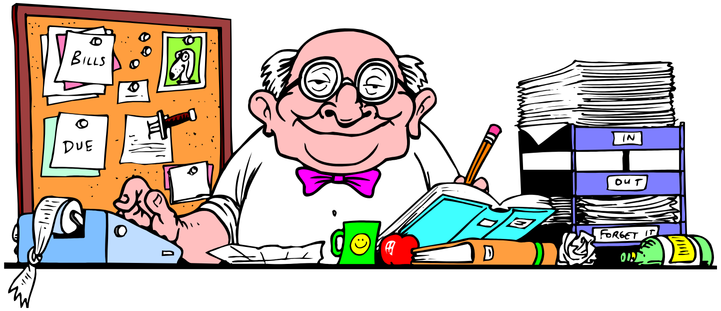
SE350: Operating Systems

Lecture 11: Caching

Outline

- Principle of locality
    - Temporal locality: Locality in time
    - Spatial locality: Locality in space
- Cache organizations
    - Direct mapped, set associative, fully associative
- Major categories of cache misses
    - Compulsory, conflict, capacity, coherence
- Translation Lookaside Buffer (TLB)
    - Cache relatively small number of PTEs
    - On TLB miss, page table is traversed

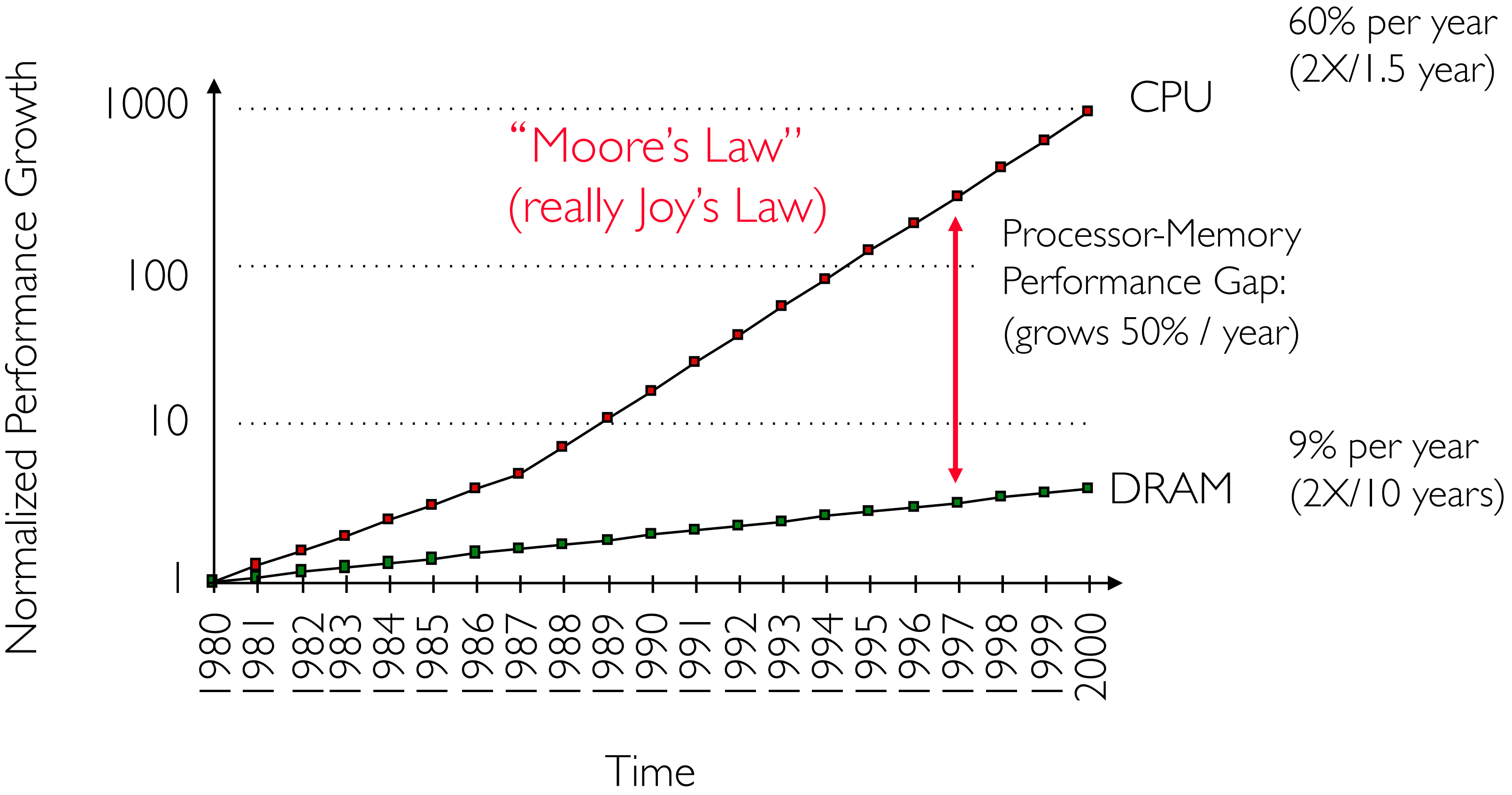
Caching Concept



- Cache is repository for copies that can be accessed more quickly
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - We can cache memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if
  - Frequent case is frequent enough and
  - Infrequent case is not too expensive

Why Bother with Caching?

Processor-DRAM Memory Gap (latency)



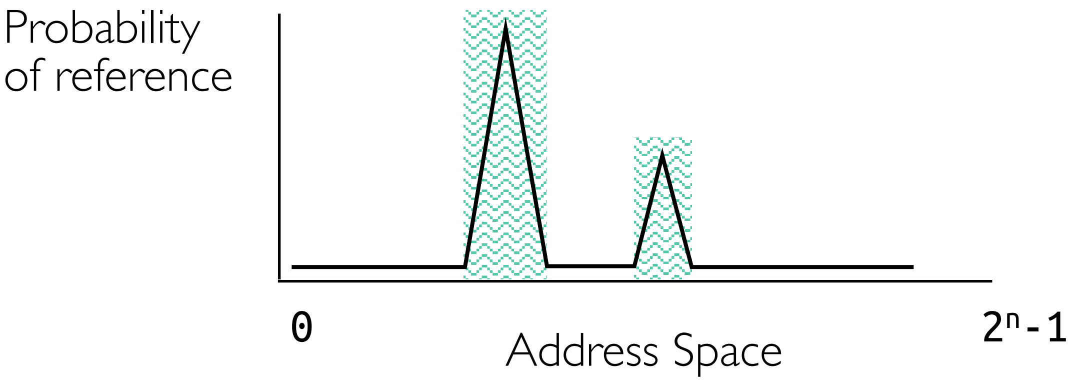
60% per year
(2X/1.5 year)

“Moore’s Law”
(really Joy’s Law)

Processor-Memory
Performance Gap:
(grows 50% / year)

9% per year
(2X/10 years)

Why Does Caching Help? Locality!



- Temporal locality (locality in time):
  - Cache recently accessed data items
- Spatial locality (locality in space):
  - Cache contiguous blocks

Some Terminology

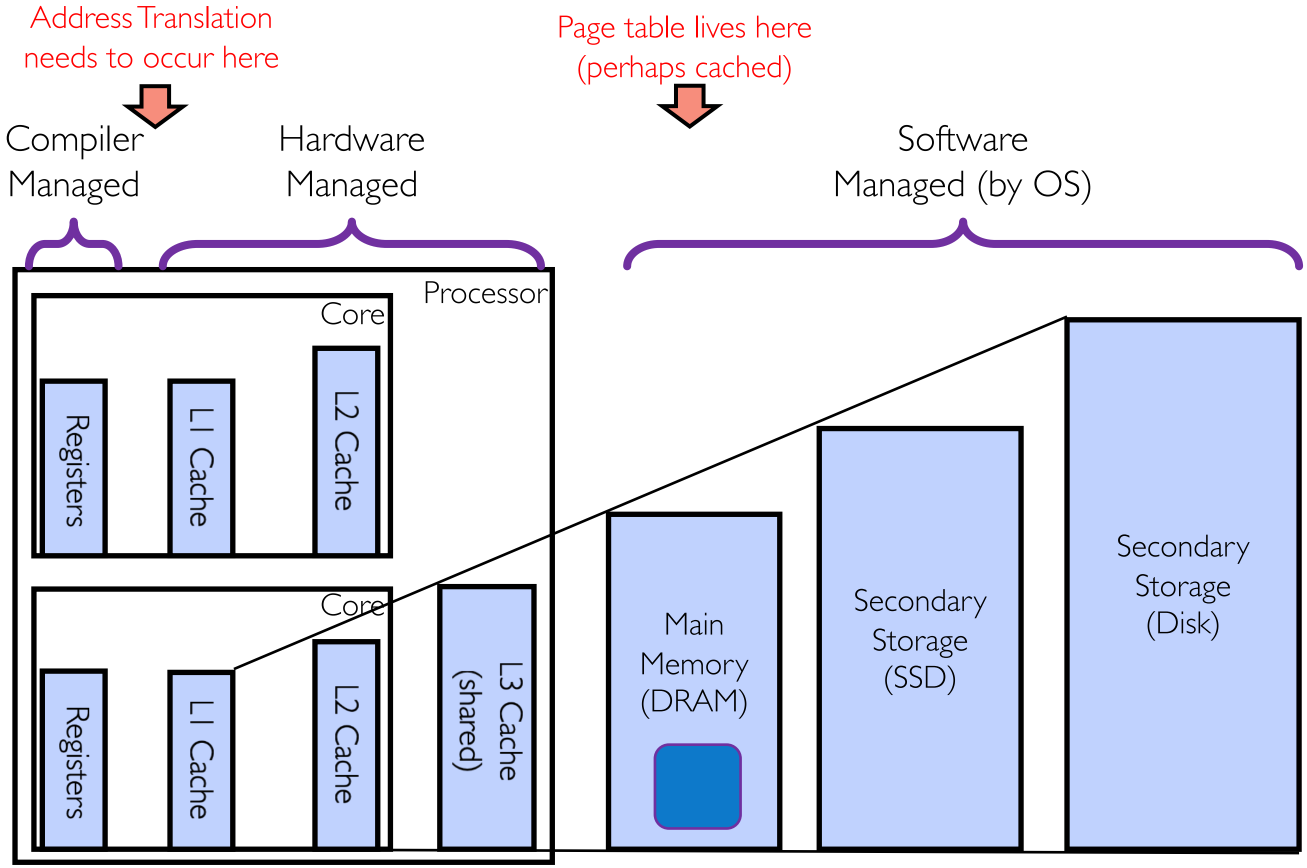
- Block: Group of spatially contiguous and aligned bytes (words)
  - Typical sizes are 32B, 64B, 128B
- Hit: Access level of memory and find what we want
  - Hit time: Time to hit (or discover miss)
- Miss: Access level of memory and do NOT find what we want
  - Miss time: Time to satisfy miss
  - Misses are expensive (take a long time) ⇒ Try to avoid them
  - But, if they happen, amortize their costs ⇒ Bring in more than just specific word you want ⇒ Bring in whole block (multiple words)

Some Terminology (cont.)

- Hit rate: num of hits / (num of hits + num of misses)
  - Miss rate = 1 – hit rate
  - High hit rate means high probability of finding what we want
- Avg. access time: hit rate x hit time + miss rate x (hit time + miss time)
  - Equal to hit time + miss rate x miss time
- Problem: hard to get low hit time and miss rate in one memory structure
  - Large memory structures have low miss rate but high hit time
  - Small memory structures have low hit time but high miss rate
- Solution: use *hierarchy* of memory structures

Memory Hierarchy of Modern Computer Systems

- Goal: Bring average memory access time close to L1's

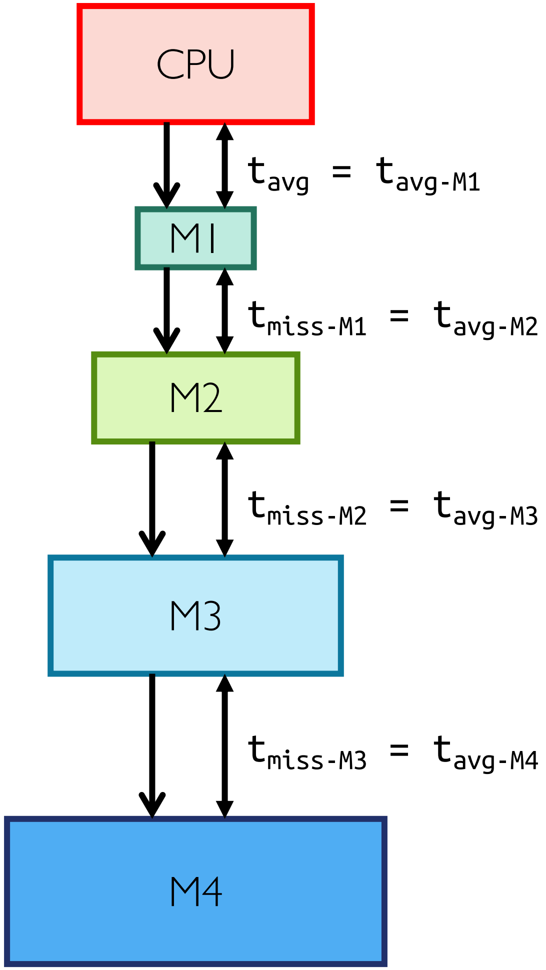


Address Translation needs to occur here

Page table lives here (perhaps cached)

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

Abstract Hierarchy Performance



$t_{avg} = t_{avg\text{-}M1}$

$t_{miss\text{-}M1} = t_{avg\text{-}M2}$

$t_{miss\text{-}M2} = t_{avg\text{-}M3}$

$t_{miss\text{-}M3} = t_{avg\text{-}M4}$

How do we compute t_{avg} ?

$$= t_{avg\text{-}M1}$$
$$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} \times t_{miss\text{-}M1})$$
$$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} \times t_{avg\text{-}M2})$$
$$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} \times (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} \times t_{miss\text{-}M2})))$$
$$= t_{hit\text{-}M1} + (\%_{miss\text{-}M1} \times (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} \times t_{avg\text{-}M3})))$$
$$= \ldots$$

Note: Miss at level X = Access at level X+1

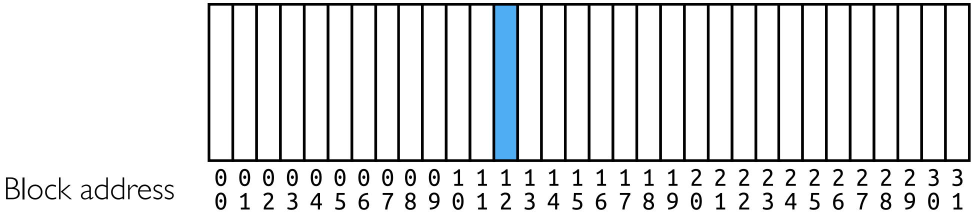
Where to Put Blocks in Cache?

- Divide cache into *sets*
  - Each block can only go in its set ⇒ there is 1-to-1 mapping from block address to set
  - Each set holds some number of blocks ⇒ set associativity
    - E.g., 4 blocks per set ⇒ 4-way set-associative
- At extremes
  - Whole cache has just one set ⇒ fully associative
    - Most flexible (longest access latency)
  - Each set has 1 block ⇒ 1-way set-associative ⇒ direct mapped
    - Least flexible (shortest access latency)

Where to Put Blocks in Cache? (cont.)

- Example: where is block 12 placed in 8-block cache?

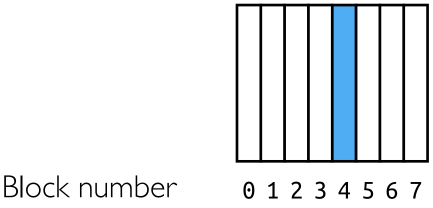
32-Block Address Space

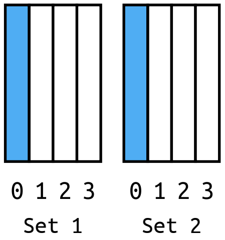


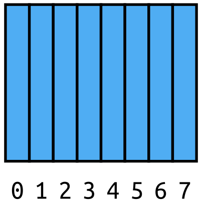
Direct mapped
Block 12 can go
only into block 4
(12 mod 8)

2-way set-associative
Block 12 can go
anywhere in set 0
(12 mod 4)

Fully associative
Block 12 can go
anywhere

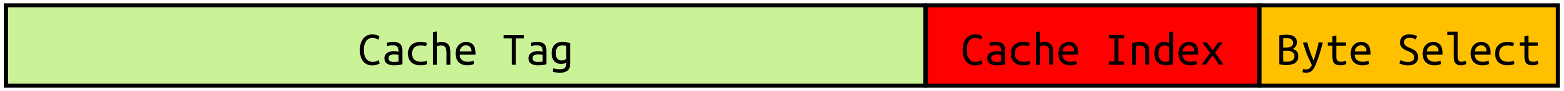






How is Block Found in Cache?

Memory address



Cache Tag

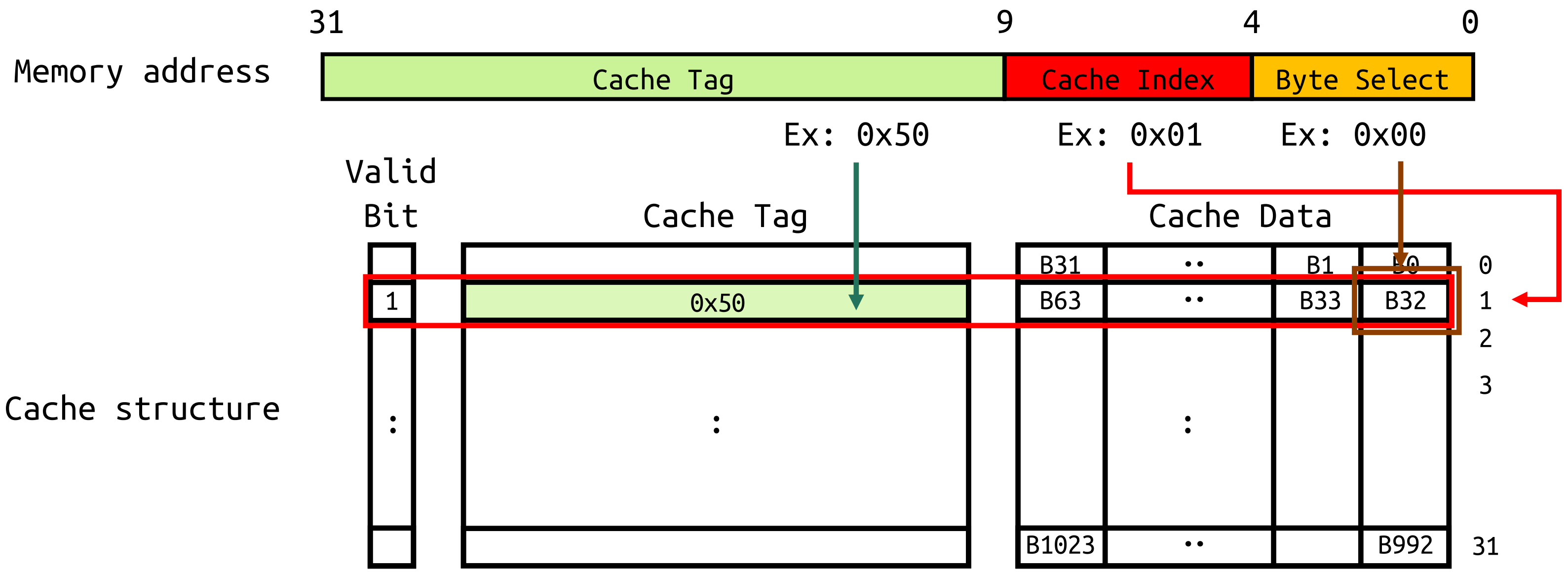
Cache Index

Byte Select

- Byte select field used to select data within block
  - Offset of byte in block
- Cache index used to lookup candidate blocks in cache
  - Index identifies set
- Cache tag used to identify actual copy
  - If no candidate matches, then declare cache miss

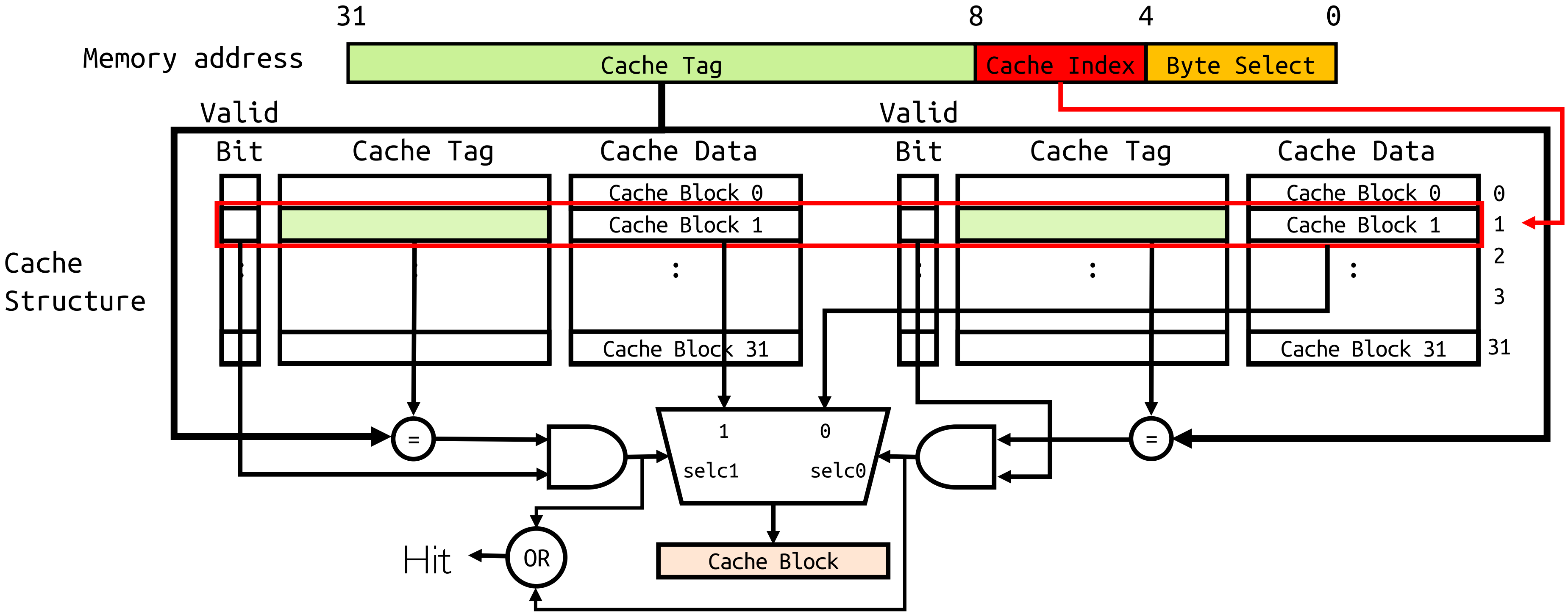
Direct Mapped Cache

- Direct mapped 2^N byte cache with block size of 2^M bytes
  - Uppermost $(32 - N)$ bits of address are cache tag
  - Lowest M bits are byte select, rest are cash index
- Example: 1KB direct mapped cache with 32B blocks
  - $Log_2 32 = 5$ bits for byte select, $32 - Log_2 1024 = 22$ bits for cache tag
  - $32 - 5 - 22 = 5$ bits for cache index



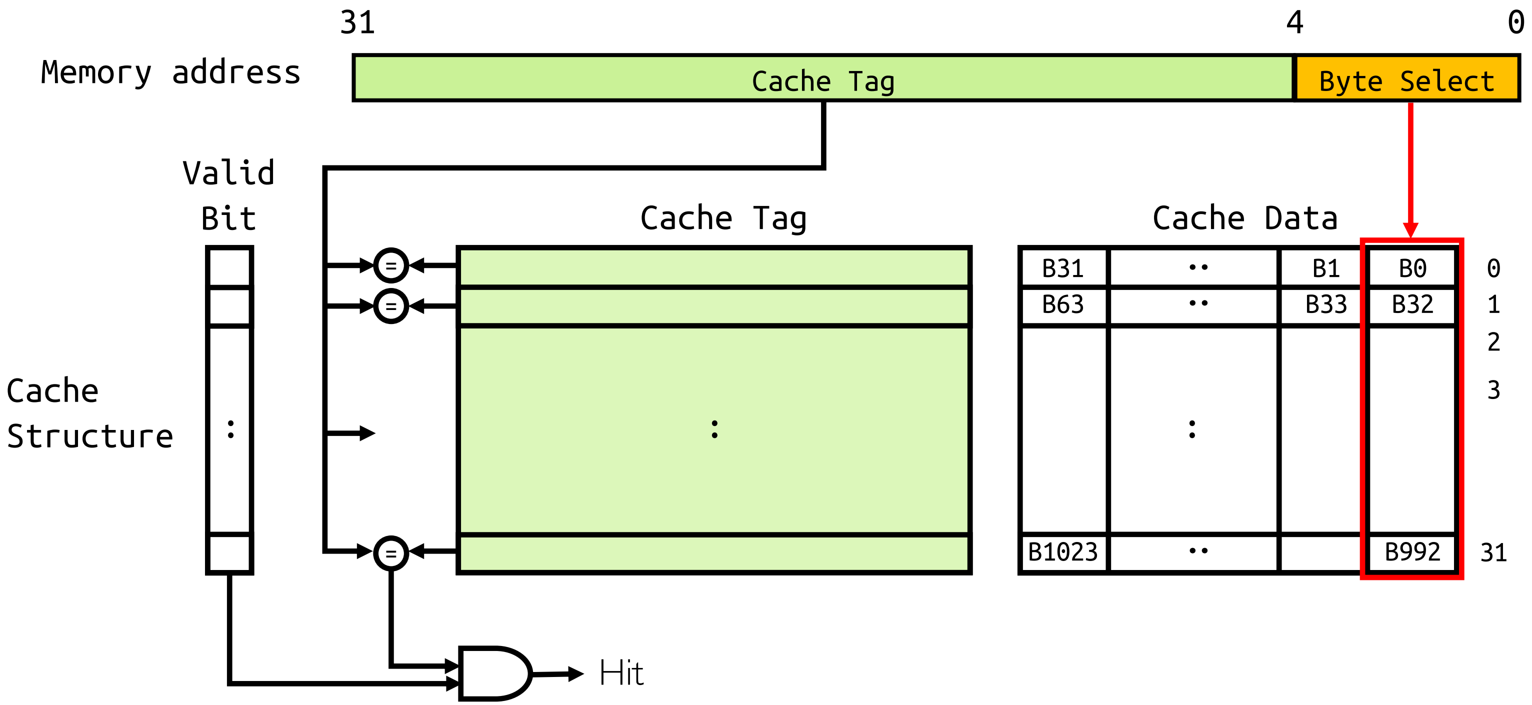
Set-Associative Cache

- 2^K-way set-associative 2^N **byte** cache with block size of 2^M **bytes**
  - Lowest **M bits** for byte select, (32 - N + K) **bits** for cache tag, rest for cache index
  - 2^K direct mapped caches operates in parallel
- Previous example, now with 2-way set-associativity
  - Cache Index selects "set" from cache, there are 16 sets ⇒ **4 bits** for index



Fully Associative Cache

- Every cache block can hold any memory block
  - Address does not include cache index
  - Compare cache tags of all cache blocks in parallel
- Previous example now with fully associative cache



Possible Sources of Cache Misses

- Compulsory (cold)
  - Cache hasn't seen this block before (start or migration of process)
  - "Cold" fact of life: not whole lot you can do about it
  - When running billions of instruction, compulsory misses are insignificant
- Capacity
  - Cache cannot contain all blocks accessed by program
  - Solution: increase cache size
- Conflict (collision)
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (no conflict misses in fully associative cache)
- Coherence (invalidation)
  - Other process (e.g., I/O) updates memory

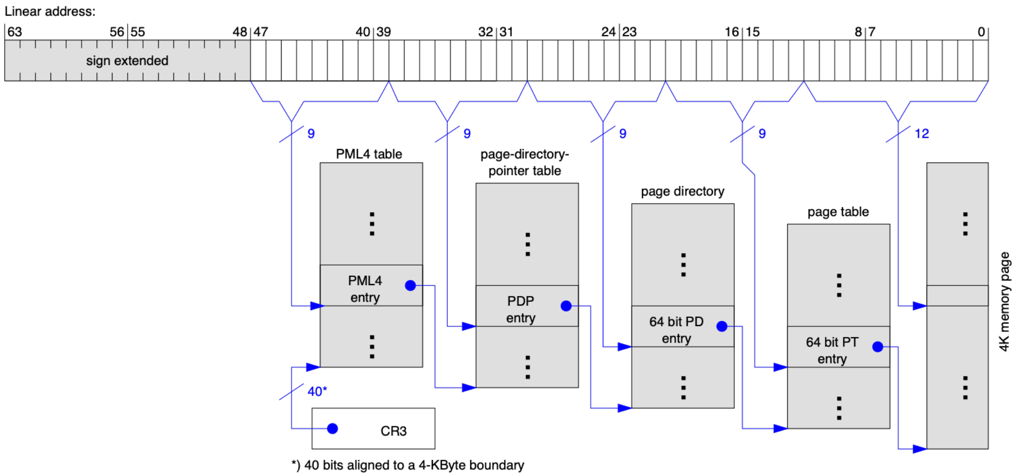
Which Block Should be Replaced on Cache Miss?

- Easy for direct mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - Least Recently Used (LRU, more on this later)

What Happens on Write?

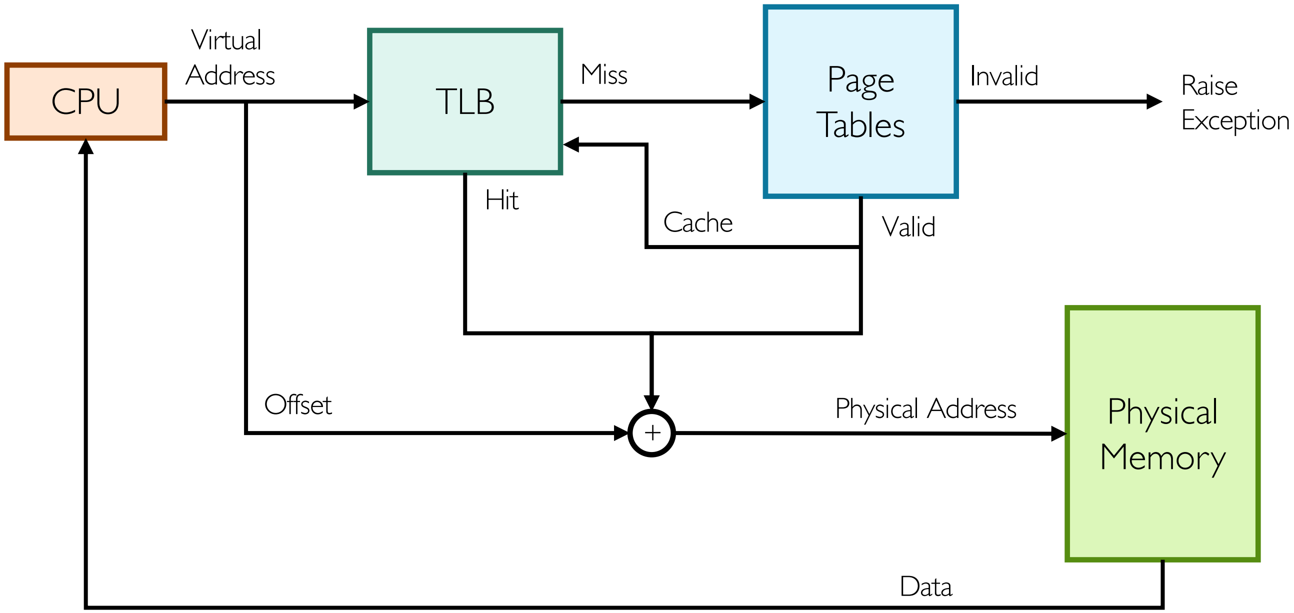
- Write through: Write to both cache and lower-level memory
- Write back: Write only to cache
  - Modified cache block is marked dirty
  - On replacement, dirty block is written to lower-level memory
- Pros and Cons of each?
  - WT
    - PRO: Read misses cannot result in writes
    - CON: Processor held up on writes unless writes are buffered
  - WB
    - PRO: Repeated writes are not sent to DRAM
      Processor not held up on writes
    - CON: More complex
      Read miss may require writeback of dirty data

Address Translation: Major Application of Caching



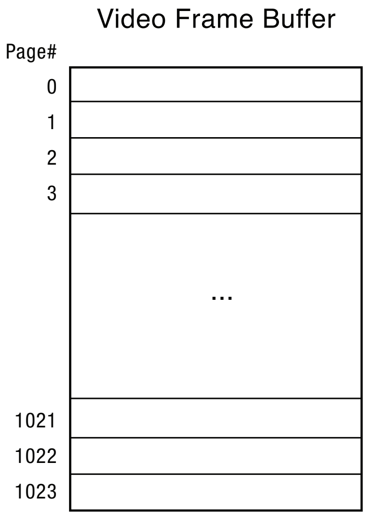
- Cannot afford to translate on every access
  - At least five DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially resides on disk!
  - Even worse, what if we use caches to make memory access faster than DRAM access?
- Solution?
  - Cache translations in Translation Lookaside Buffer (TLB)
    - Fully associative (since conflict misses are expensive)

Caching Applied to Address Translation



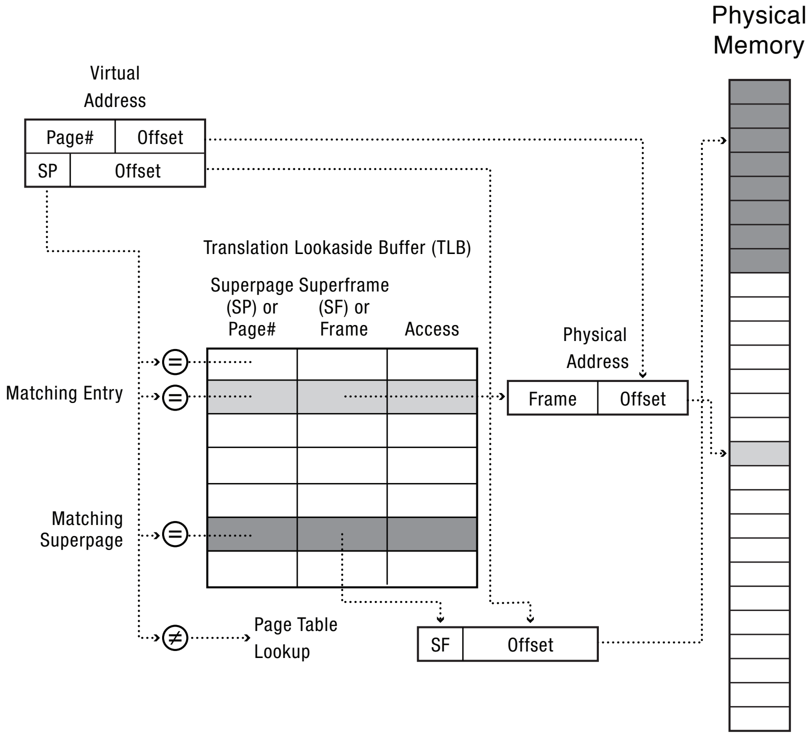
- Does page locality exist?
  - Instruction accesses: Sequential accesses ⇒ Frequent accesses to the same page ⇒ Yes!
  - Stack accesses: Definite locality of reference ⇒ Yes!
  - Data accesses: Less page locality, but still some ⇒ Yes, so so!
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds

Do TLBs Always Work for Sequential Accesses?



- Example: For HD displays, video frame buffer could be large
  - E.g., 4k display: 32 bits x 4K x 3K = 48MB (spans12K of 4KB pages)
- Even large on-chip TLB with 256 entries cannot cover entire display
- Each horizontal line of pixels could be on a page
- Drawing a vertical line could require loading a new TLB entry

Superpages: Improving TLB Hit Rate



- Reduce number of TLB entries for large, contiguous regions of memory
  - Represent 2 adjacent 4KB pages by single 8KB superpage
- By setting a flag, TLB entry can be a page or a supperpage
  - E.g., in x86: 4KB (12 bits offset), 2MB (21 bits offset), or 1GB (30 bits offset)

What Happens on TLB Miss?

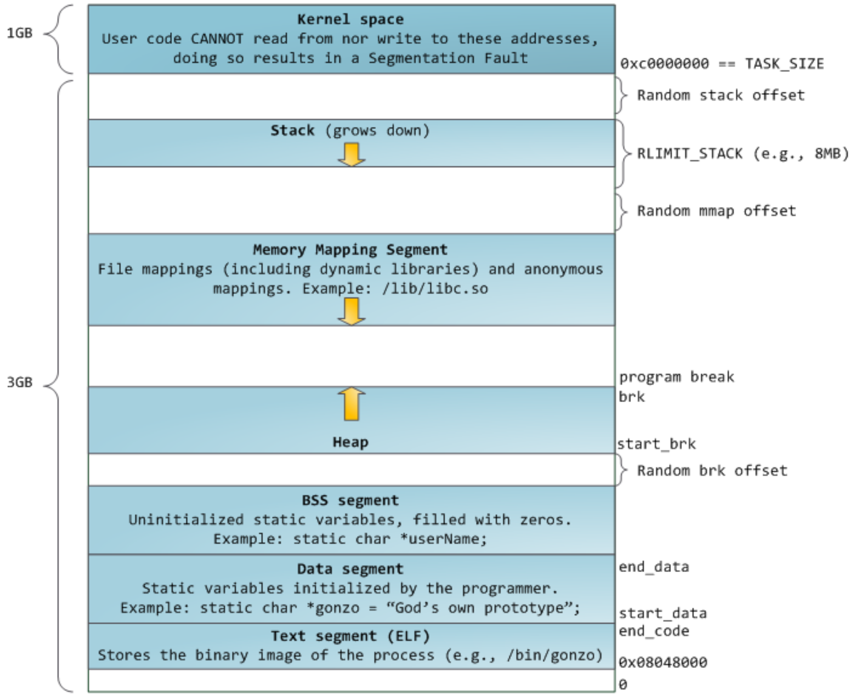
- Hardware-traversed page tables
  - On TLB miss, hardware walks through current page tables to fill TLB (could be multiple levels)
    - Valid PTE: Hardware fills TLB and processor never notices
    - Invalid PTE: CPU raises page fault ⇒ Kernel decides what to do next

- Software-traversed page tables
  - On TLB miss, CPU raises TLB fault
  - Kernel walks through page table(s) to find PTE
    - Valid PTE: Fills TLB and returns from fault
    - Invalid, internally calls page fault handler

What Happens on Context Switch?

- TLBs map virtual addresses to physical addresses
  - Address space just changed, so TLB entries are no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - What if switching frequently between processes?
  - Include Process-ID in TLB
    - This is microarchitectural solution ⇒ needs extra 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!

Recall: 32-bit Linux Memory Layout (Pre-Meltdown patch!)



1GB

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE

Random stack offset

Stack (grows down)

RLIMIT_STACK (e.g., 8MB)

Random mmap offset

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

3GB

program break
brk

Heap

start_brk

Random brk offset

BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;

end_data

Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";

start_data
end_code

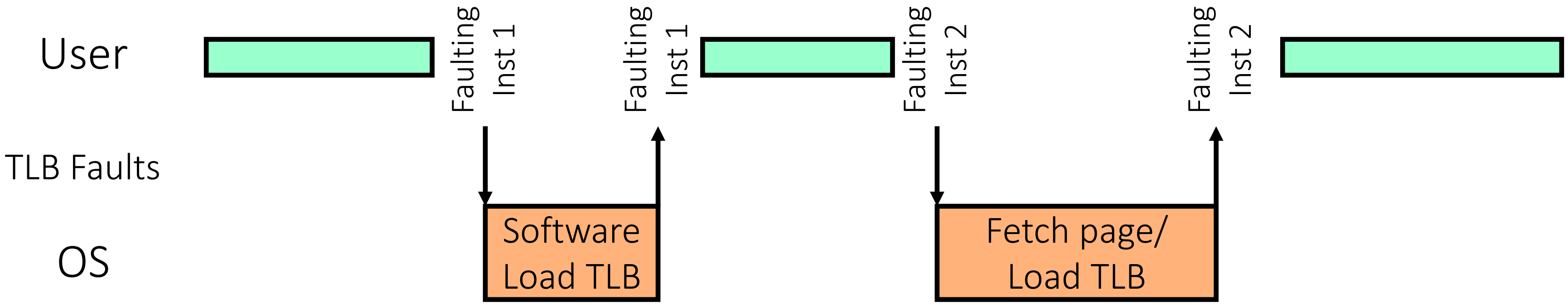
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

0x08048000
0

- Interrupts are frequent; on each interrupts, kernel could access interrupted process' memory very fast (using same page table and TLB entries)
- Translated kernel space addresses can stay in TLB after each context switch
- …

http://static.duartes.org

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 save Faulting instruction and partial state
    - Need to know which instruction caused fault
    - Processor state is needed to restart user thread
      - Save/restore registers, stack, etc.

Permission Reduction

- Keeping TLB consistent with page table is OS's responsibility
- Nothing needs to be done when permission is added
  - E.g., changing invalid to read-only
  - Any reference would cause exception, OS re-loads TLB
- If permission is reduced, TLB should be updated
  - Early computers discarded the entire content of TLB
  - Modern architectures (e.g., x86 and ARM) support removal of individual entries

TLB Shootdown in Multiprocessors

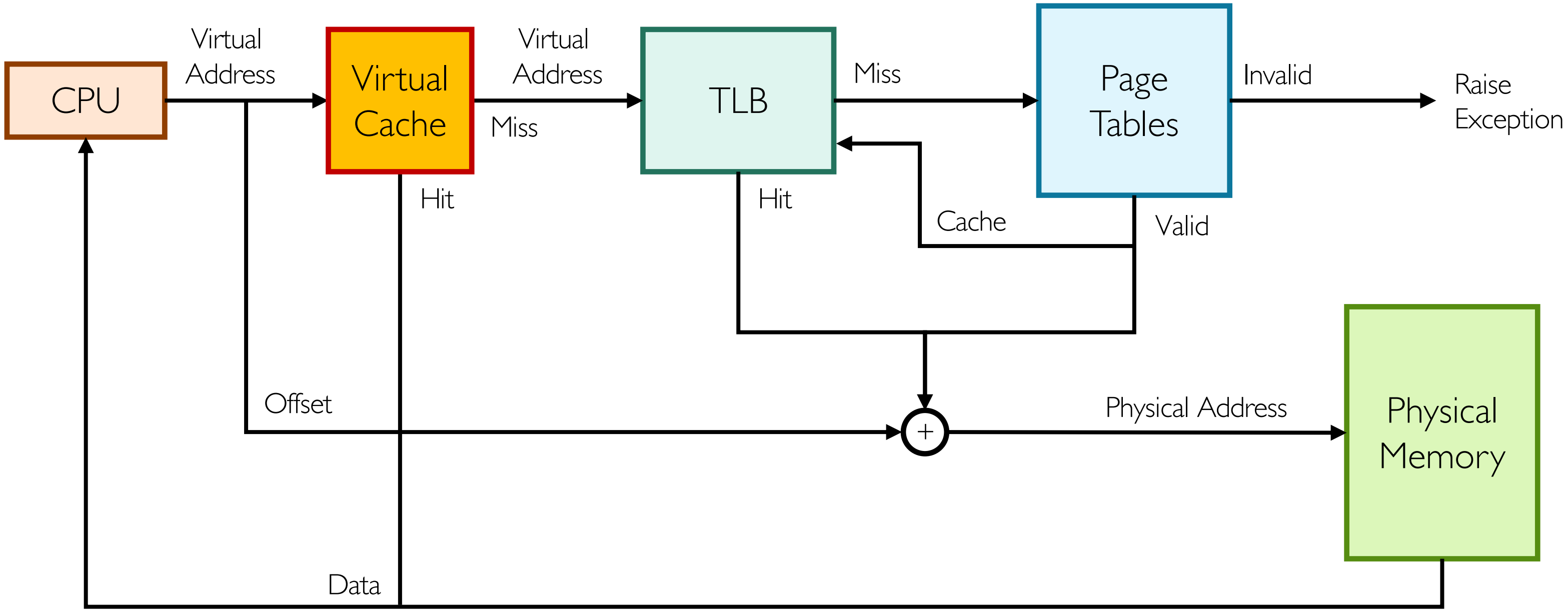
		Process ID	VirtualPage	PageFrame	Access
Processor 1 TLB	=	0	0x0053	0x0003	R/W
	=	1	0x40FF	0x0012	R/W
Processor 2 TLB	=	0	0x0053	0x0003	R/W
	=	0	0x0001	0x0005	Read
Processor 3 TLB	=	1	0x40FF	0x0012	R/W
	=	0	0x0001	0x0005	Read

- Suppose processor 1 wants to update entry for page 0x53 in process 0
  - First, it must remove the entry from its TLB
  - Then, it must send an interprocessor interrupt to each processor requesting it to remove the old translation
- Shootdown is complete only when all processors verify that the old translation has been removed
- TLB shootdown overhead increases linearly with the # of processors

Improve Efficiency Even More!

- TLB improves performance by caching recent translations
- How to improve performance even more?
  - Add another layer of cache!
- What is the cost of first-level TLB miss?
  - Second-level TLB lookup
- What is the cost of a second level TLB miss?
  - x86: 2-4 level page table walk

Improve Efficiency Even More: Virtually Addressed Cache

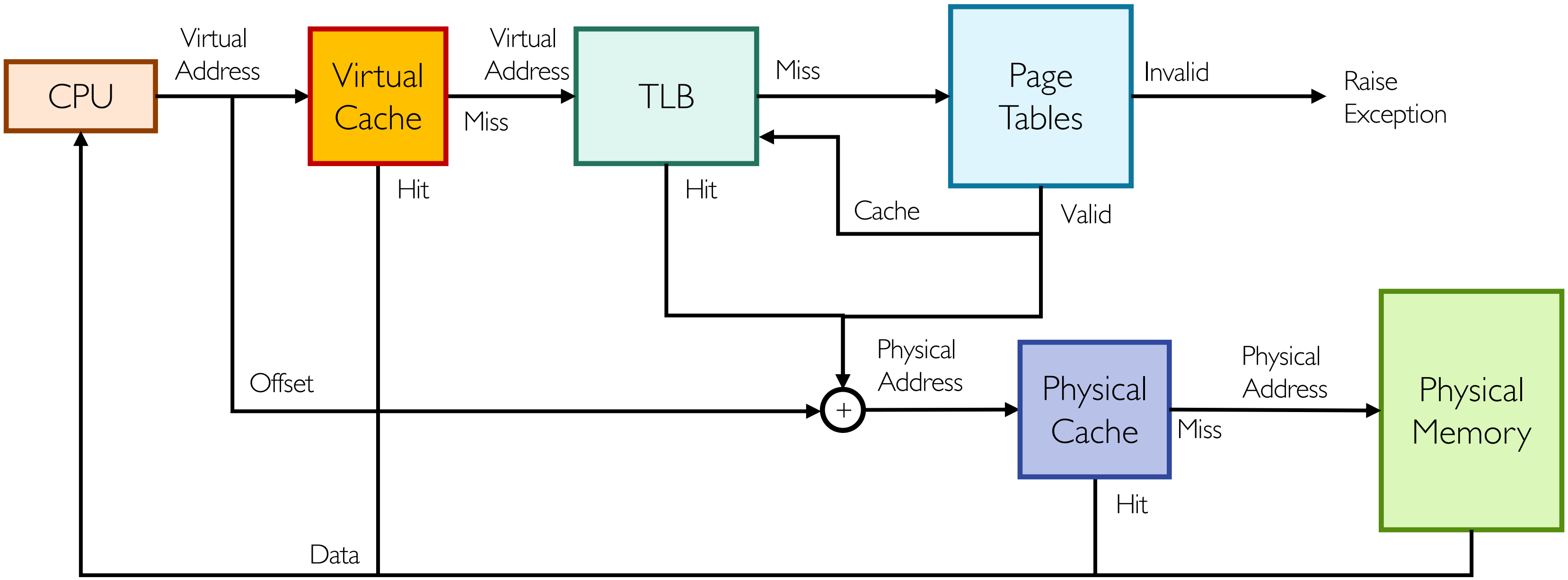


- Too slow to access TLB before looking up address in memory
- Instead, add virtually addressed cached
- In parallel, access TLB to generate physical address in case of cache miss

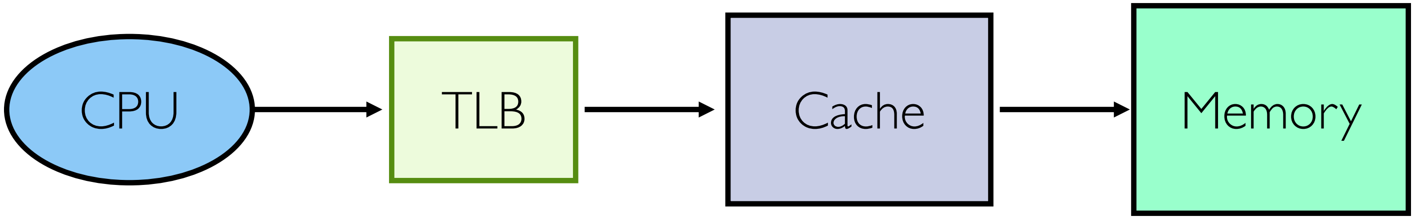
Aliasing

- Multiple virtual addresses could refer to same physical address
- When one process modifies its copy, how does system know to update other processes' copy?
- Typical solution
  - Keep both virtual and physical address for each entry in virtually addressed cache
  - Lookup virtually addressed cache and TLB in parallel
  - Check if physical address from TLB matches multiple entries, and update/invalidate other copies

Putting it All Together: TLB, Virtual, and Physical Caches



TLB Set Associativity



- TLB is on critical path of memory access
  - $t_{avg\text{-}mem\text{-}acc} = t_{hit\text{-}TLB} + (\%_{miss\text{-}TLB} \times t_{miss\text{-}TLB})$
  - TLB access time is added to all memory accesses
  - This seems to argue for direct mapped or low associativity!
  - However, TLB needs to have very few conflicts!
    - Miss time is extremely high!
  - This argues that cost of conflict (miss time) is much higher than slightly increased cost of access (hit time)

Where Are Some of Places That Caching Arises?

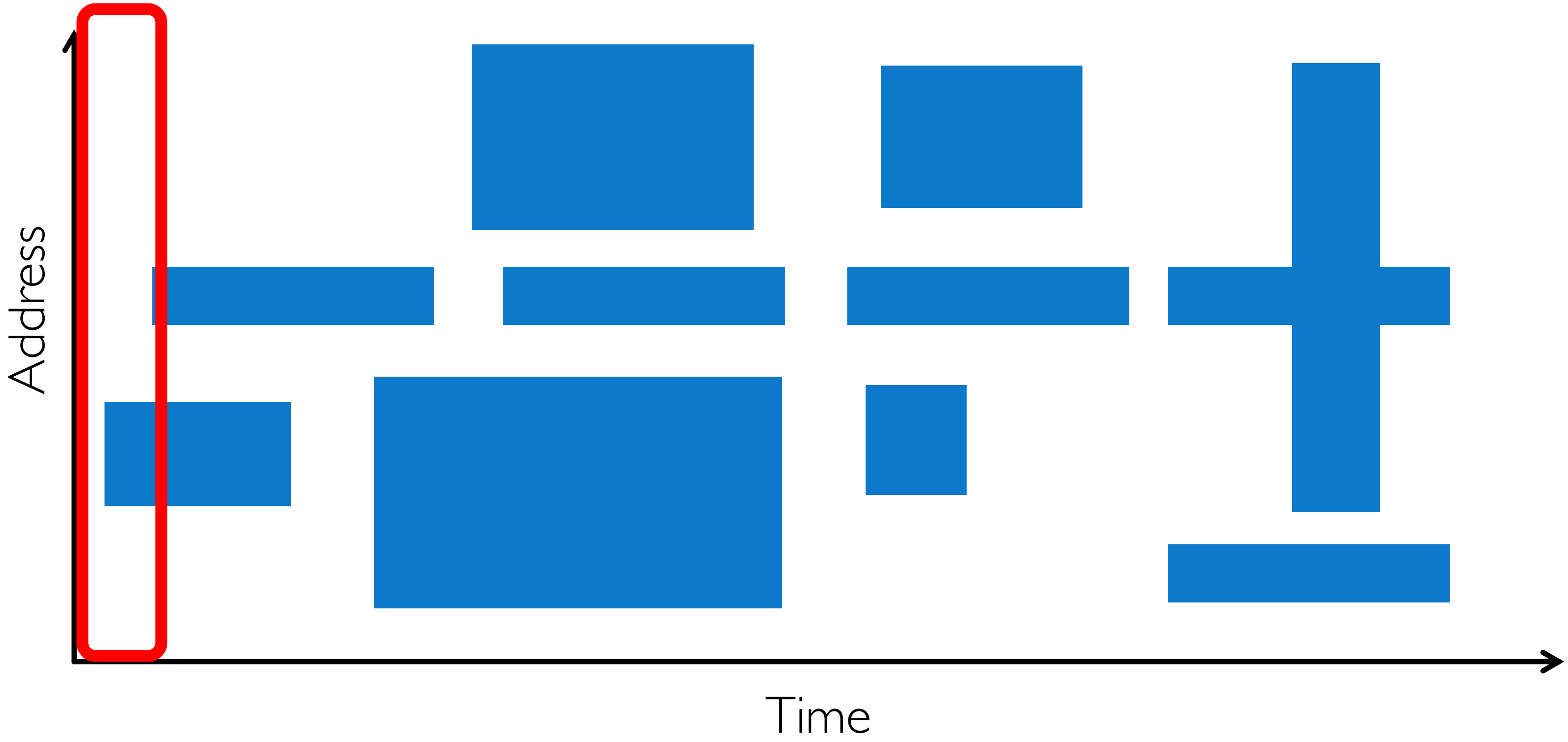
- Direct use of caching techniques
  - TLB (cache of PTEs)
  - Paged virtual memory (memory as cache for disk)
  - File systems (cache disk blocks in memory)
  - DNS (cache hostname to IP address translations)
  - Web proxies (cache recently accessed pages)

Caches and Operating Systems

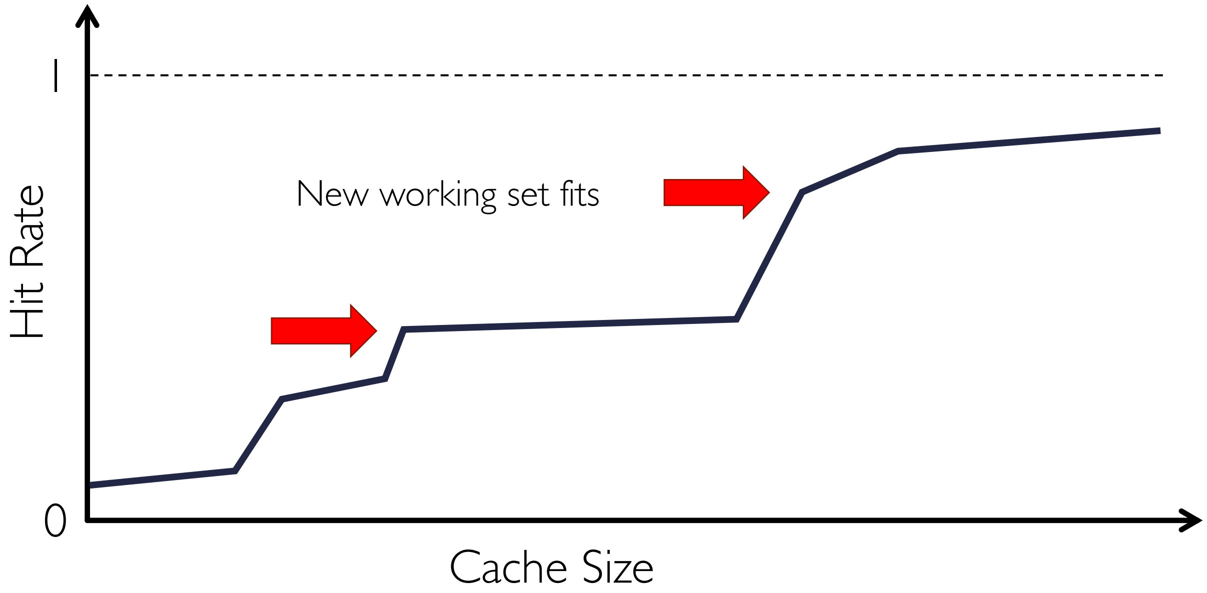
- Indirect - dealing with cache effects (e.g., sync state across levels)
    - Maintaining correctness of various caches
    - E.g., TLB consistency:
        - With PT across context switches?
        - Across updates to PT?
- Process scheduling
    - Which and how many processes are active? Priorities?
    - Large memory footprints versus small ones?
    - Shared pages mapped into VAS of multiple processes?
- Impact of thread scheduling on cache performance
    - Rapid interleavings (small quantum) may degrade cache performance
- Designing operating system data structures for cache performance

Working Set Model

- As program executes, it transitions through sequence of Working Sets (WS) consisting of varying sized subsets of address space

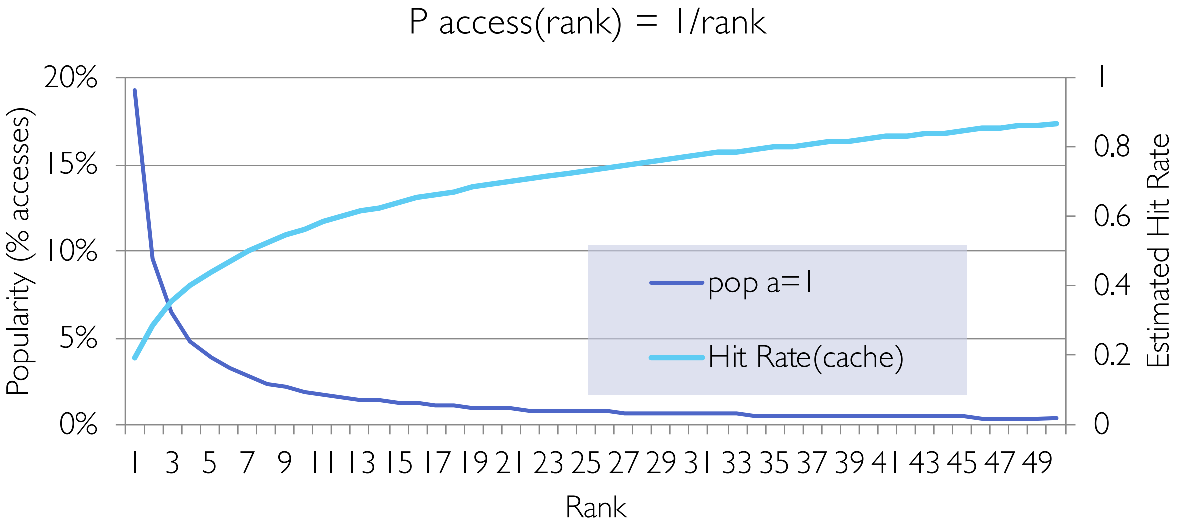


Cache Behavior Under WS Model



- Amortized by fraction of time working set is active
- Transitions from one WS to the next
- Applicable to memory caches, pages, …

Another Model of Locality: Zipf



- Likelihood of accessing item of rank r is ∝ $1/r^a$, for a ∈ [1,2]
- Popularity of webpages, population of cities, distribution of salaries, size of friend lists in social networks, and distribution of references in scientific papers
- Although rare to access items below top few, there are so many of them that it yields heavy tailed distribution
- Substantial value from even a tiny cache
- Substantial misses from even a very large cache

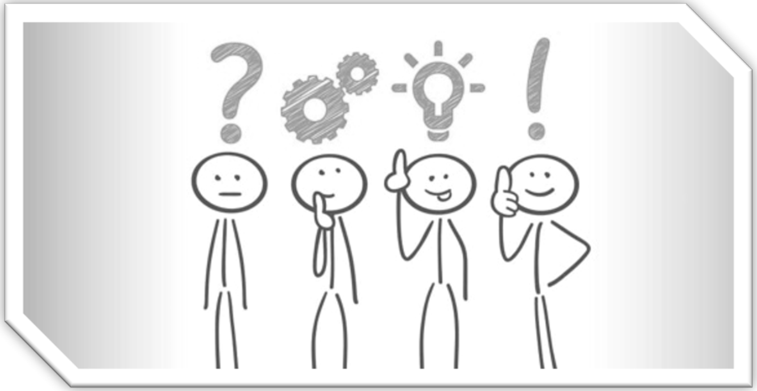
Summary (1/2)

- Principle of locality
  - Program likely to access relatively small portion of address space at any instant of time
    - Temporal locality: Locality in time
    - Spatial locality: Locality in space
- Cache organizations
  - Direct mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent
- Three (+1) major categories of cache misses
  - Compulsory: Sad facts of life
  - Conflict: Increase cache size and/or associativity
  - Capacity: Increase cache size
  - Coherence: Caused by external processors or I/O devices

Summary (2/2)

- Cache of translations called "Translation Lookaside Buffer" (TLB)
  - Relatively small number of PTEs and optional process IDs (< 512)
  - Fully associative (since conflict misses expensive)
  - On TLB miss, page table is traversed and if PTE is invalid, cause page fault
  - On change in page table, TLB entries must be invalidated

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Sorin, Culler, Stoica, Silberschatz, Joseph, and Canny