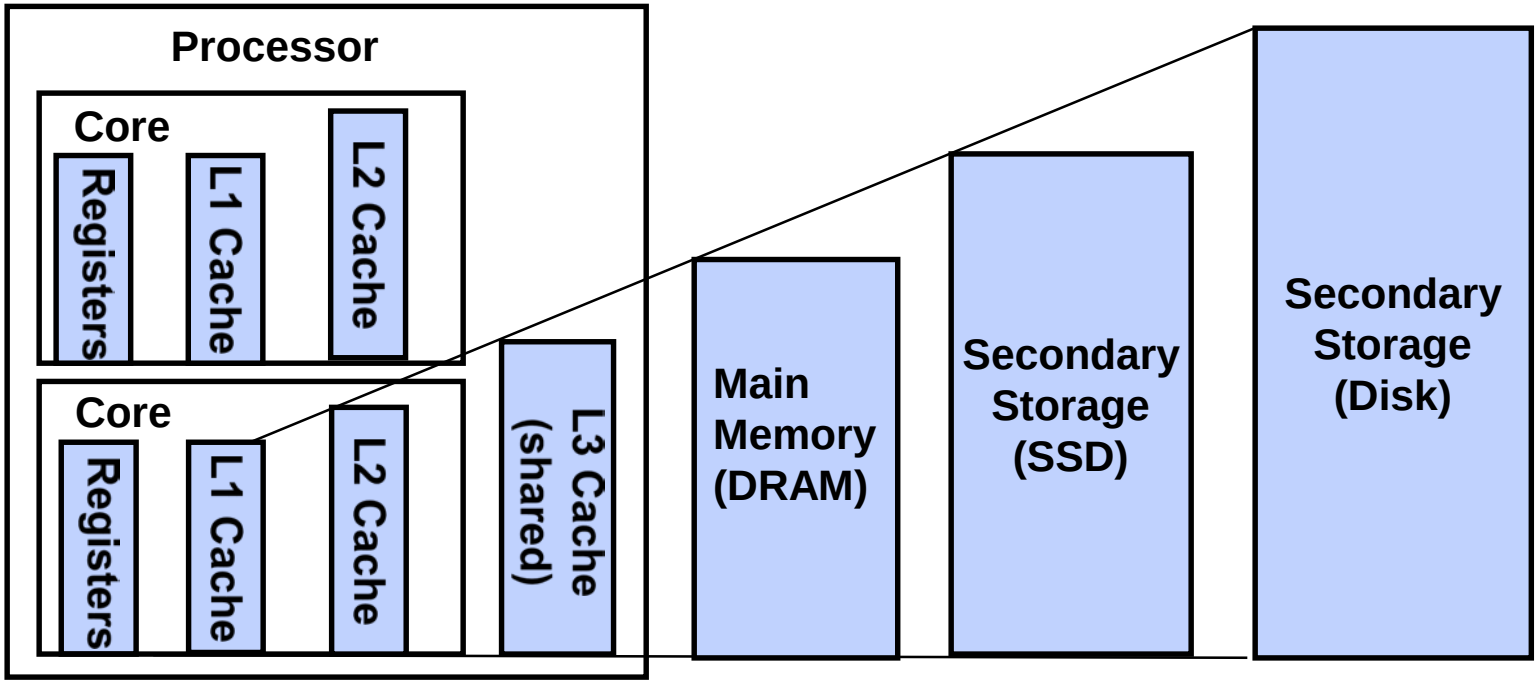
CSE150
Operating Systems
Lecture 15

Caches and TLBs

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 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

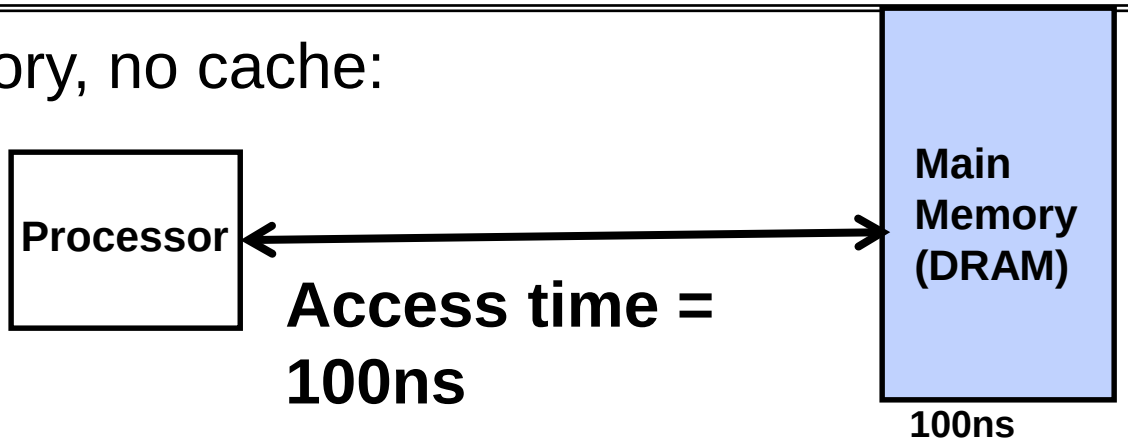
Caching Concept

- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching at different levels
  - 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 =

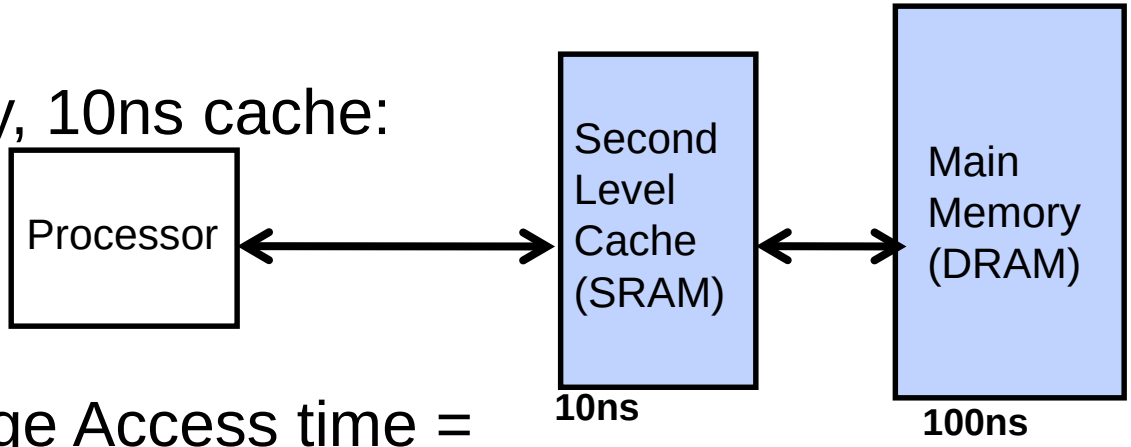
  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Example

- Data in memory, no cache:



- Data in memory, 10ns cache:

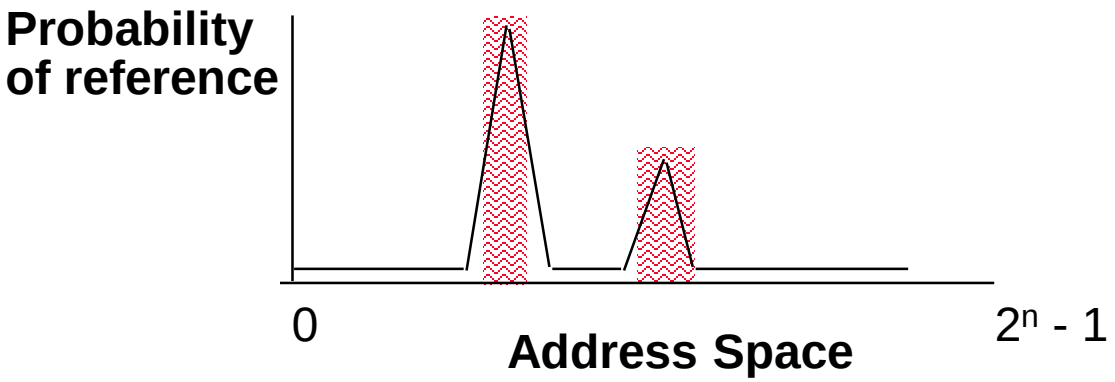


Average Access time =

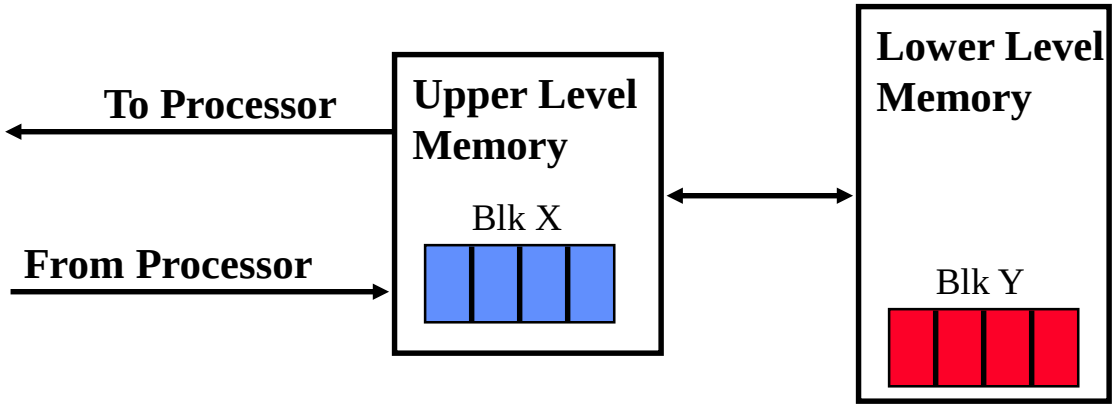
(Hit Rate x HitTime) + (Miss Rate x MissTime)

- HitRate + MissRate = 1

Why Does Caching Help? Locality!

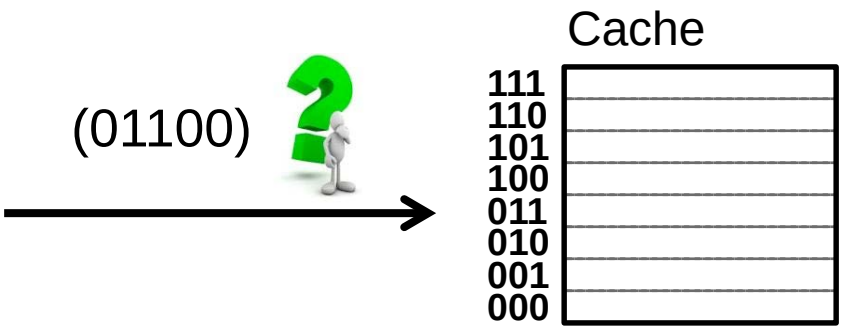


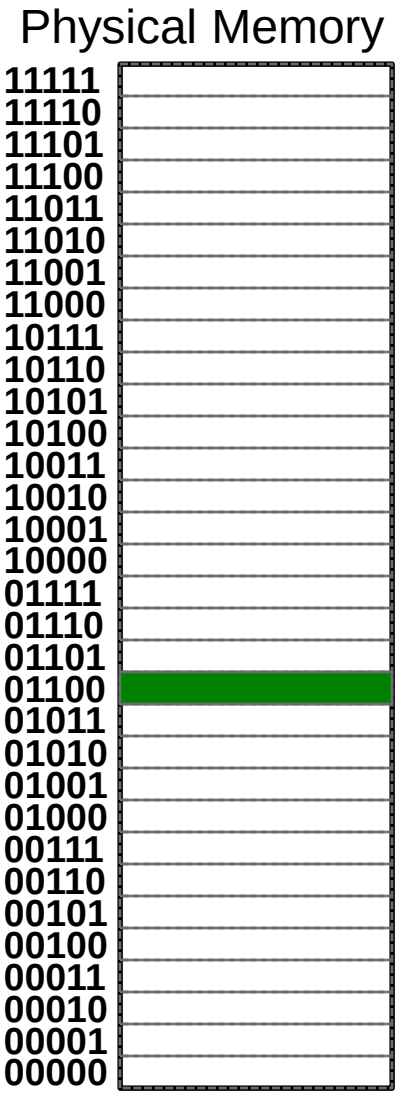
- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



Caching Questions

- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

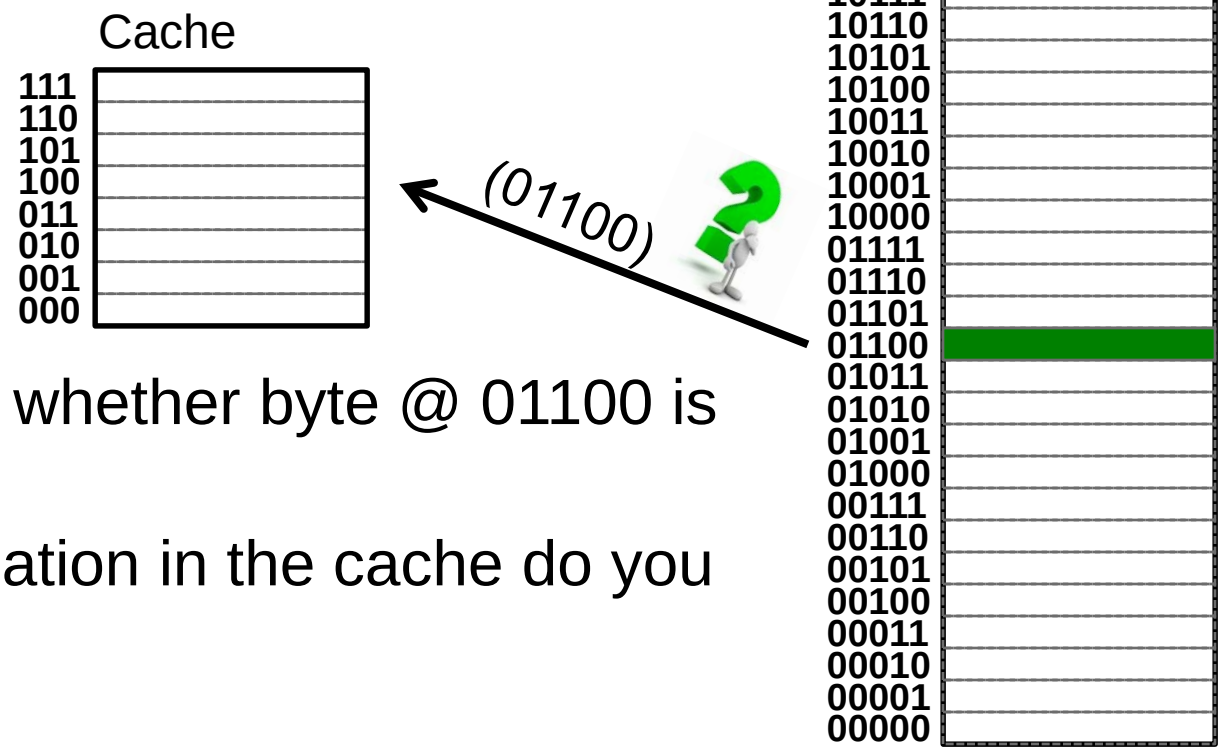




1. How do you know whether byte @ 01100 is cached?

Caching Questions

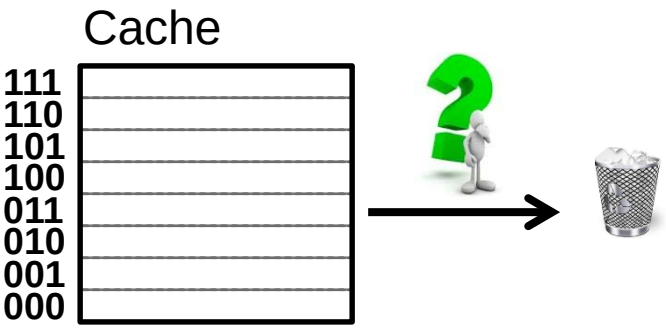
- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

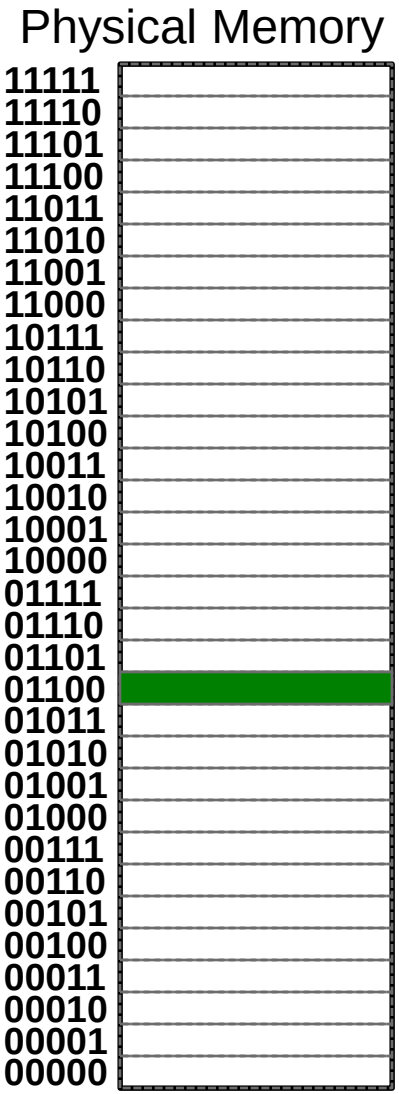


1. How do you know whether byte @ 01100 is cached?
2. If not, at which location in the cache do you place the byte?

Caching Questions

- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

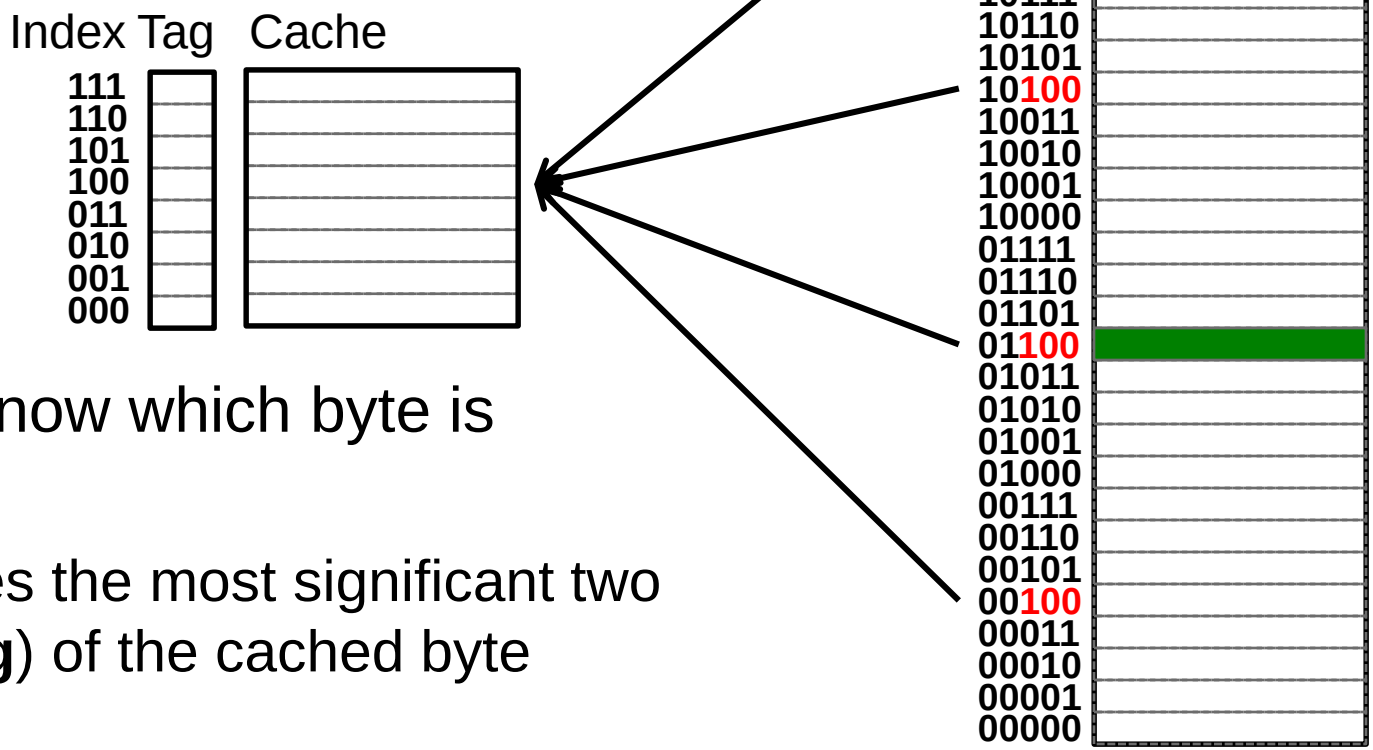




1. How do you know whether byte @ 01100 is cached?
2. If not, at which location in the cache do you place the byte?
3. If cache full, which cached byte do you evict?

Simple Example: Direct Mapped Cache

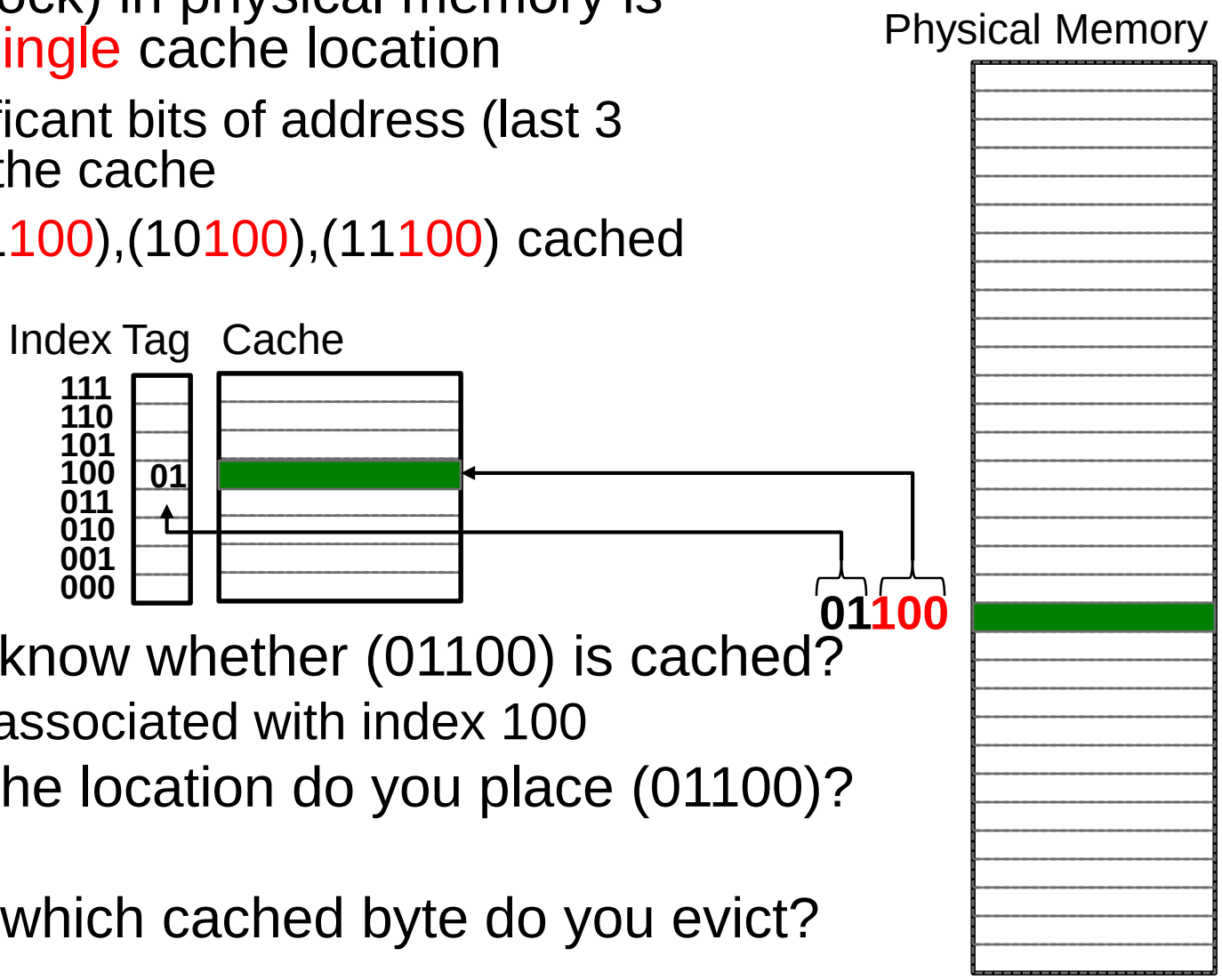
- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100),(01100),(10100),(11100) cached to 100



- How do you know which byte is cached?
  - Cache stores the most significant two bits (i.e., **tag**) of the cached byte

Simple Example: Direct Mapped Cache

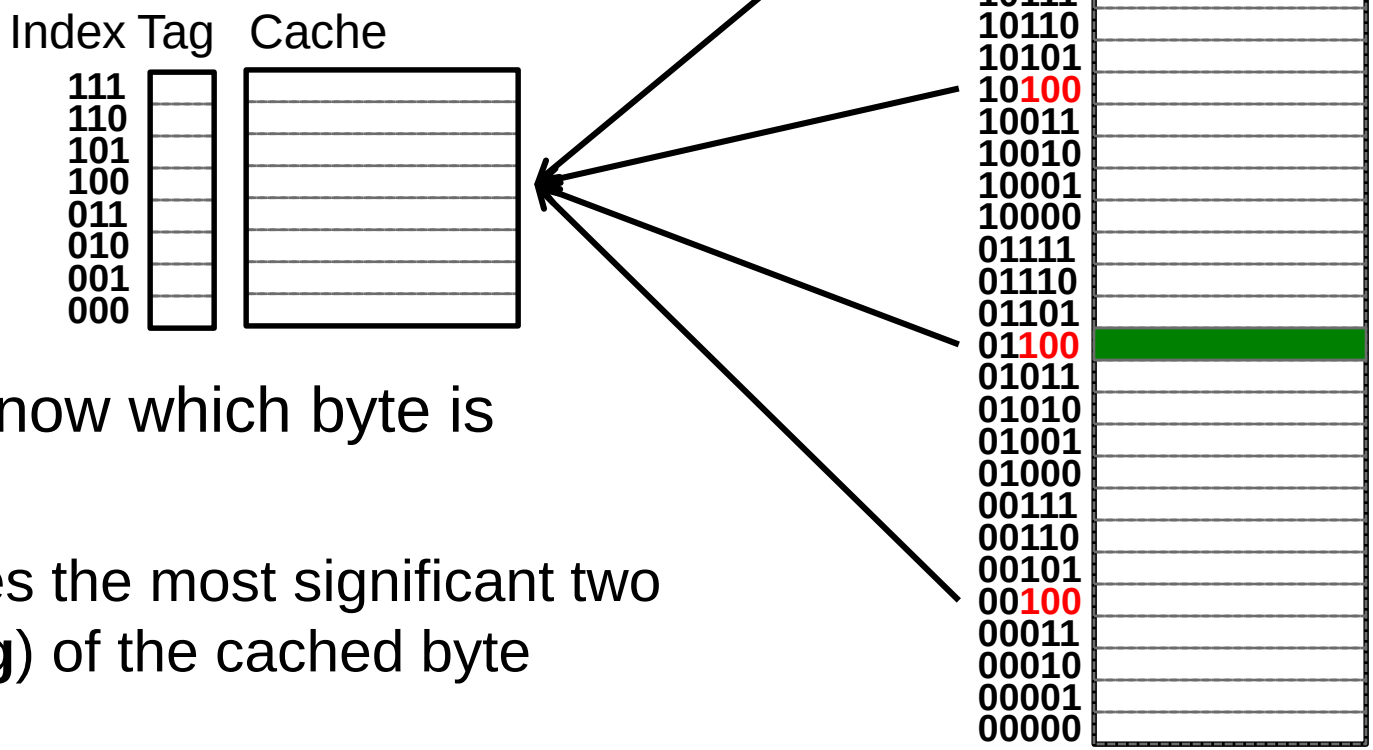
- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100),(01100),(10100),(11100) cached to 100



1. How do you know whether (01100) is cached?
   - Check **tag** associated with index 100
2. At which cache location do you place (01100)?
   - 100
3. If cache full, which cached byte do you evict?
   - 100

Simple Example: Direct Mapped Cache

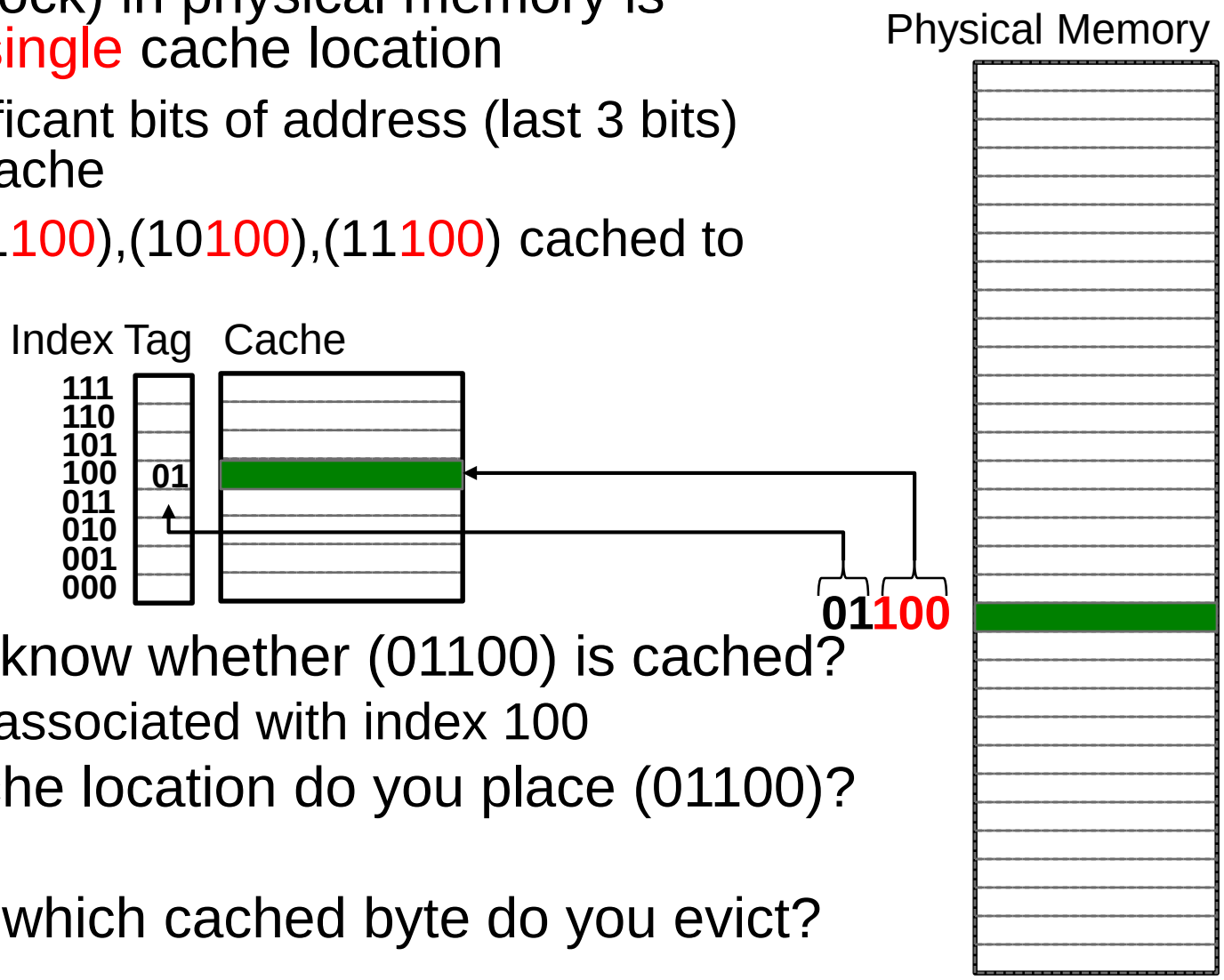
- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100),(01100),(10100),(11100) cached to 100



- How do you know which byte is cached?
  - Cache stores the most significant two bits (i.e., **tag**) of the cached byte

Simple Example: Direct Mapped Cache

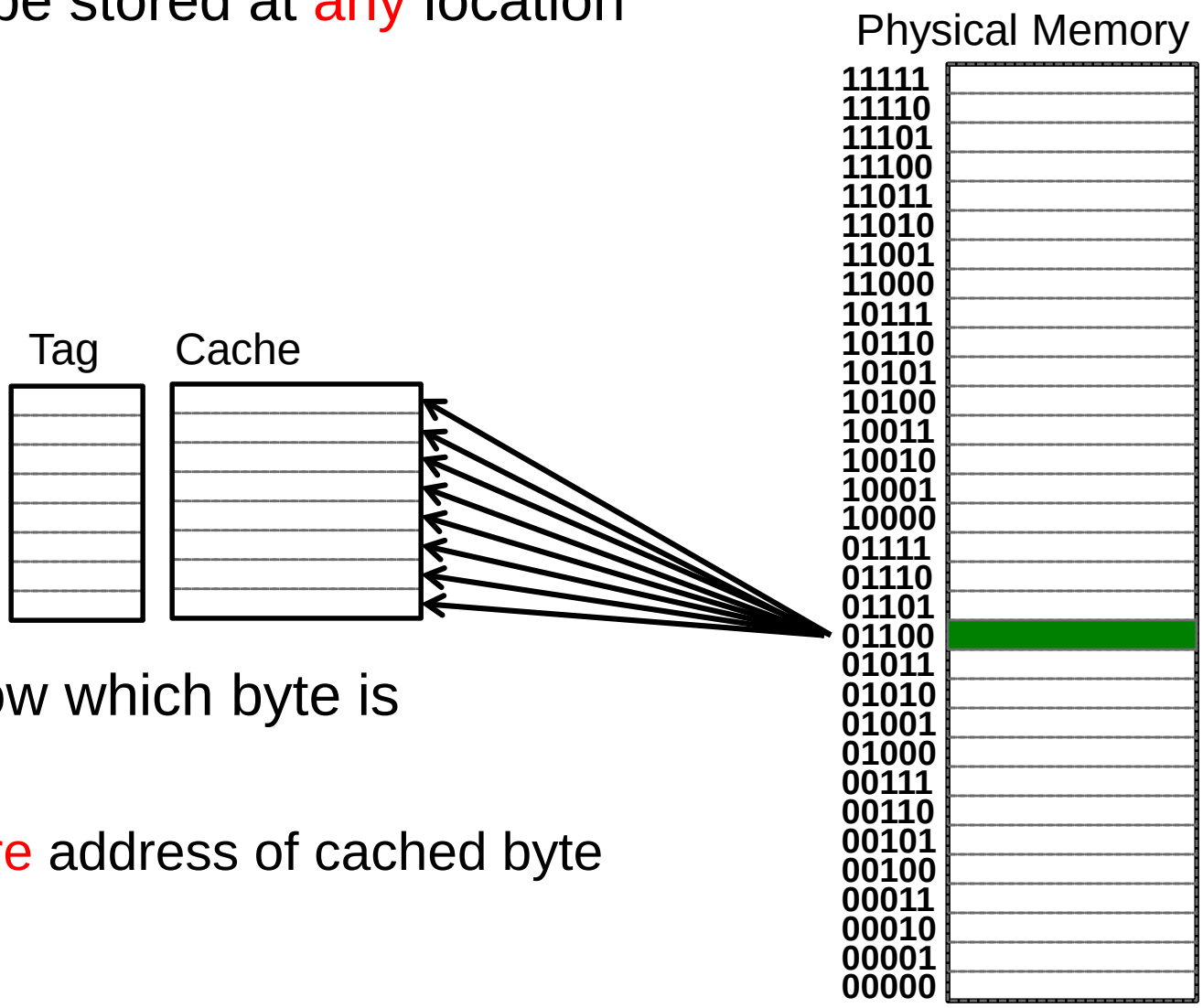
- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100),(01100),(10100),(11100) cached to 100



1. How do you know whether (01100) is cached?
   - Check **tag** associated with index 100
2. At which cache location do you place (01100)?
   - 100
3. If cache full, which cached byte do you evict?
   - 100

Simple Example: Fully Associative Cache

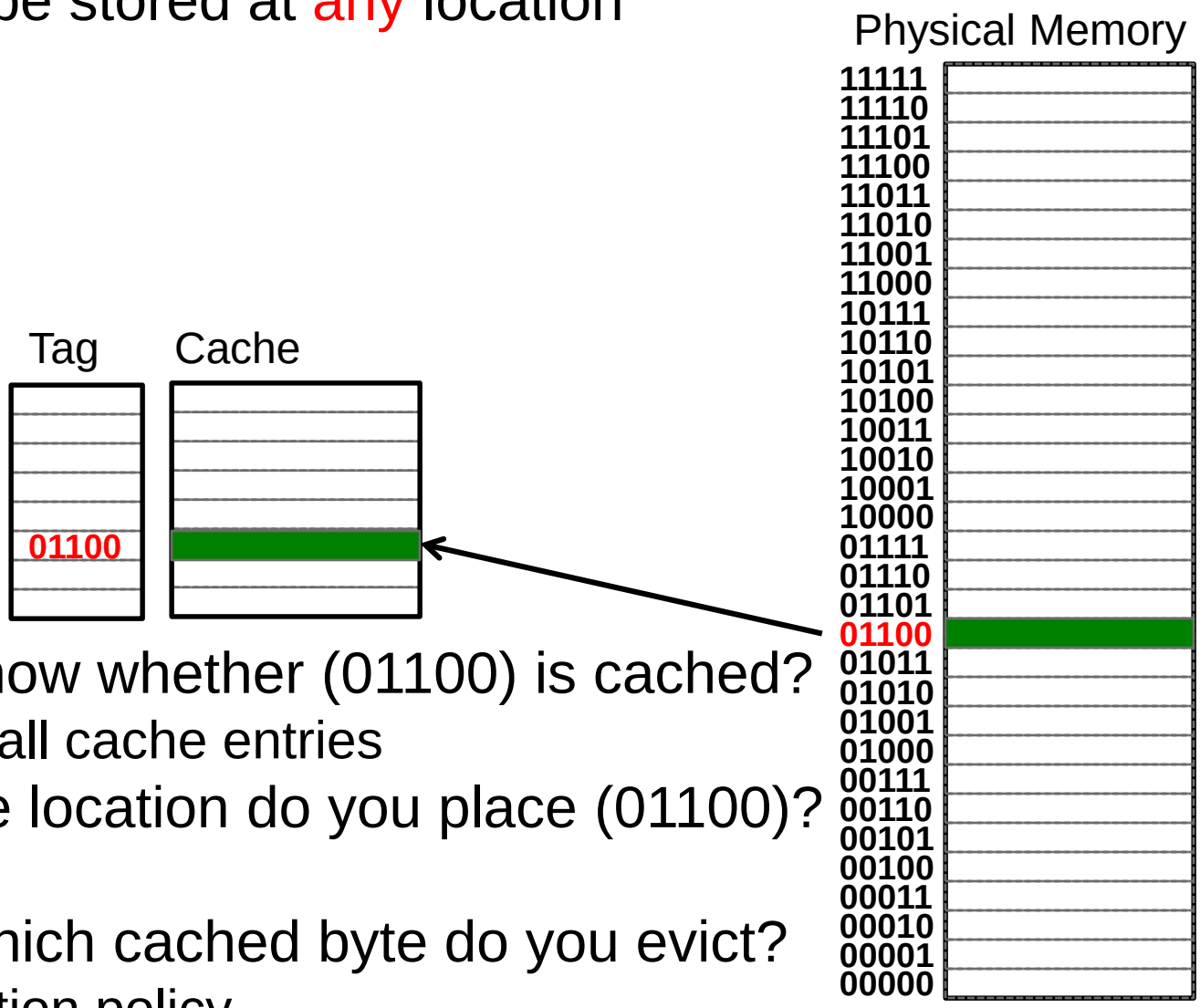
- Each byte can be stored at any location in the cache



- How do you know which byte is cached?
  - Tag store entire address of cached byte

Simple Example: Fully Associative Cache

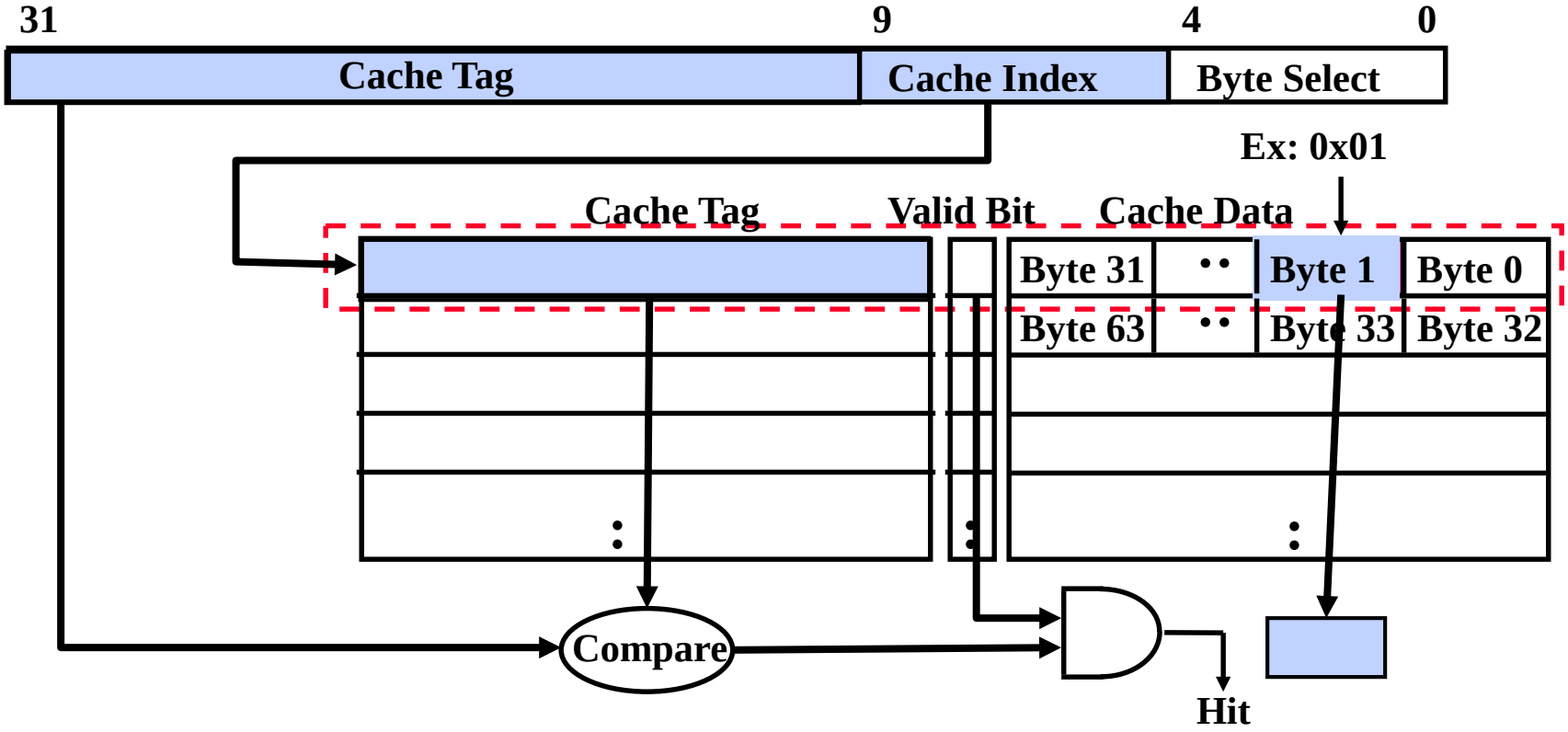
- Each byte can be stored at any location in the cache



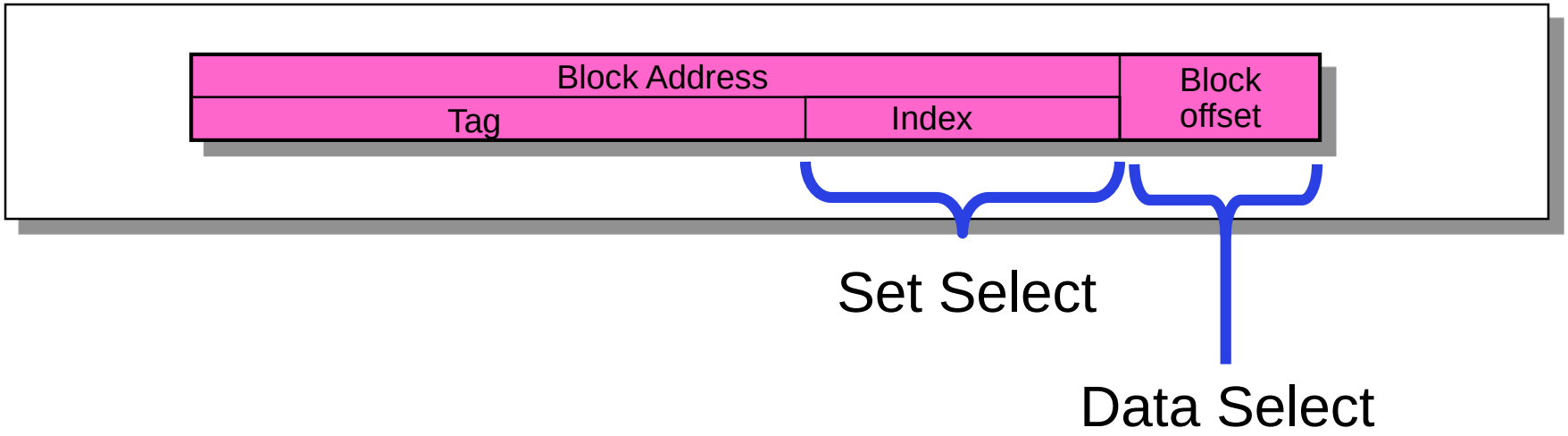
1. How do you know whether (01100) is cached?
   - Check **tag** of all cache entries
2. At which cache location do you place (01100)?
   - Any
3. If cache full, which cached byte do you evict?
   - Specific eviction policy

Direct Mapped Cache

- Cache index selects a cache block
- "Byte select" selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same "cache index" shares the same cache entry
  - Conflict misses



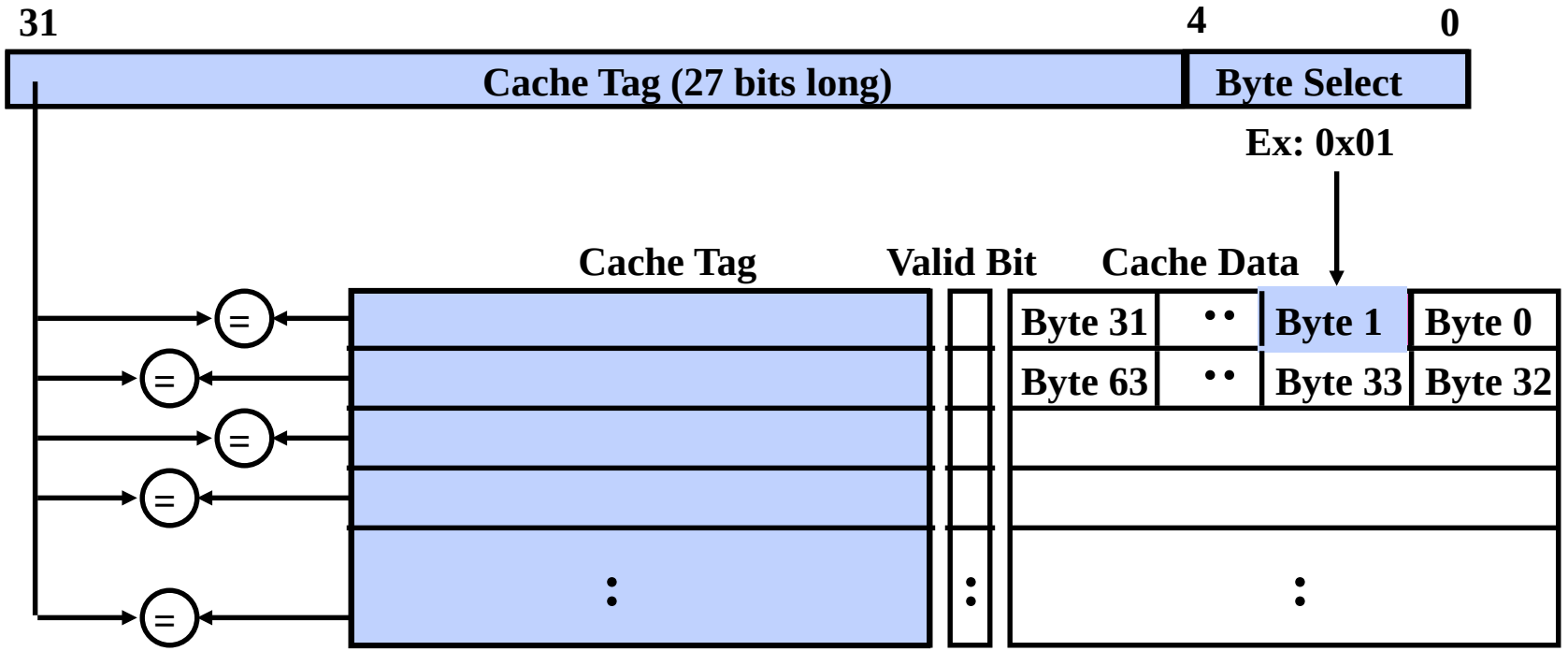
How is a Block found in a Direct Mapped Cache?



- Index Used to Lookup Candidates in Cache
  - Index identifies the set
- 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

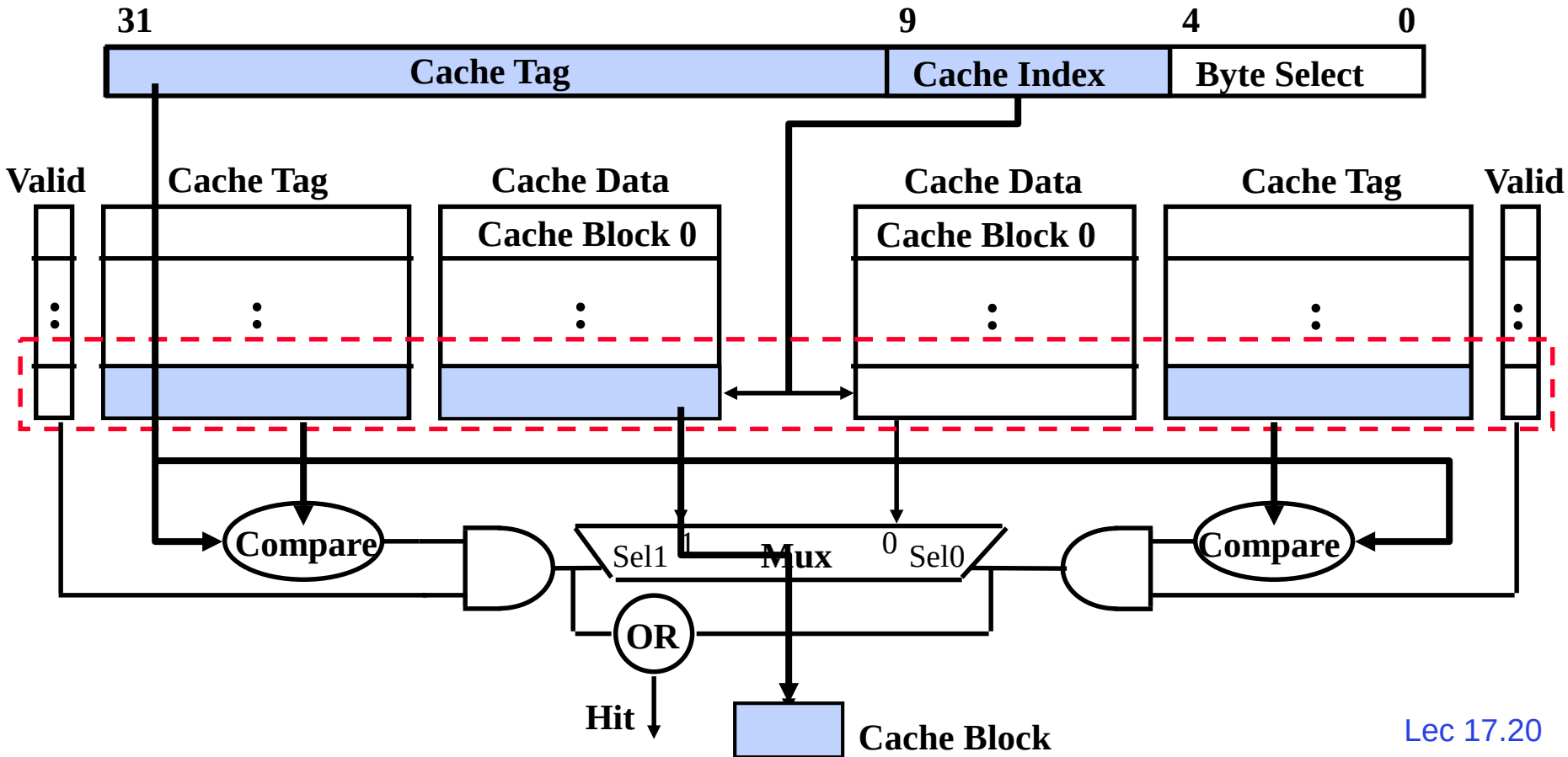
Fully Associative Cache

- 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



Set Associative Cache

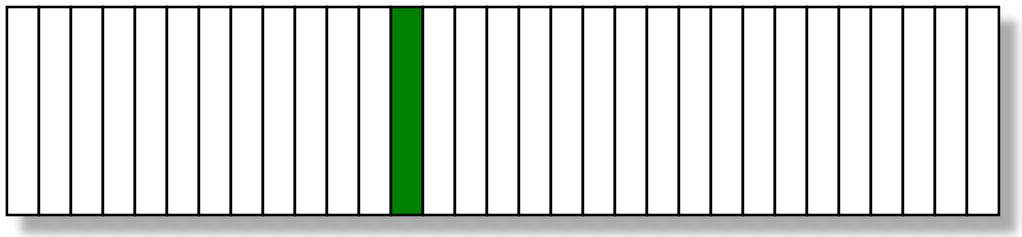
- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



Where does a Block Get Placed in a Cache?

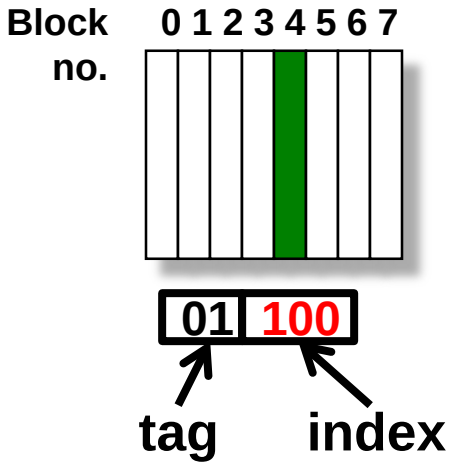
- Example: Block 12 placed in 8 block cache

32-Block Address Space:

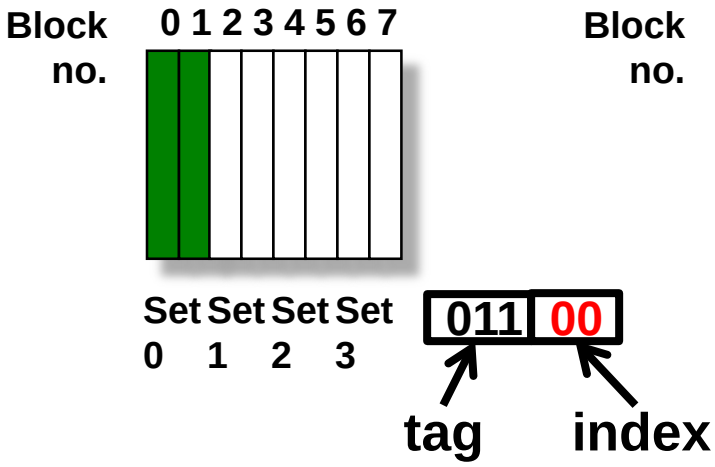


Block no. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

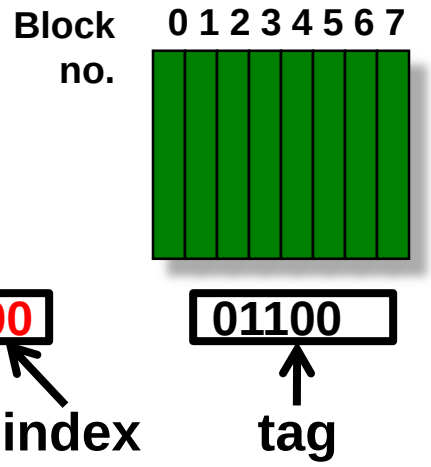
Direct mapped:
block 12 (01100) can go only into block 4 (12 mod 8)



Set associative:
block 12 can go anywhere in set 0



Fully associative:
block 12 can go anywhere



Sources of Cache Misses

- Compulsory (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - Coherence (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

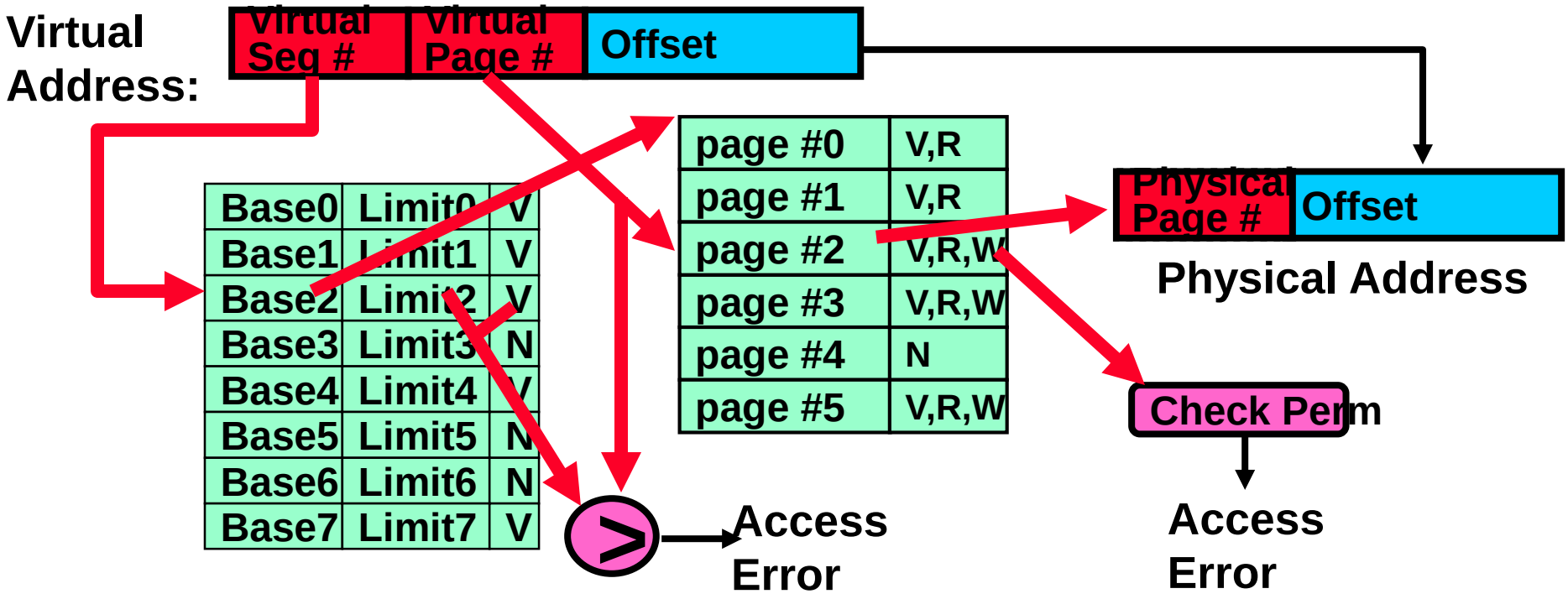
Which Block Should be Replaced on a Miss?

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

What Happens on a Write?

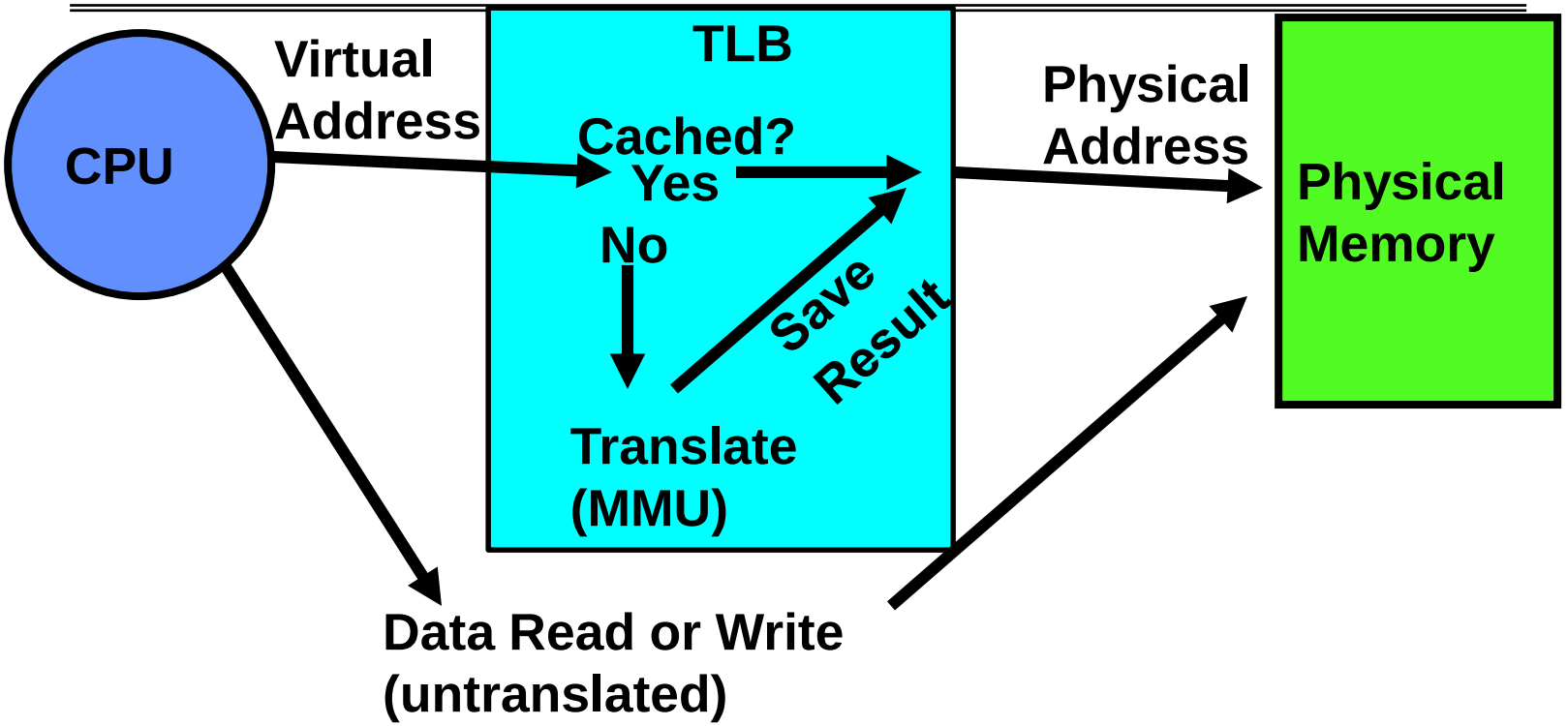
- Write through: The information is written both to the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache.
  - Modified cache block is written to main memory only when it is replaced
- Pros and Cons of each?
  - WT:
    - » PRO: read misses cannot result in writes
    - » CON: processor held up on writes unless writes buffered
  - WB:
    - » PRO: repeated writes not sent to DRAM
processor not held up on writes
    - » CON: More complex

Another Major Reason to Deal with Caching



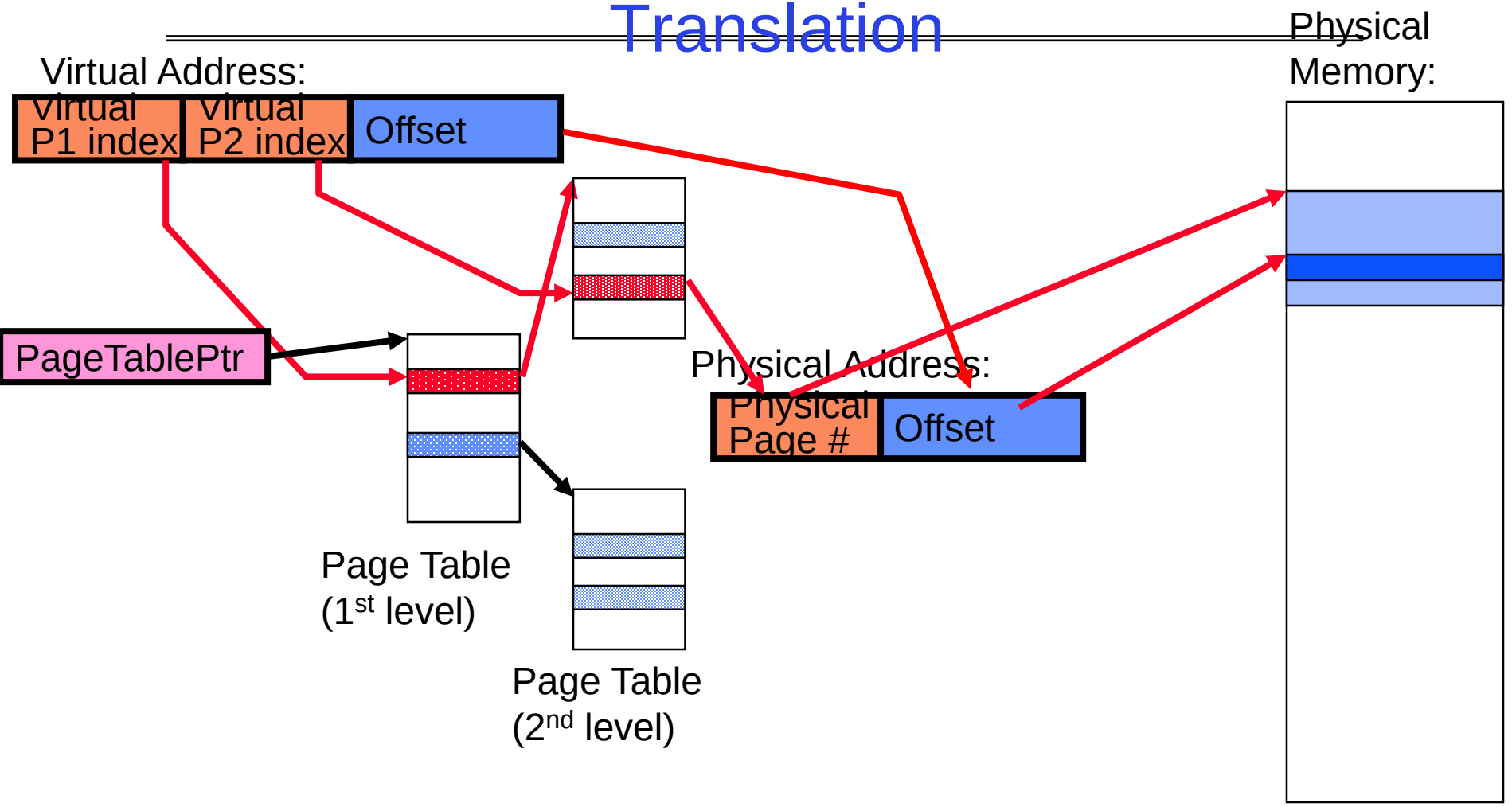
- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
- Solution? Cache translations!
  - Translation Cache: TLB (“Translation Lookaside Buffer”)

Caching Applied to Address Translation

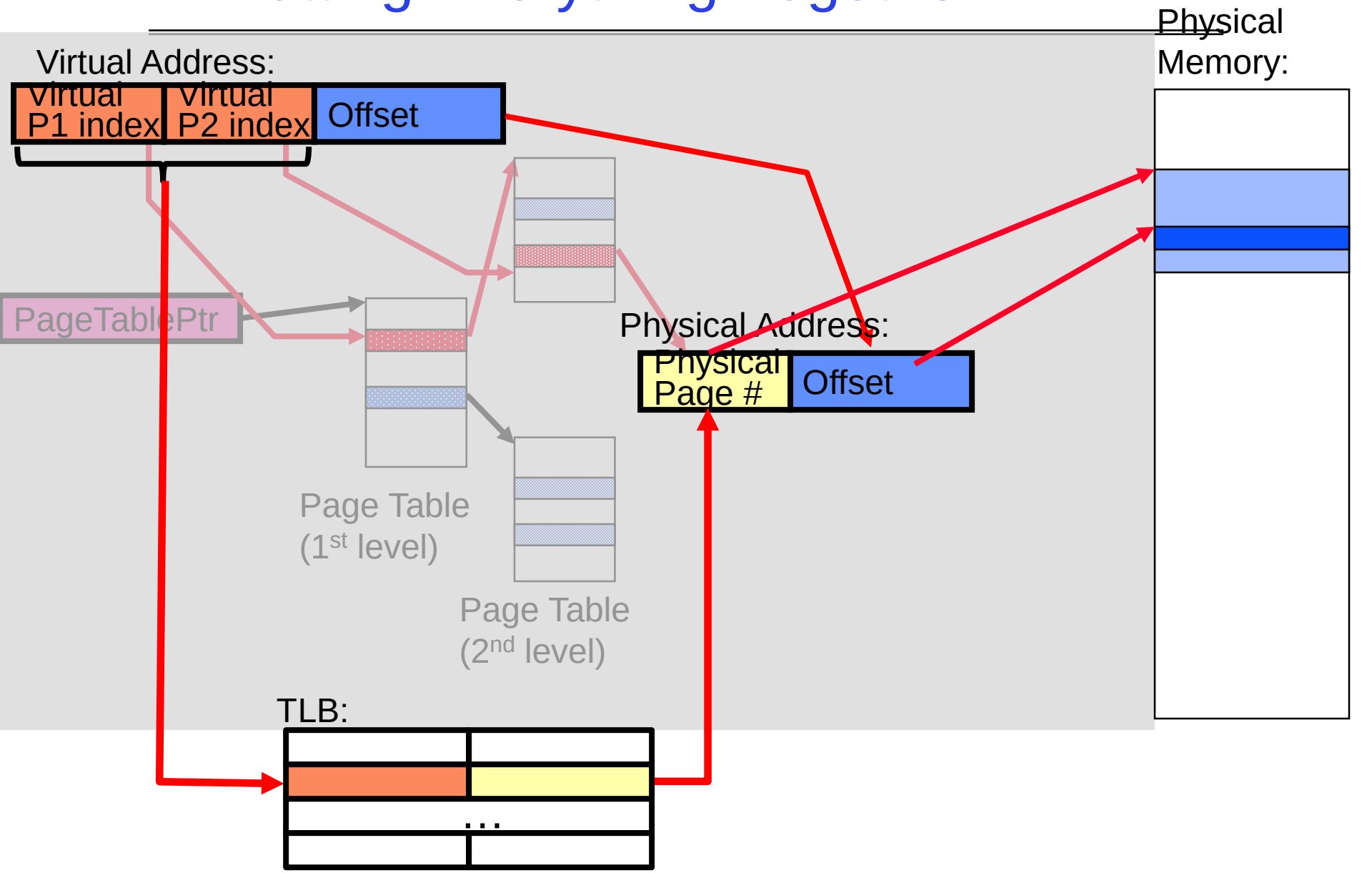


- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…

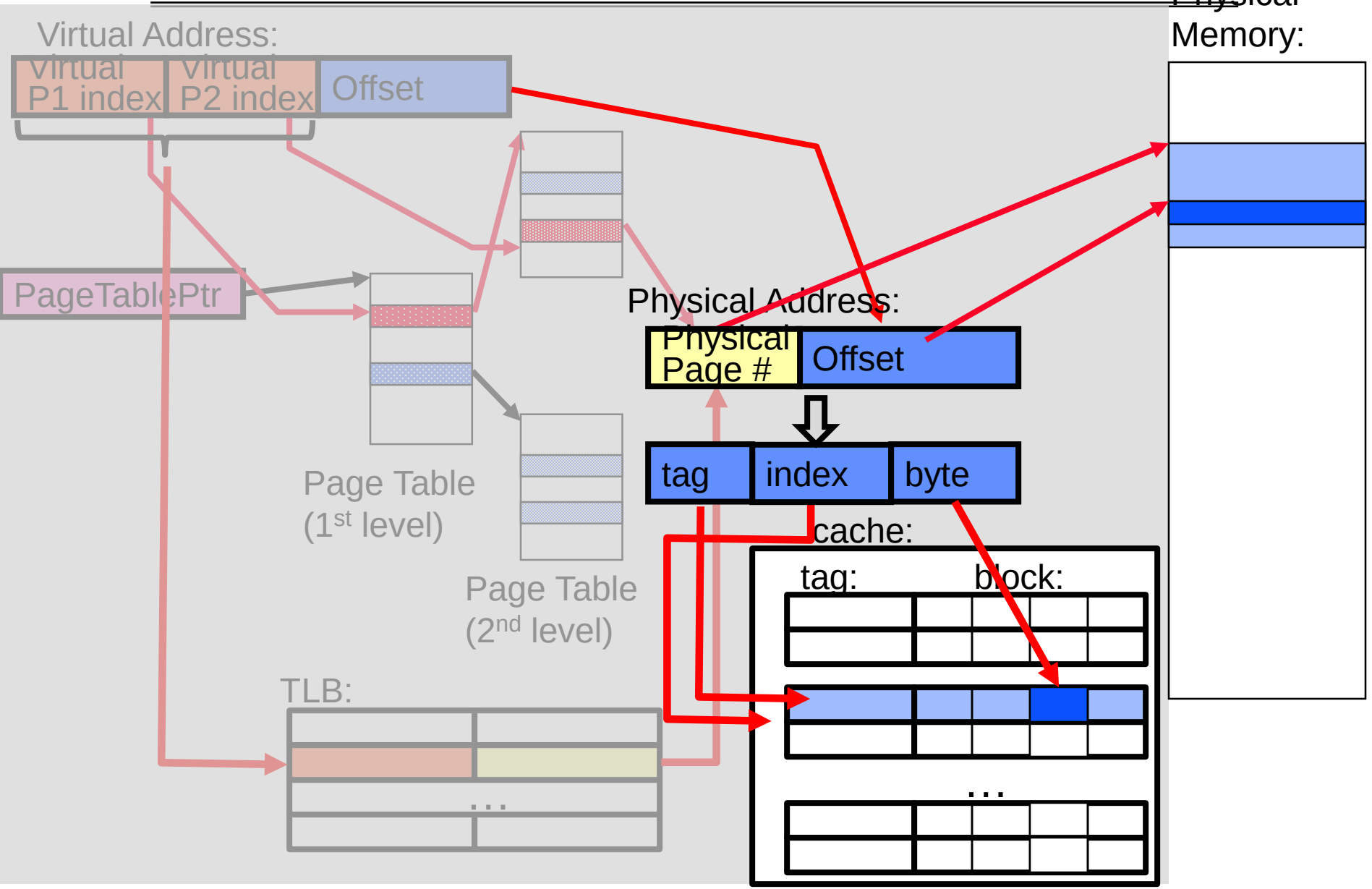
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: 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

Summary (1/2)

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - » Temporal Locality: Locality in Time
    - » Spatial Locality: Locality in Space

- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices

Summary (2/2)

- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent

- TLB is cache on address translations
  - Fully associative to reduce conflicts

Next week

- Chapter 9, including but not limited to the following key words.
  - Page replacement
  - Kernel user