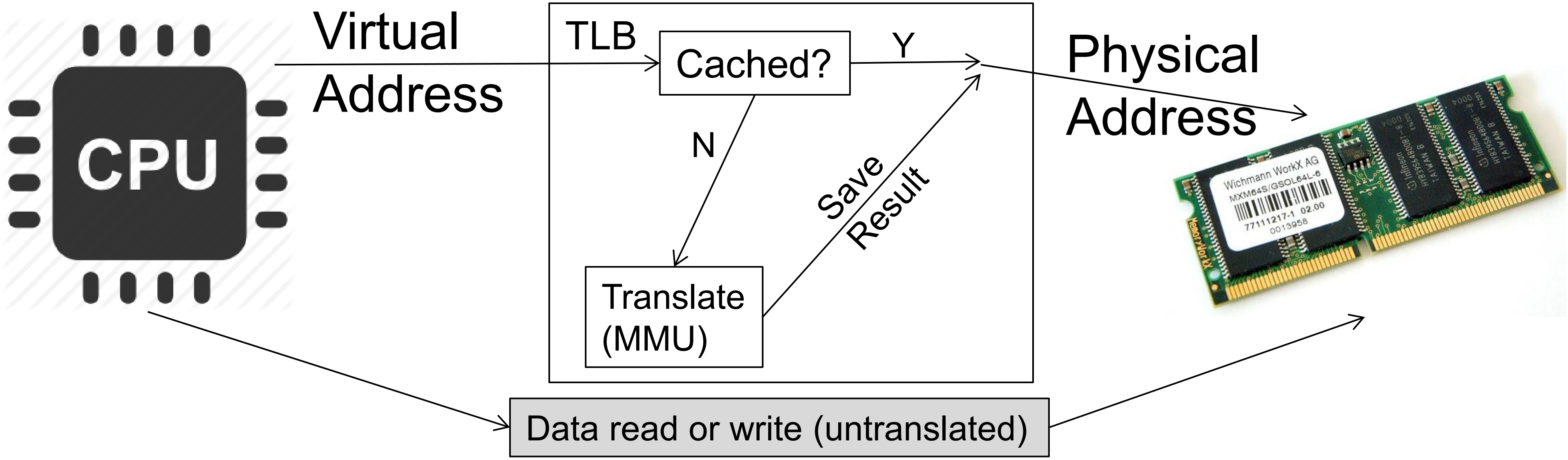
Write Back

- The information is written only to the block in the cache.
  - Modified cache block is written to main memory only when it is replaced
  - Question: Is block clean or dirty?
- 👍
  - repeated writes not sent to DRAM
  - processor not held up on writes
- 👎
  - More complex
  - Read miss may require writeback of dirty data

So Far

- Caching Basics
- Caching on Address Translations (TLB)
- Demand Paging

Caching Applied to Address Translation



Virtual Address

TLB

Cached?

Y

N

Save Result

Translate (MMU)

Physical Address

Data read or write (untranslated)

- 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

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 (like MIPS)

- 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

What Actually Happens on a TLB Miss?

- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things

- Examples:
  - Shared segments
  - User-level portions of an operating system

Transparent Exceptions: TLB/Page fault

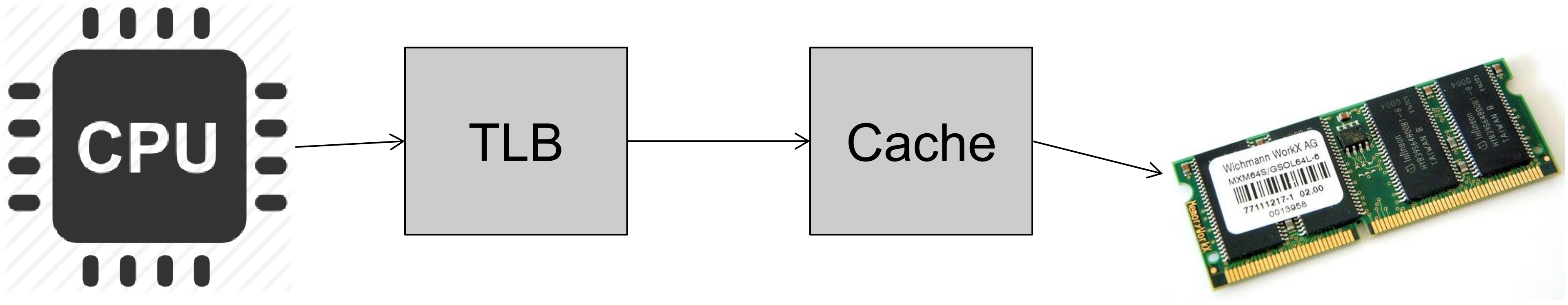
- How to transparently restart faulting instructions?
  - (Consider load or store that gets TLB or Page fault)
  - Could we just skip faulting instruction?
    - No: need to perform load or store after reconnecting physical page
- Hardware must help out by saving:
  - Faulting instruction and partial state
    - Need to know which instruction caused fault
    - Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    - Save/restore registers, stack, etc
- What if an instruction has side-effects?

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!
- What can we do?
  - Invalidate TLB: simple but might be expensive
    - What if switching frequently between processes?
  - Include ProcessID in TLB
    - This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - Otherwise, might think that page is still in memory!
  - Called "TLB Consistency"

What TLB organization makes sense?

- Needs to be really fast
  - Critical path of memory access
    - In simplest view: before the cache
    - Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - Thus the cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)



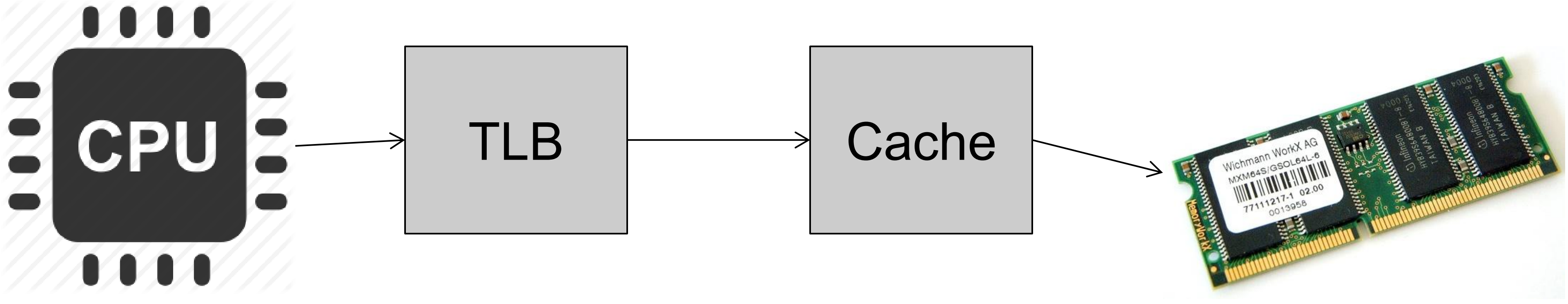
CPU

TLB

Cache

What TLB organization makes sense?

- Thrashing: Continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - First page of code, data, stack may map to same entry
    - Need 3-way associativity at least?
  - What if use high order bits as index?
    - TLB mostly unused for small programs



TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity

- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info

- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"

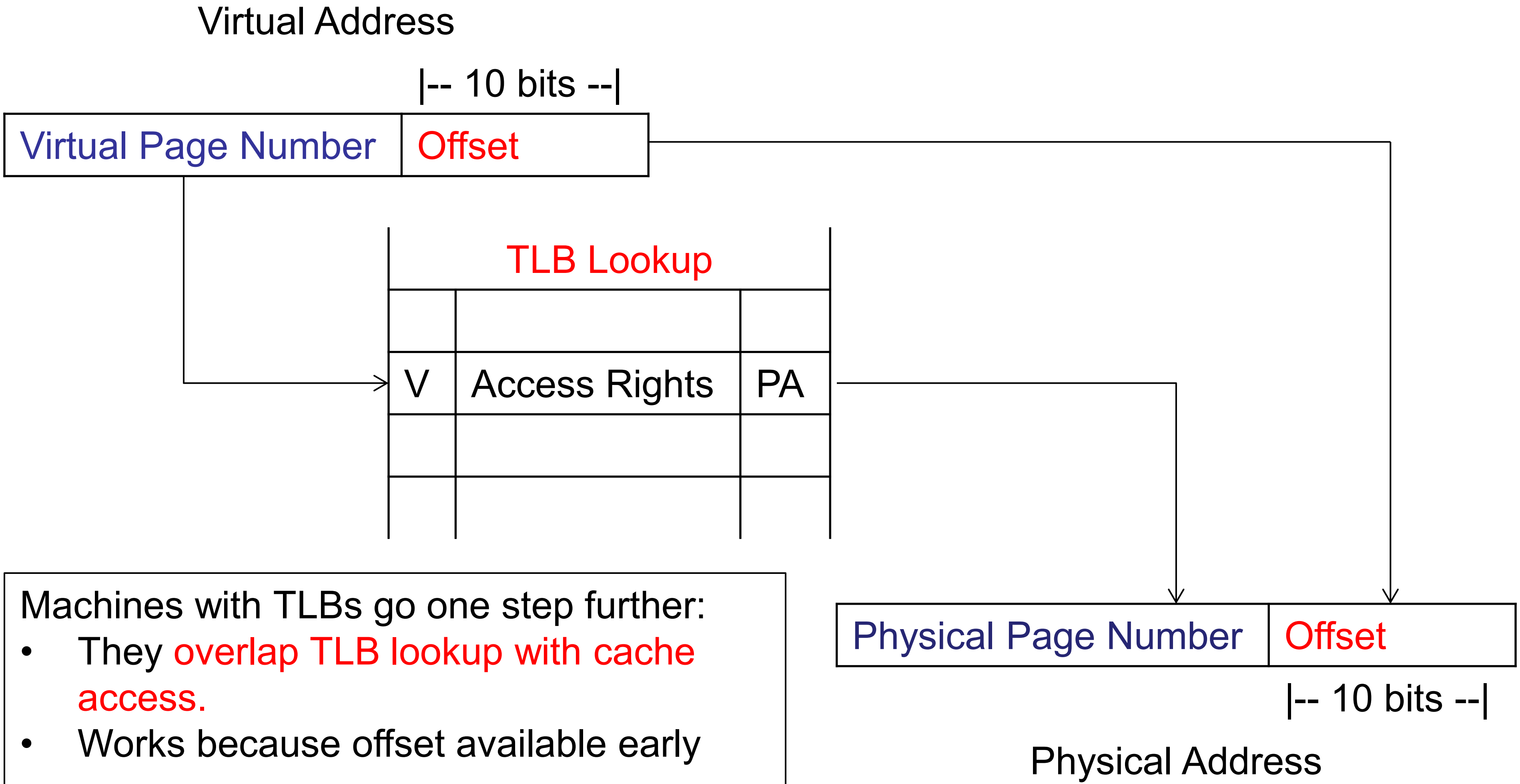
TLB organization: include protection

- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty?	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0

Reducing translation time further

- As described, TLB lookup is in serial with cache lookup:



Virtual Address

|-- 10 bits --|

Virtual Page Number | Offset

TLB Lookup

V | Access Rights | PA

Physical Page Number | Offset

|-- 10 bits --|

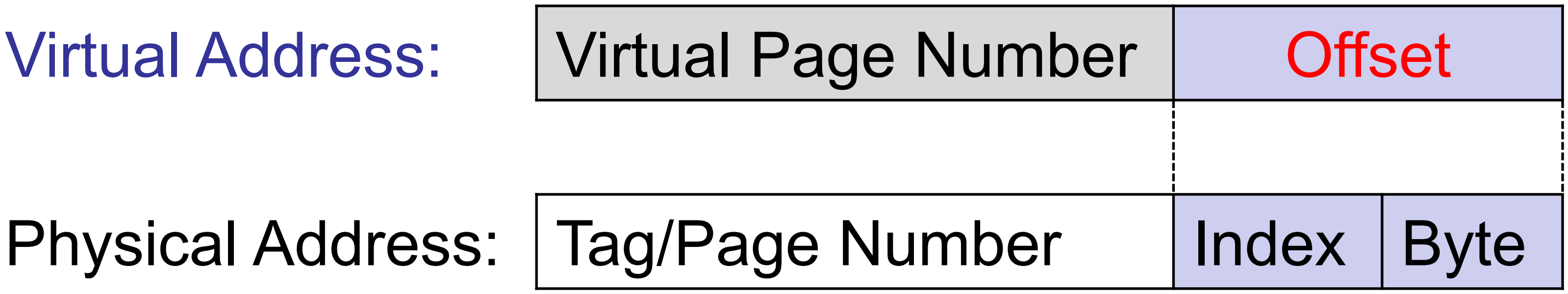
Physical Address

Machines with TLBs go one step further:

- They overlap TLB lookup with cache access.
- Works because offset available early

Overlapping TLB & Cache Access

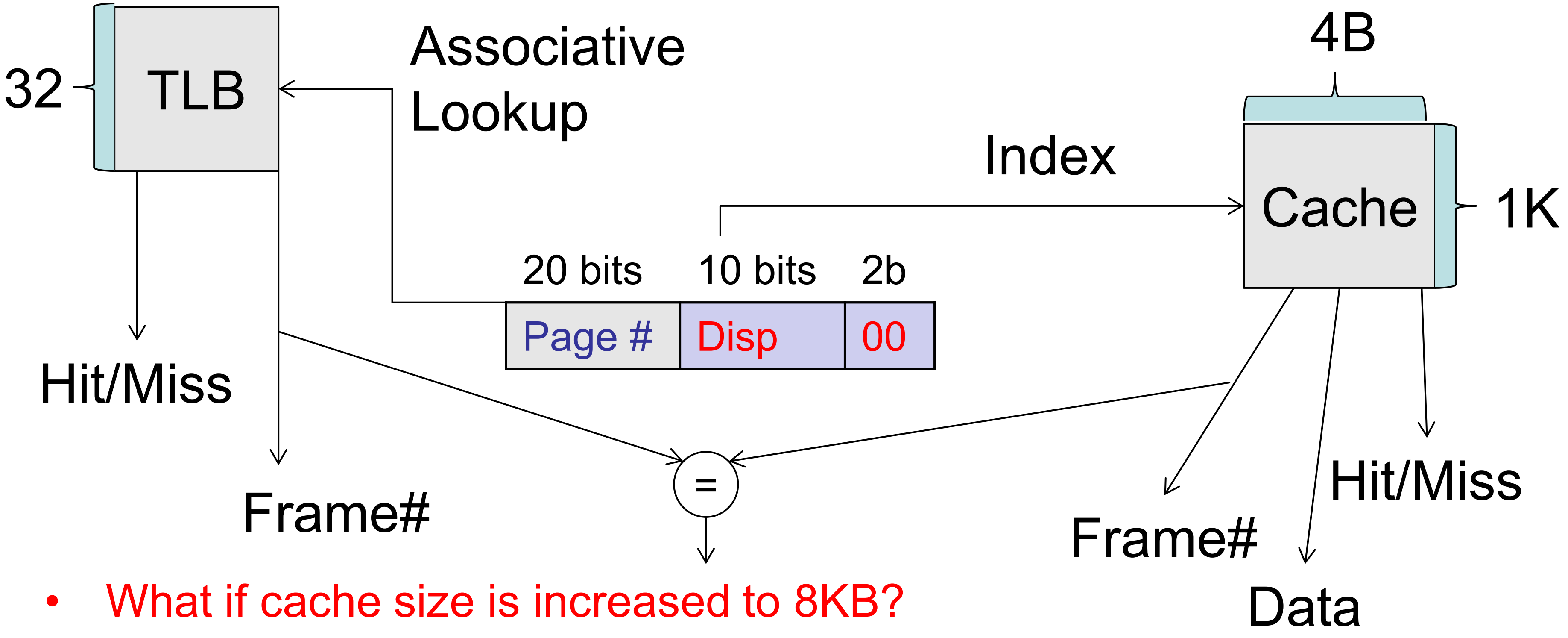
- Main idea:
  - Offset in virtual address exactly covers the “cache index” and “byte select”
  - Thus can select the cached byte(s) in parallel to perform address translation



Virtual Address: Virtual Page Number | Offset

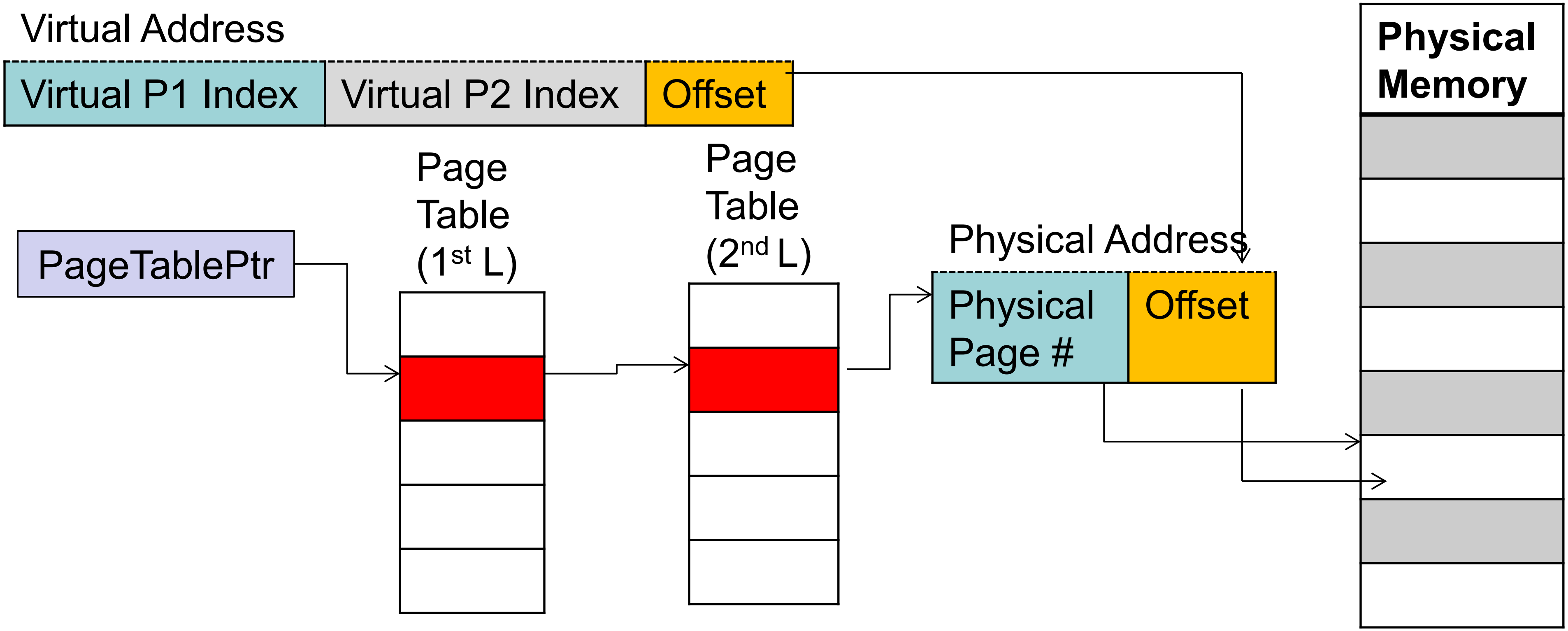
Physical Address: Tag/Page Number | Index | Byte

Overlapping TLB & Cache Access: 4KB Cache

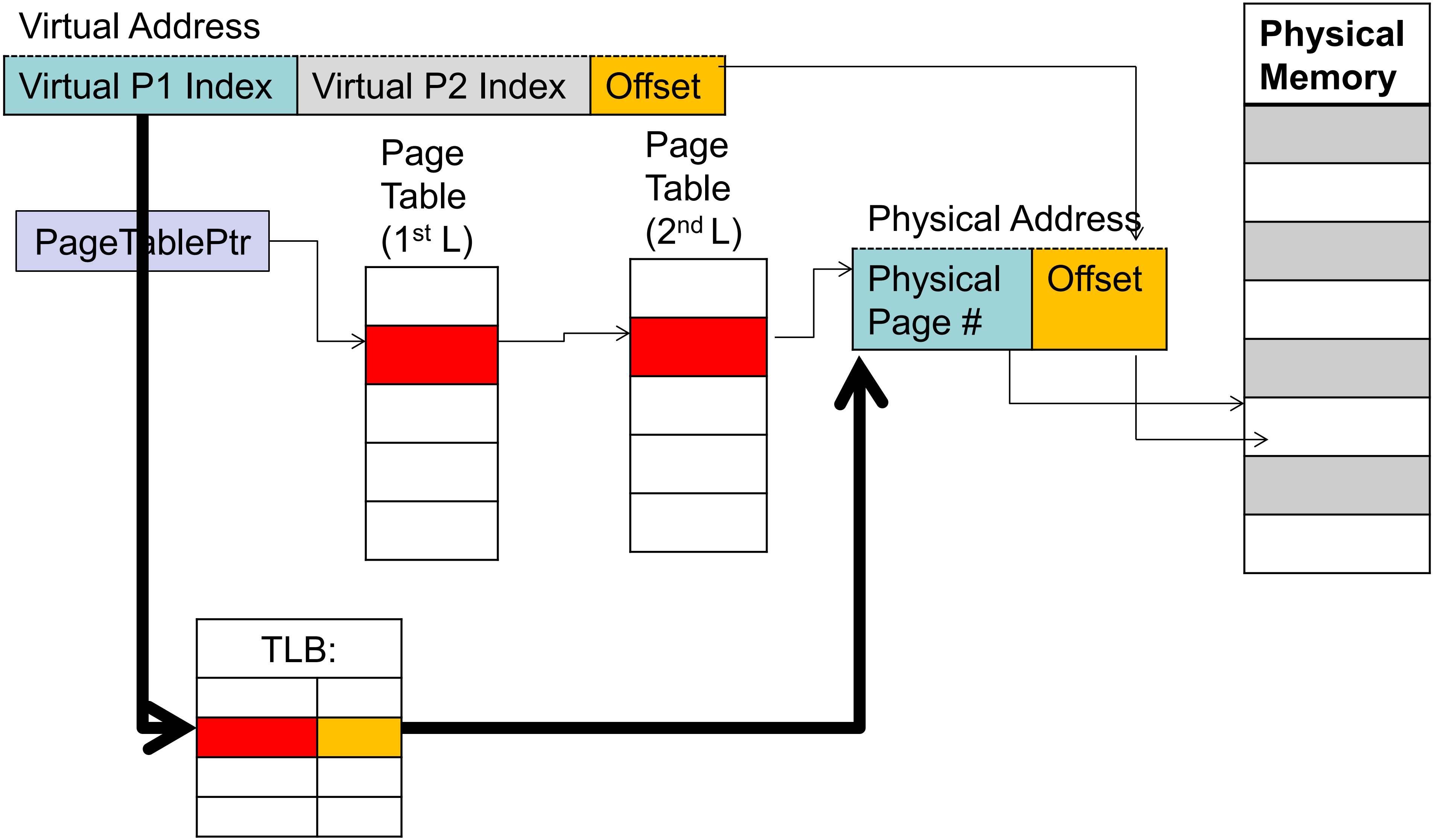


- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See מבנה המחשב
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

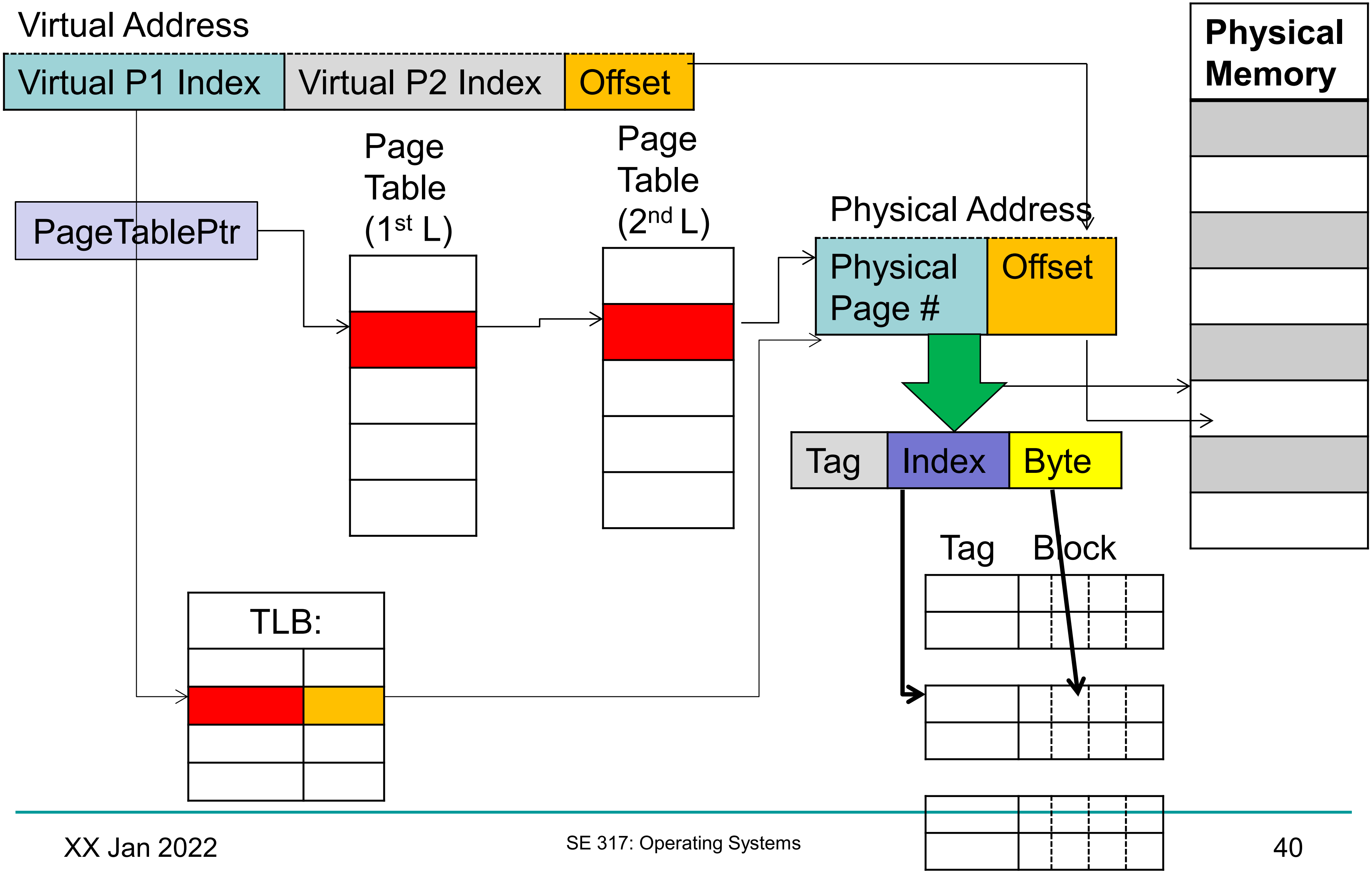
Putting Everything Together: Address Translation



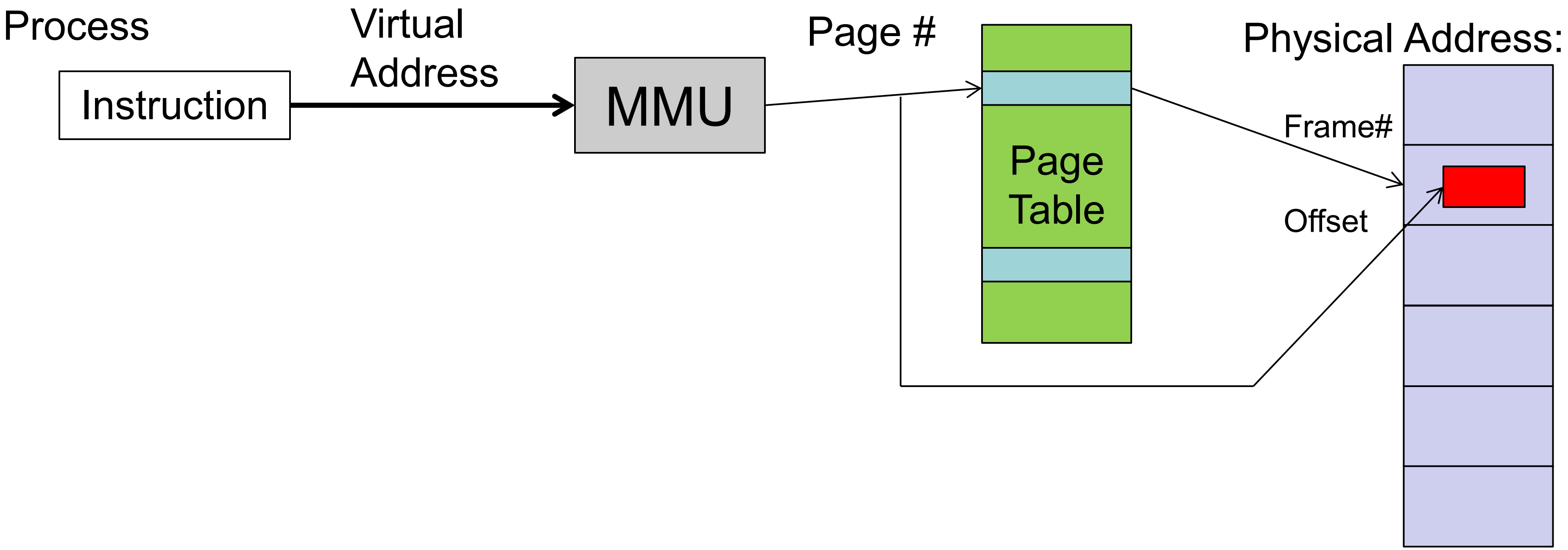
Putting Everything Together: TLB



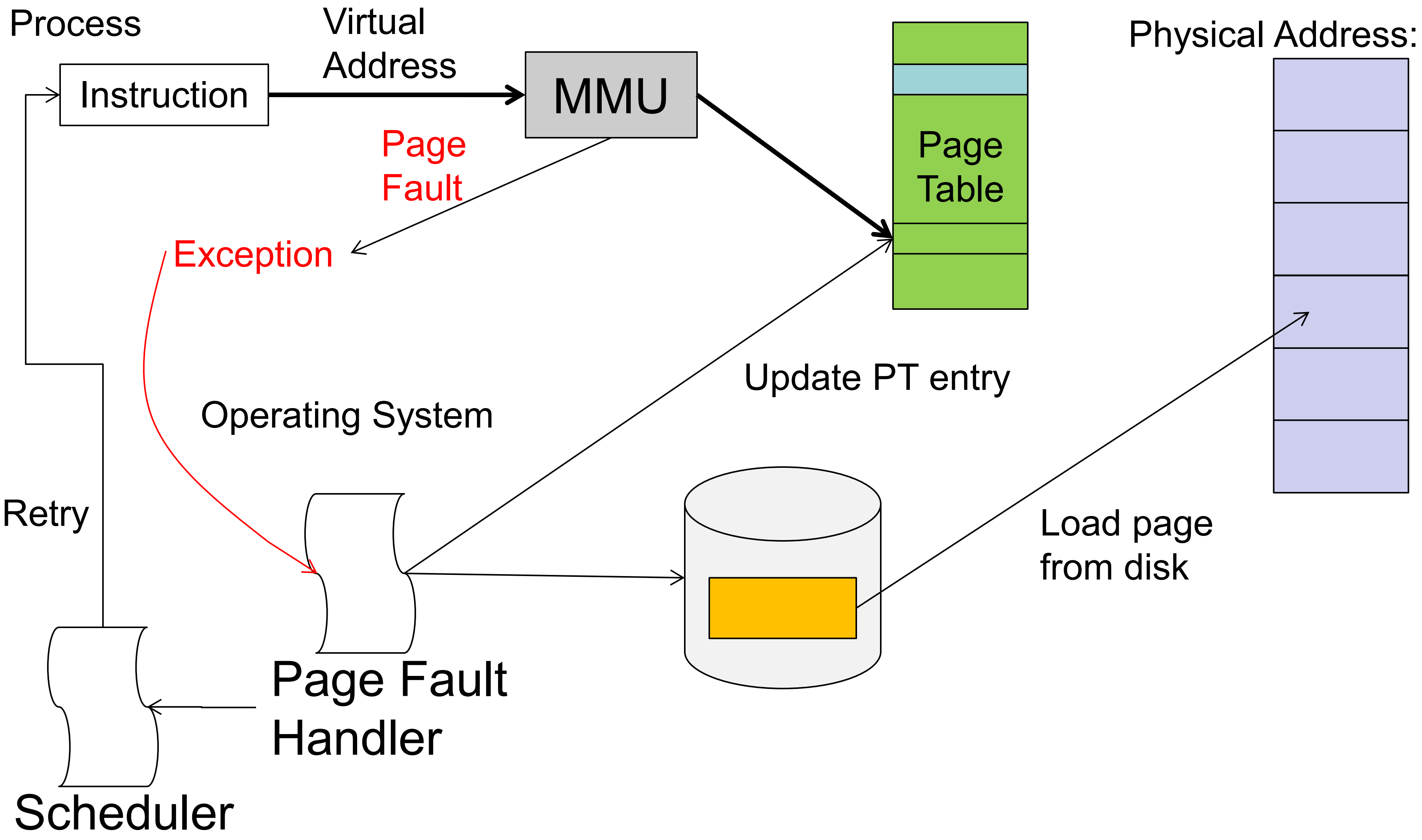
Putting Everything Together: Cache



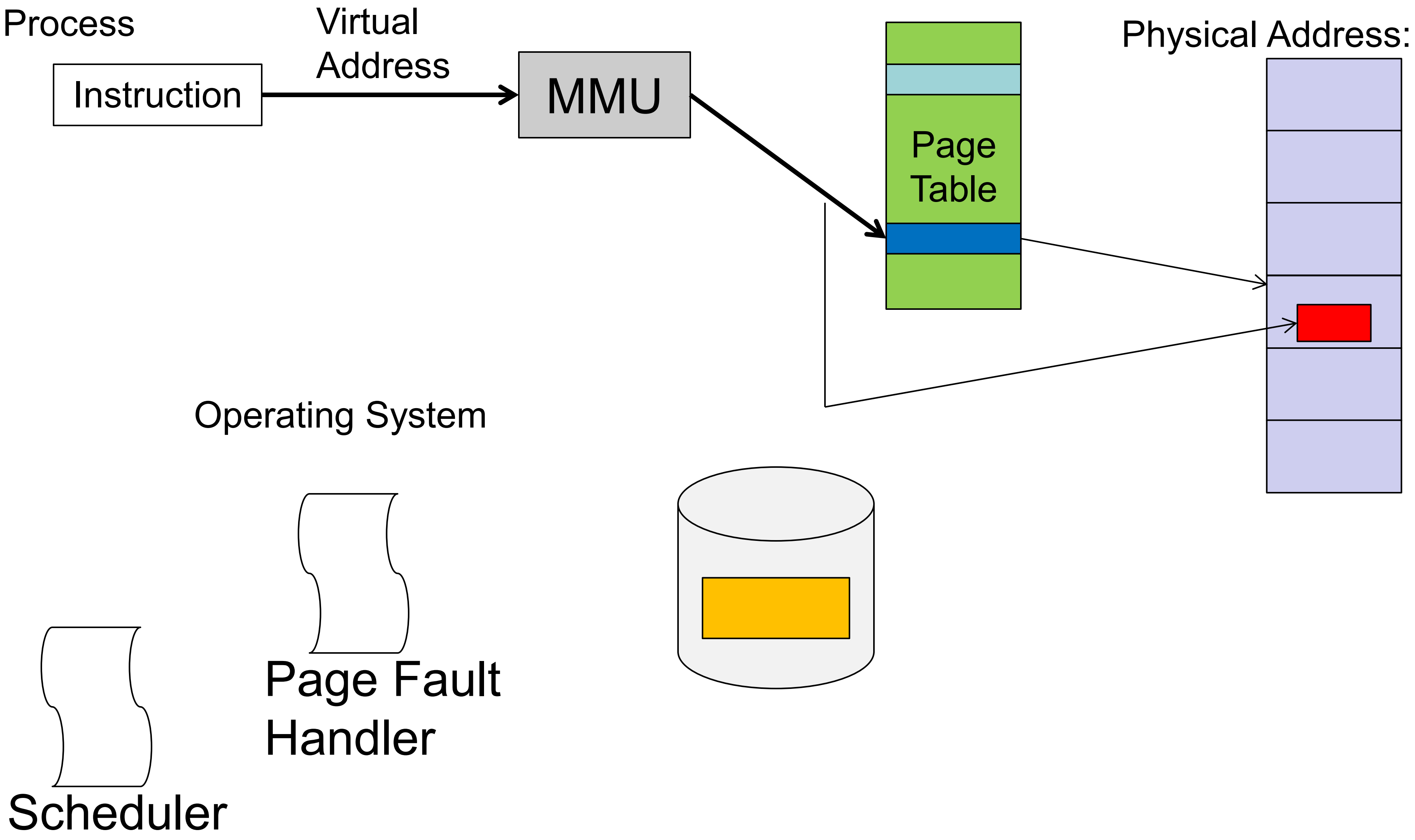
What happens when …



What happens when …



What happens when …



So Far

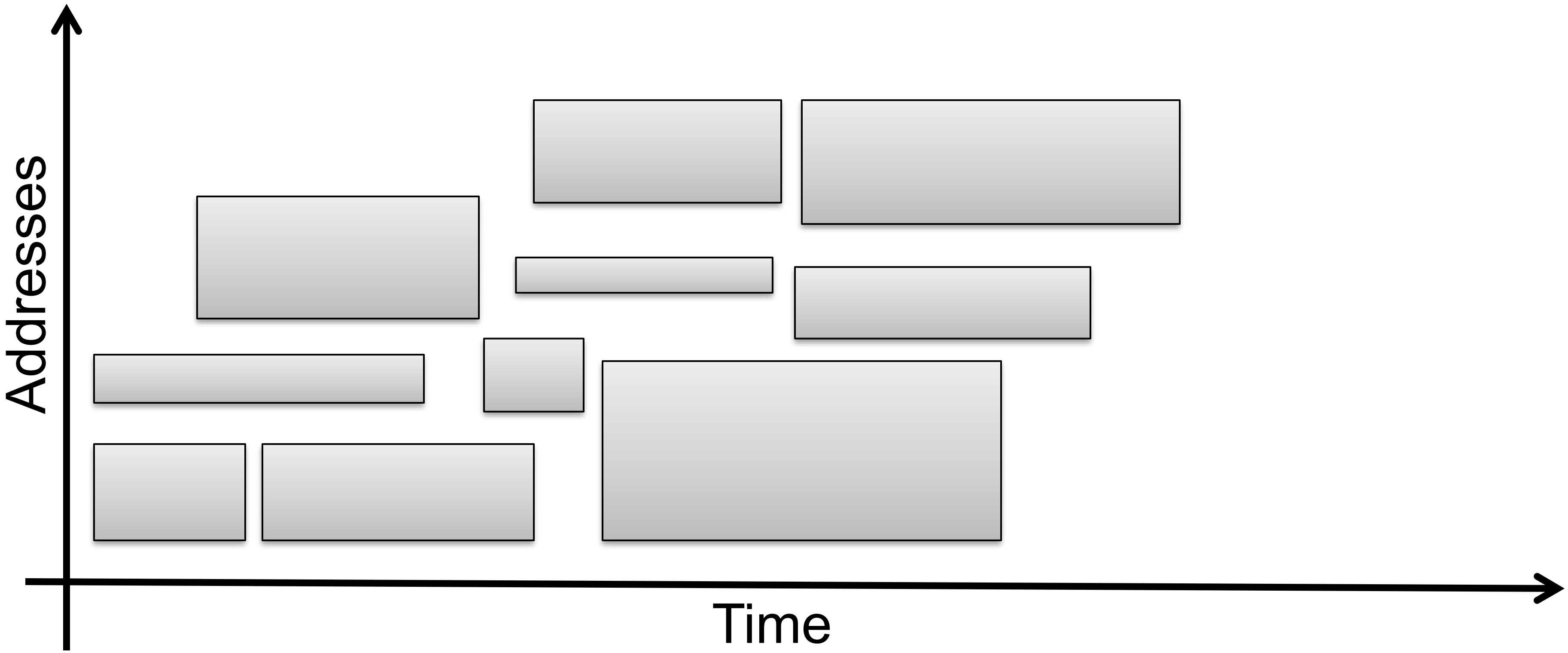
- Caching Basics
- Caching on Address Translations (TLB)
- Demand Paging

Impact of caches on Operating Systems

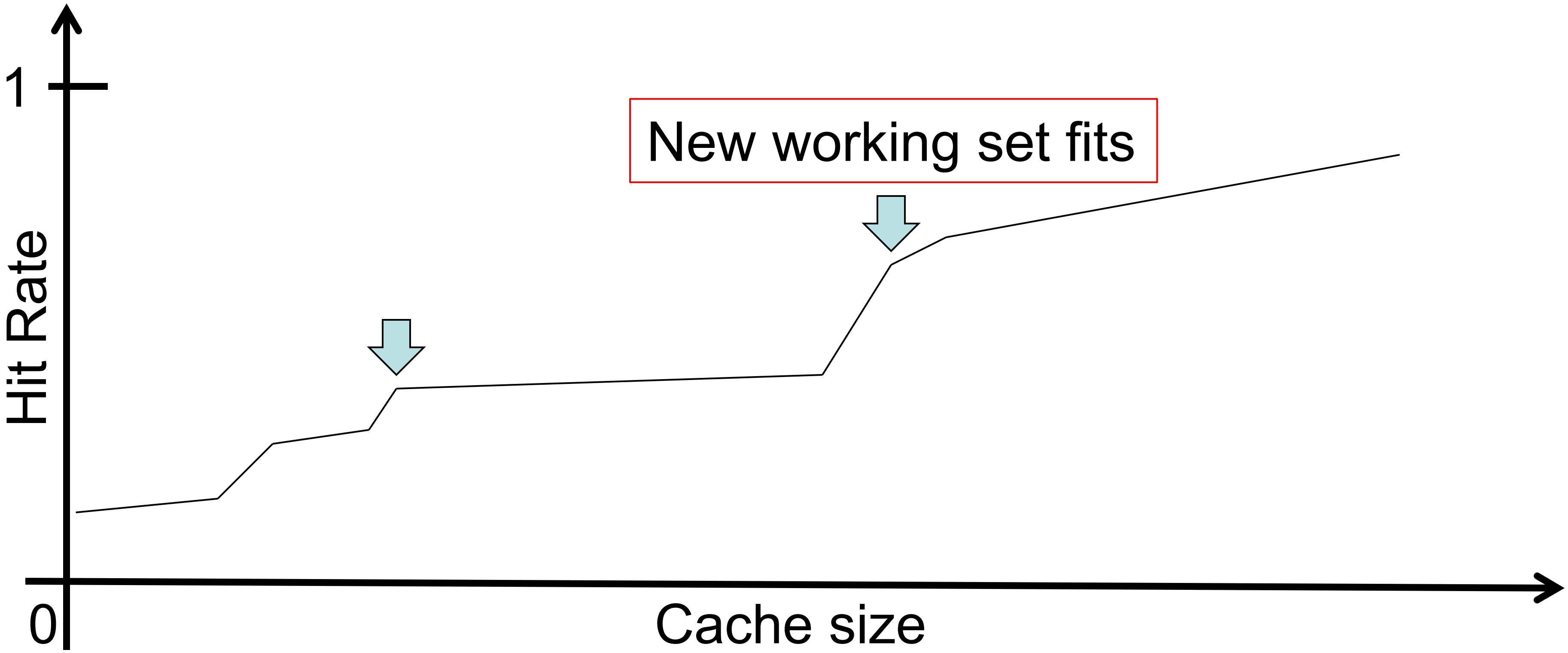
- Indirect - dealing with cache effects
- Process scheduling
  - Which and how many processes are active?
  - Large memory footprints versus small ones?
  - Priorities?
  - Shared pages mapped into Virtual Address Space (VAS) of multiple processes?
- Impact of thread scheduling on cache performance
  - Rapid interleaving of threads (small quantum) may degrade cache performance
    - Increase Average Memory Access Time (AMAT)
- Designing OS data structures for cache performance
- Maintaining the correctness of various caches
  - TLB consistency:
    - With PT across context switches ?
    - Across updates to the PT ?

Working Set Model

- As a program executes it transitions through a sequence of “working sets” consisting of varying sized subsets of the address space

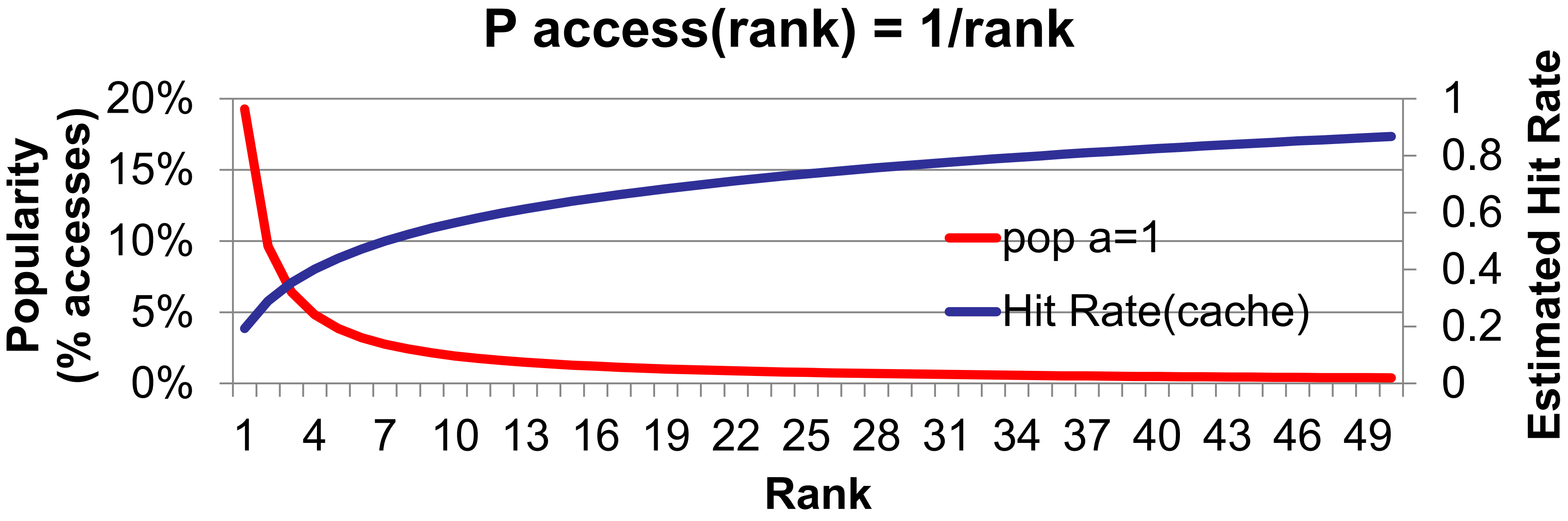


Cache Behavior under WS model



- Amortized by fraction of time the WS is active
- Transitions from one WS to the next
- Capacity, Conflict, Compulsory misses
- Applicable to memory caches and pages. Others ?

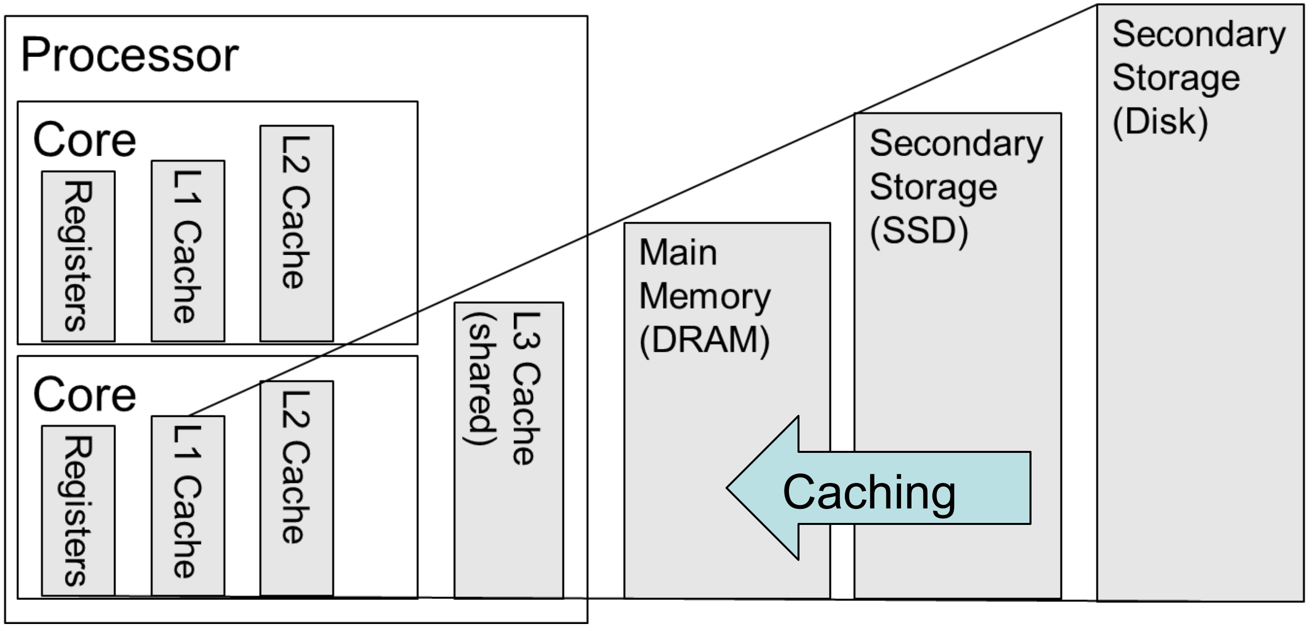
Another model of Locality: Zipf



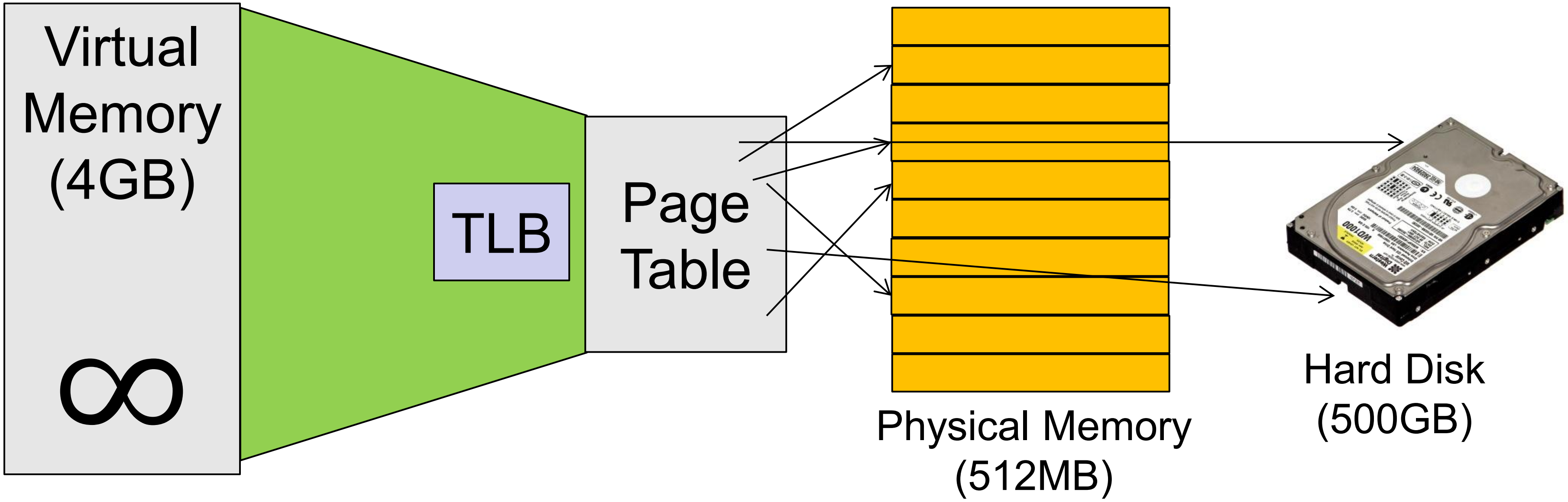
- Likelihood of accessing item of rank r is $1/r^{\alpha}$
- Although rare to access items below the top few, there are so many that it yields a “heavy tailed” distribution.
- Substantial value from even a tiny cache
- Substantial misses from even a very large one

Demand Paging

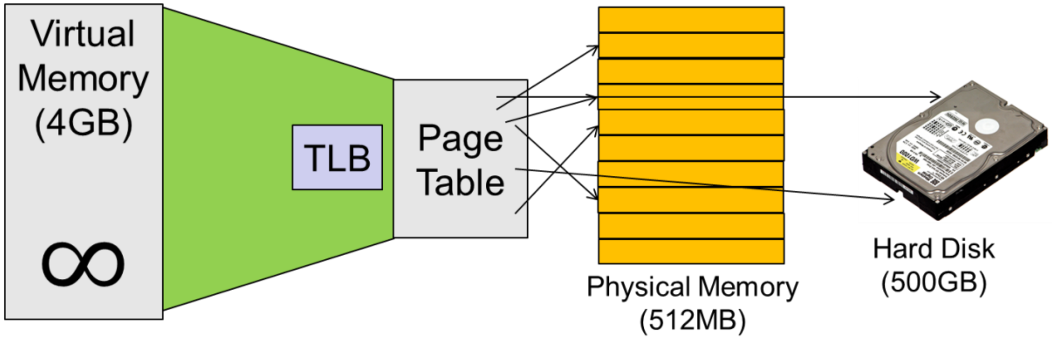
- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: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



Illusion of Infinite Memory



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
- 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

- So we 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→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… (kind of like LRU)
  - 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!

What is in a Page Table Entry? (Review)

- 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 in 10, 10, 12-bit offset format
  - 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

What is in a Page Table Entry? (Review)

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

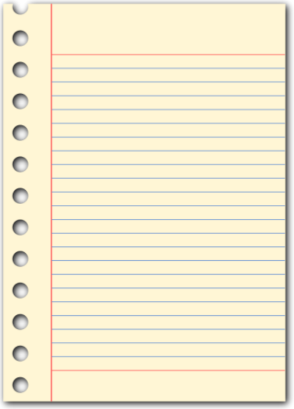
D: Dirty (PTE only): page has been modified recently

L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

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"



Demand Paging Mechanisms

- 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 the old 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

Some follow-up 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
  - As a last resort, evict a dirty page first

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

- How many page frames per process?
  - Like thread scheduling, need to “schedule” memory resources:
    - Utilization? Fairness? Priority?
  - allocation of disk paging bandwidth

Demand Paging Cost Model

- Demand Paging is like caching, so compute Effective Access Time
  - $EAT = Hit\ Rate \times Hit\ Time + Miss\ Rate \times Miss\ Time$
  - $EAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$

Example:

- Memory access time = $200ns$
- Average page-fault service time = $8ms$
- Let $p = \ Probability\ of\ miss,\ 1 - p = Probably\ of\ hit$
- We compute EAT as follows:

$$EAT = 200ns + p \times 8ms$$
$$= 200ns + p \times \ 8{,}000{,}000ns$$

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

Capacity Misses

- Not enough memory. Must somehow increase size.
- How?
  - One option: Increase amount of DRAM (not quick fix!)
  - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!

What Factors Lead to Misses?

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

Conclusion

- Caching Basics
- Caching on Address Translations (TLB)
- Demand Paging