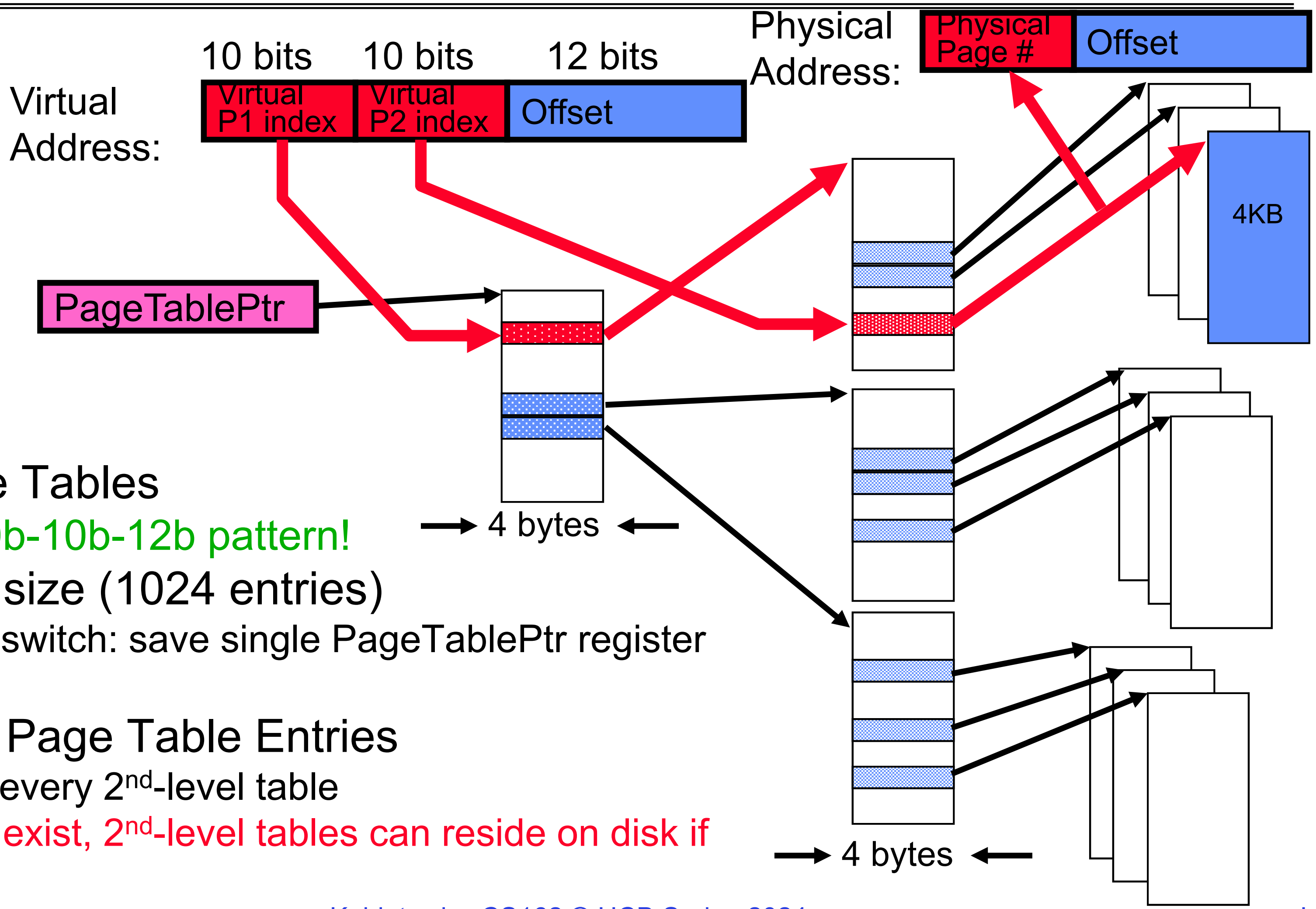
CS162
Operating Systems and Systems Programming
Lecture 16

Memory 3: Caching and TLBs (Con't), Demand Paging

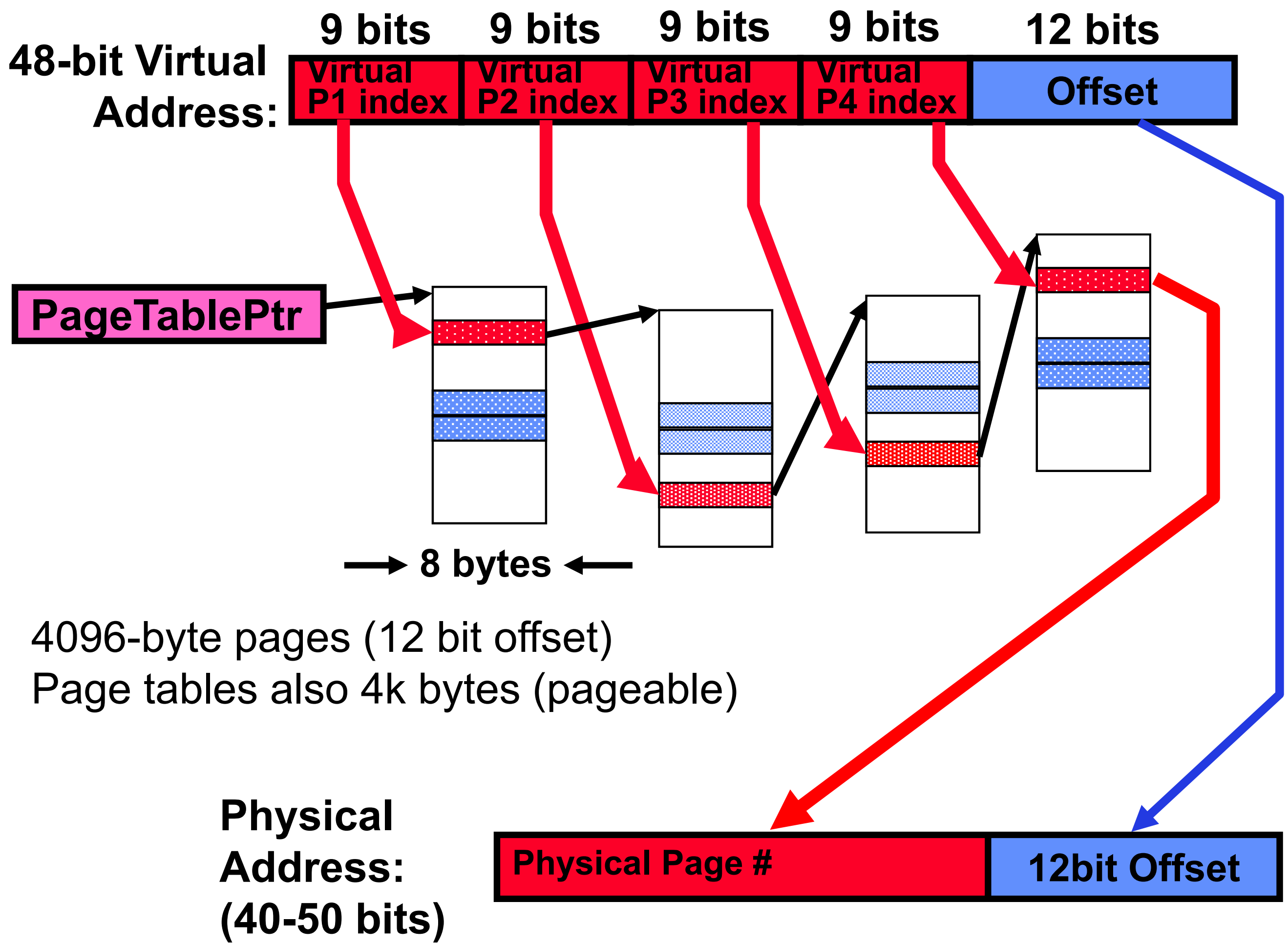
March 12th, 2024
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: The two-level page table



- Tree of Page Tables
  - "Magic" 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e. CR3)
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Recall: X86_64: Four-level page table!



4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

From x86_64 architecture specification

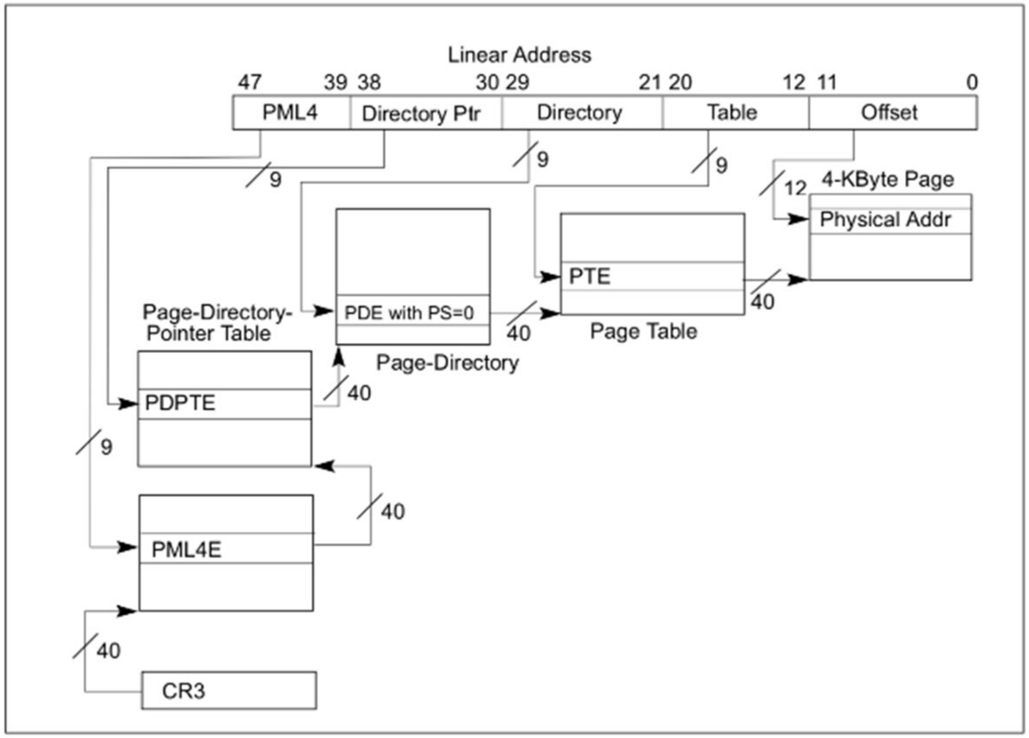


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

- All current x86 processor support a 64 bit operation
- 64-bit words (so ints are 8 bytes) but 48-bit addresses

Larger page sizes supported as well

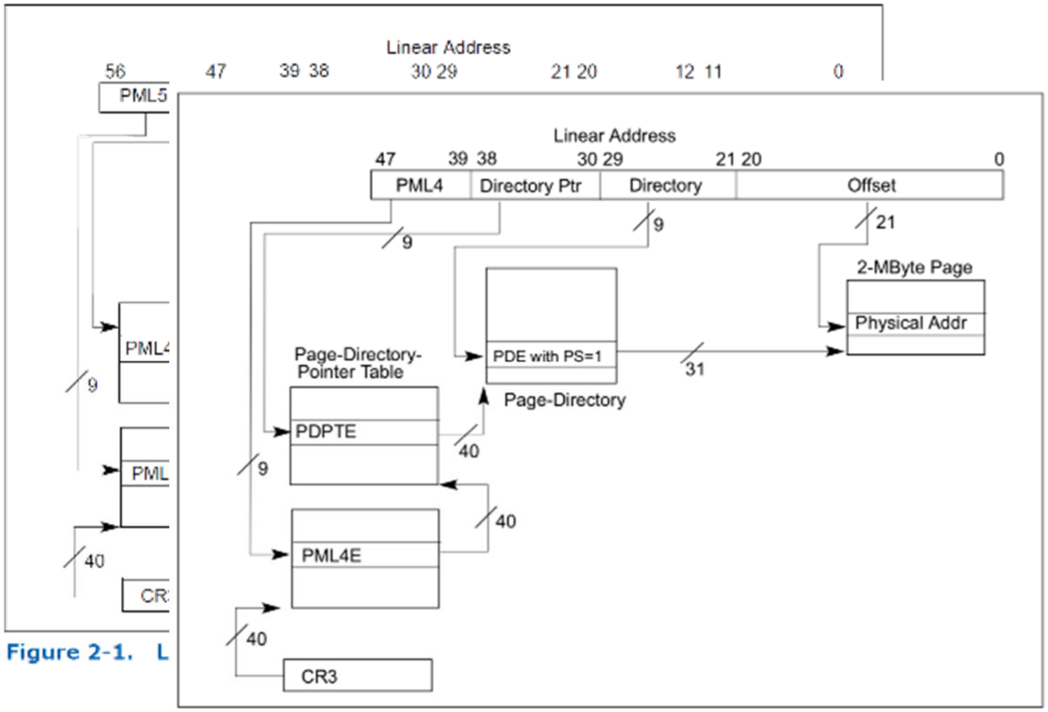


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

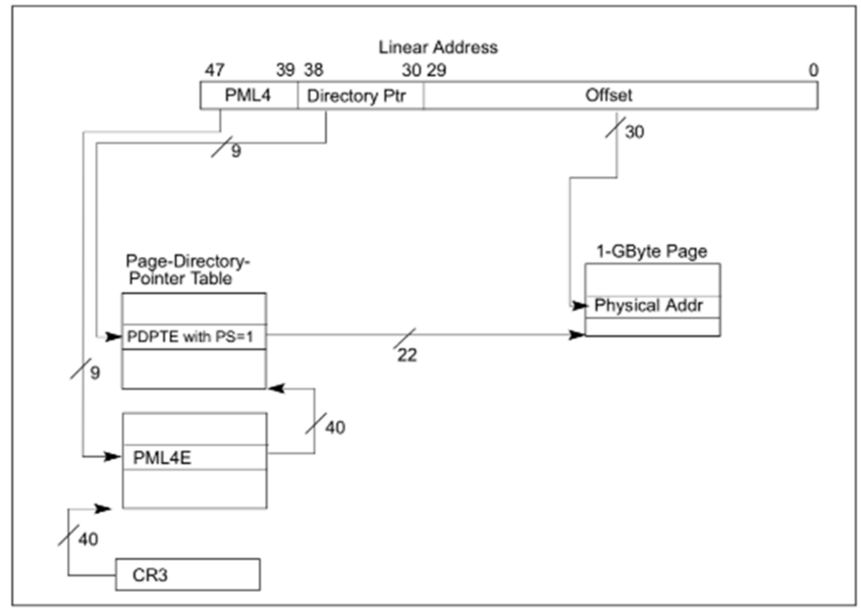
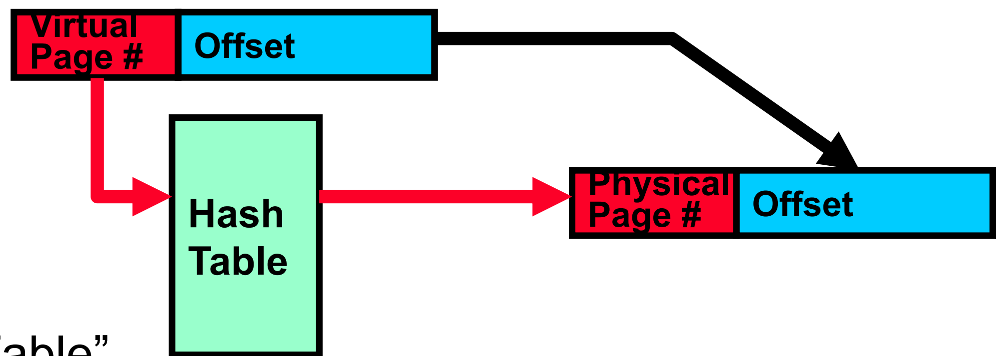


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

- Larger page sizes (2MB, 1GB) make sense since memory is now cheap
  – Great for kernel, large libraries, etc
  – Use limited primarily by internal fragmentation…

Recall: Alternative: Inverted Page Table

- With all previous examples (“Forward Page Tables”)
  – Size of page table is at least as large as amount of virtual memory allocated to processes
  – Physical memory may be much less
    » Much of process space may be out on disk or not in use

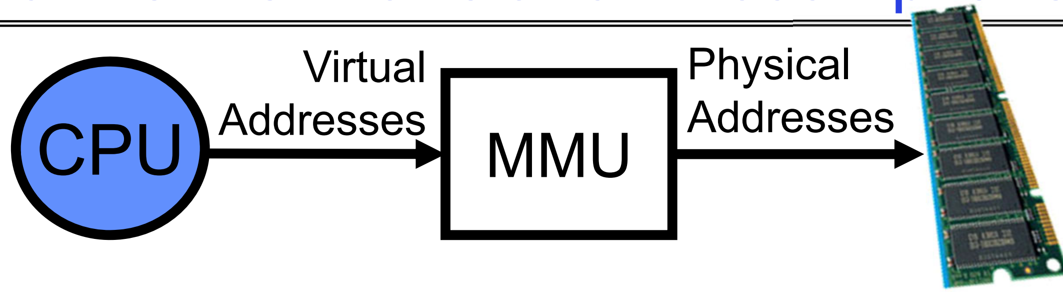


- Answer: use a hash table
  – Called an “Inverted Page Table”
  – Size is independent of virtual address space
  – Directly related to amount of physical memory
  – Very attractive option for 64-bit address spaces
    » PowerPC, UltraSPARC, IA64
- Cons:
  – Complexity of managing hash chains: Often in hardware!
  – Poor cache locality of page table

Address Translation Comparison

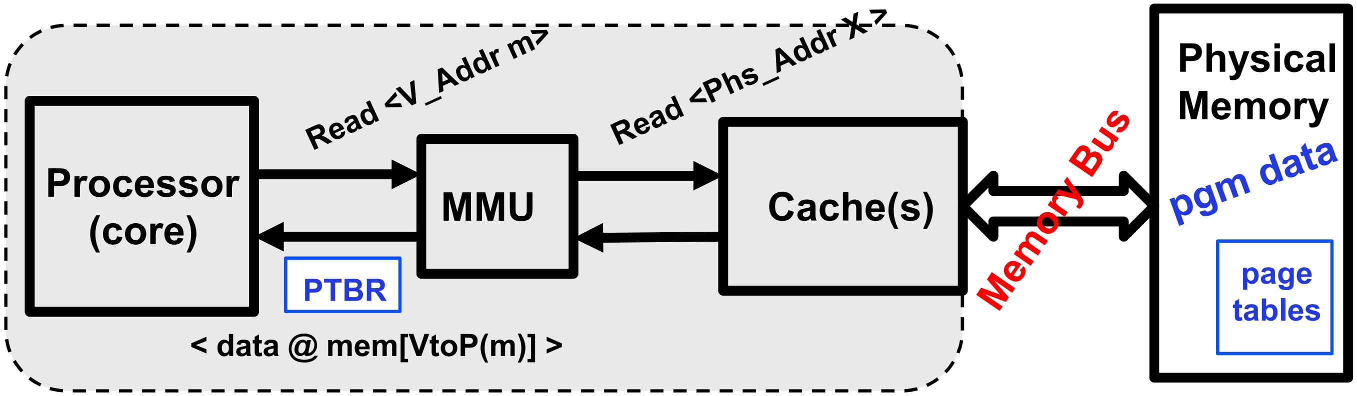
	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory Fast and easy allocation	Multiple memory references per page access
Multi-Level Paging		
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

How is the Translation Accomplished?



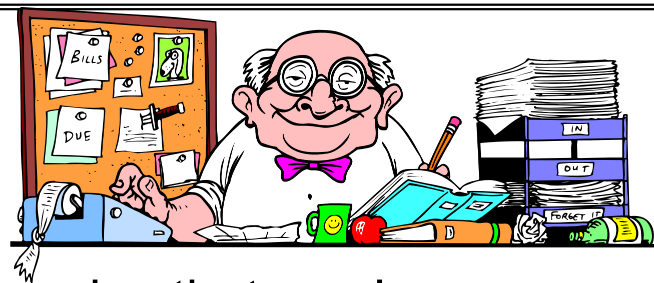
- The MMU must attempt to translate virtual address to physical address on:
  - *Every* instruction fetch, *Every* load, *Every* store
  - Generate a “Page Fault” (Trap) if it encounters invalid PTE
    - » Fault handler will decide what to do (more on this next lecture)
- What does the MMU need to do to translate an address?
  - 1-level Page Table
    - » Read PTE from memory, check valid, merge address
    - » Set “accessed” bit in PTE, Set “dirty bit” on write
  - 2-level Page Table
    - » Read and check first level
    - » Read, check, and update PTE at second level
  - N-level Page Table …
- MMU does ***page table Tree Traversal*** to translate each address
  - Turns a potentially fast memory access into a slow multi-access table traversal…
  - Need CACHING!

Where and What is the MMU ?



- The processor requests READ Virtual-Address to memory system
  – Through the MMU to the cache (to the memory)
- Some time later, the memory system responds with the data stored at the physical address (resulting from virtual → physical) translation
  – Fast on a cache hit, slow on a miss
- So what is the MMU doing?
- On every reference (I-fetch, Load, Store) read (multiple levels of) page table entries to get physical frame or FAULT
  – Through the caches to the memory
  – Then read/write the physical location

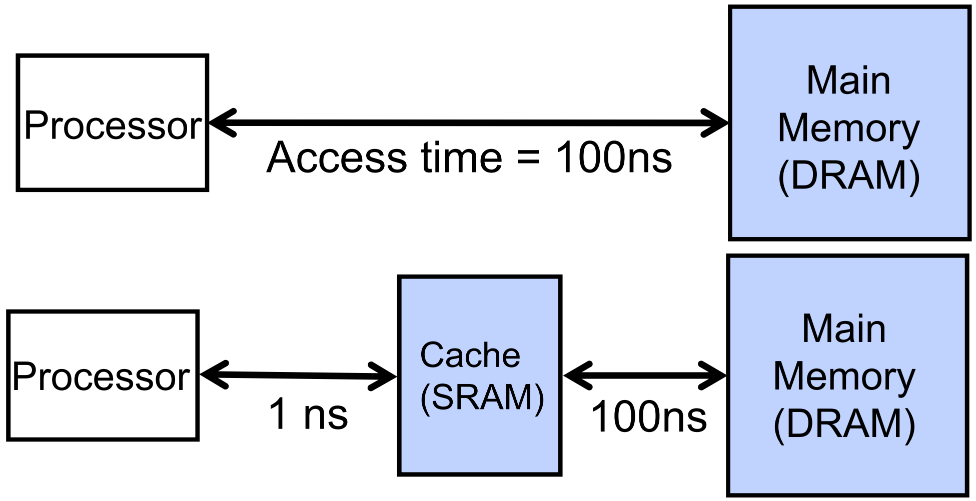
Recall: CS61c Caching Concept



- 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
- Many important OS concepts boil down to caching! We cache:
  - Pages, Files, Virtual Memory Translations, IP Addresses…

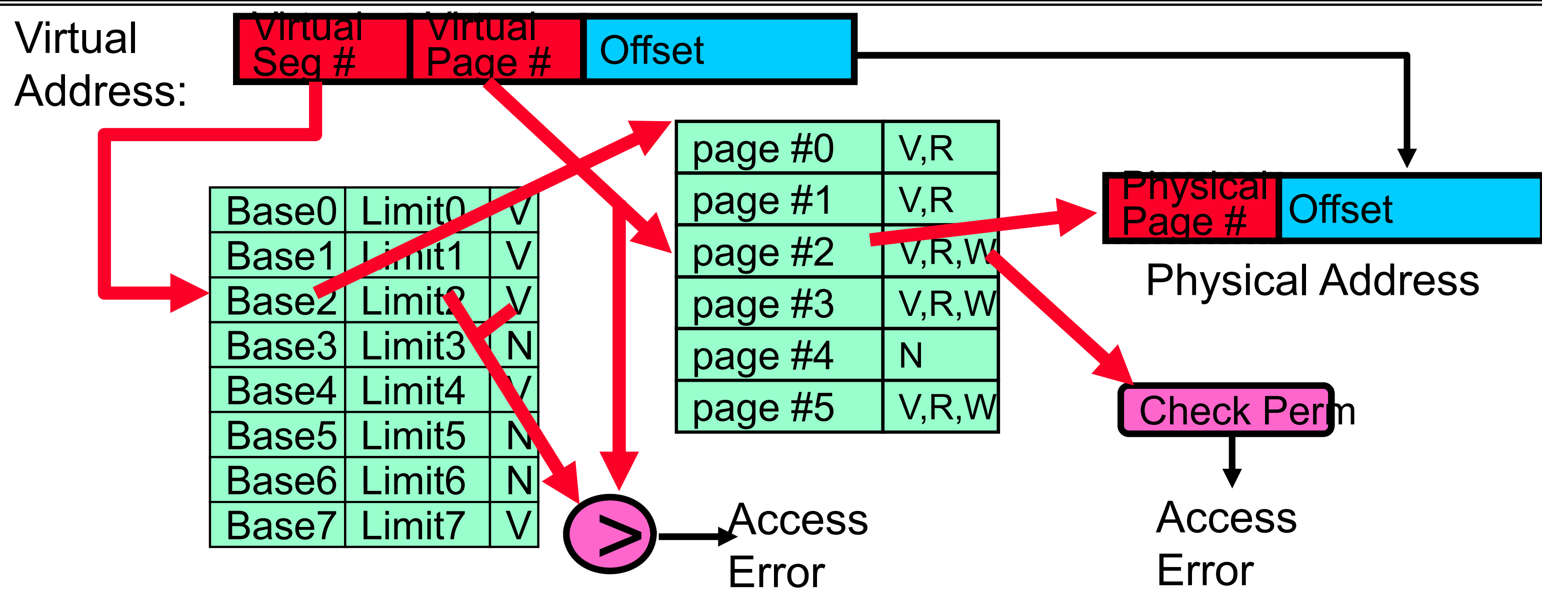
Recall: In Machine Structures (eg. 61C) …

- Hardware Caching is the key to memory system performance for CPUs:



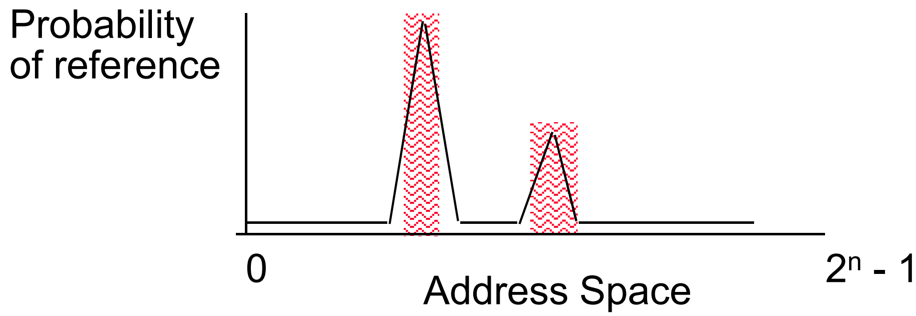
- Average Memory Access Time (AMAT)
  = (Hit Rate x HitTime) + (Miss Rate x MissTime)
- Where:
  – HitRate + MissRate = 1
  – MissTime = HitTime + MissPenalty
- Examples:
  – HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101)=11 ns
  – HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101)=2.01 ns

Another Major Reason to Deal with Caching

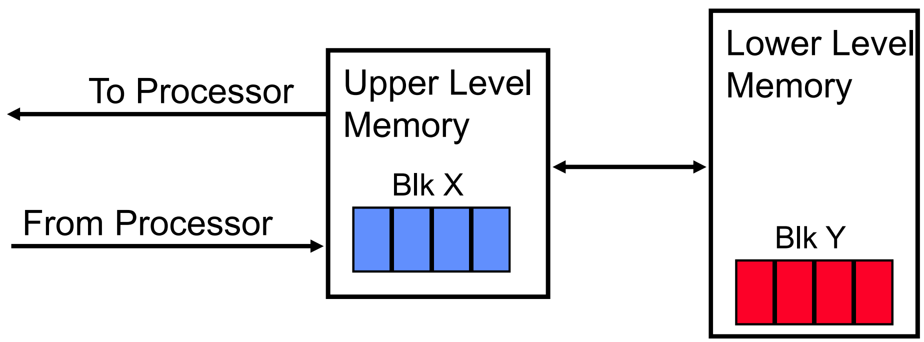


- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - Translation Cache: TLB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

Why Does Caching Help? Locality!

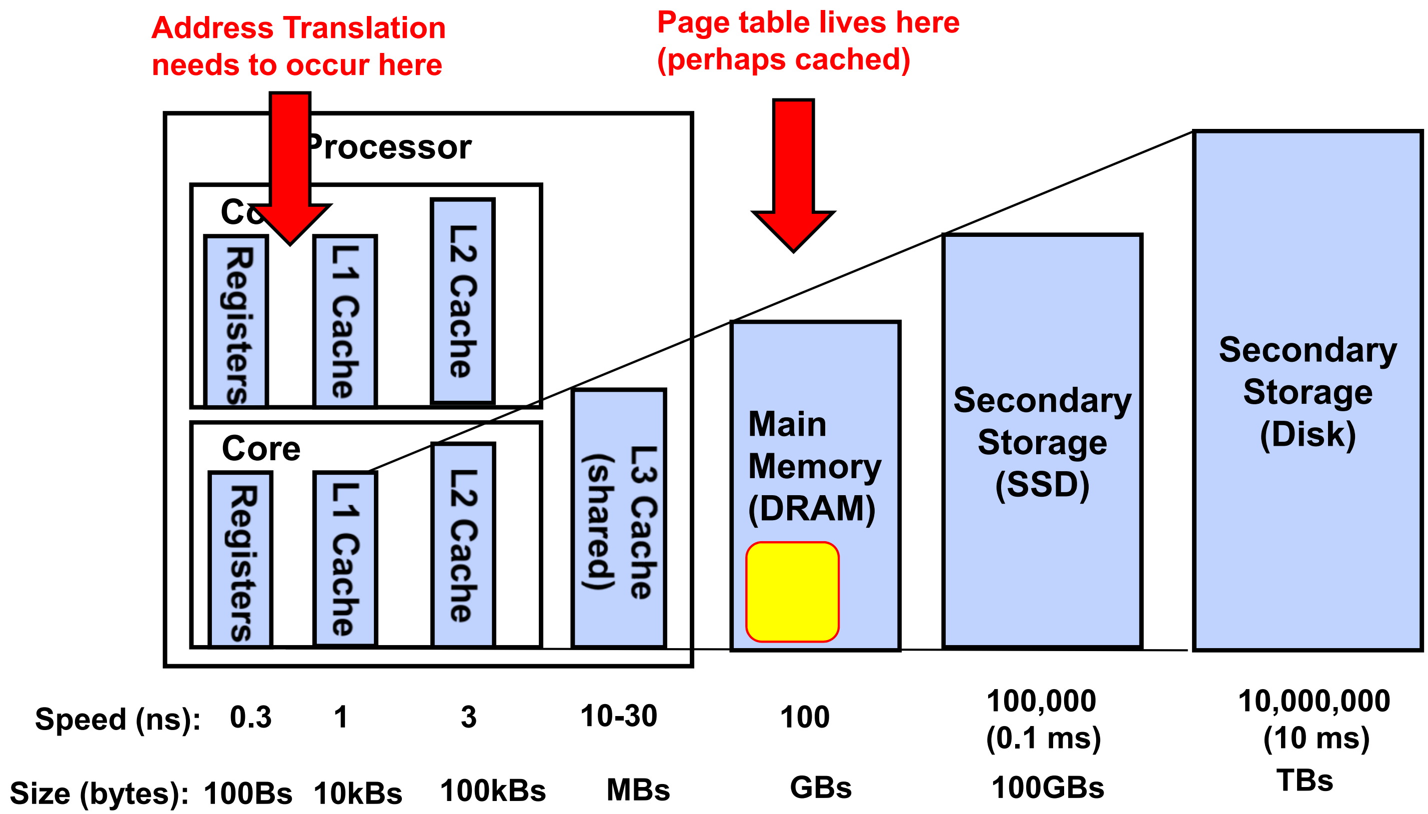


- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



Recall: Memory Hierarchy

- Caching: Take advantage of the principle of locality to:
  - Present the illusion of having as much memory as in the cheapest technology
  - Provide average speed similar to that offered by the fastest technology



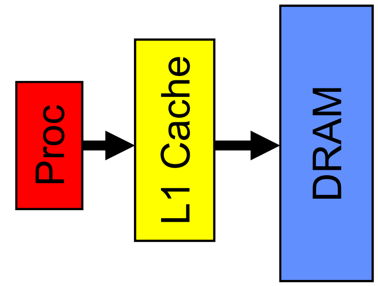
Recall 61C: Dealing with Hierarchy

- Used to compute access time probabilistically:

  $\text{AMAT} = \text{Hit Rate}_{L1} \times \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Time}_{L1}$

  $\text{Hit Rate}_{L1} + \text{Miss Rate}_{L1} = 1$
  Hit Time_{L1} = Time to get value from L1 cache.
  $\text{Miss Time}_{L1} = \text{Hit Time}_{L1} + \text{Miss Penalty}_{L1}$
  Miss Penalty_{L1} = AVG Time to get value from lower level (DRAM)

  So, $\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$

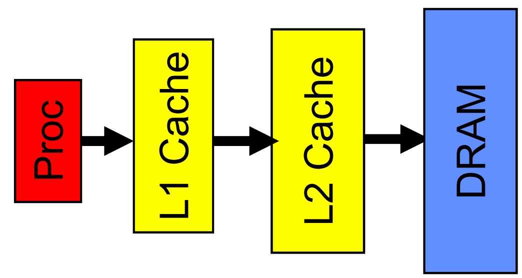


- What about more levels of hierarchy?

  $\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$

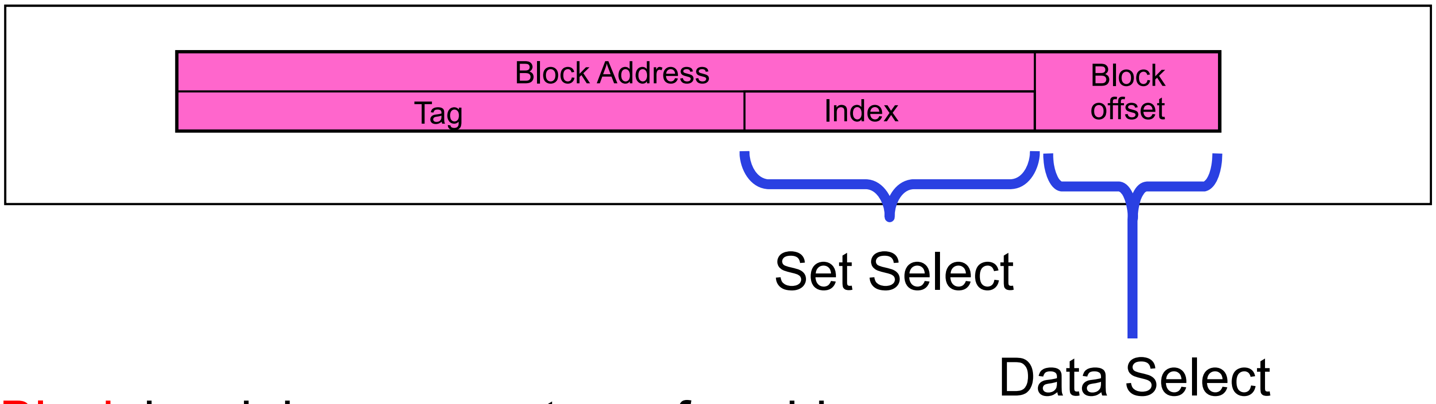
  Miss Penalty_{L1} = AVG time to get value from lower level (L2)
  $= \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$
  Miss Penalty_{L2} = Average Time to fetch from below L2 (DRAM)

  $\text{AMAT} = \text{Hit Time}_{L1} +$
  $\underline{\text{Miss Rate}_{L1}} \times (\text{Hit Time}_{L2} + \underline{\text{Miss Rate}_{L2}} \times \text{Miss Penalty}_{L2})$



- And so on … (can do this recursively for more levels!)

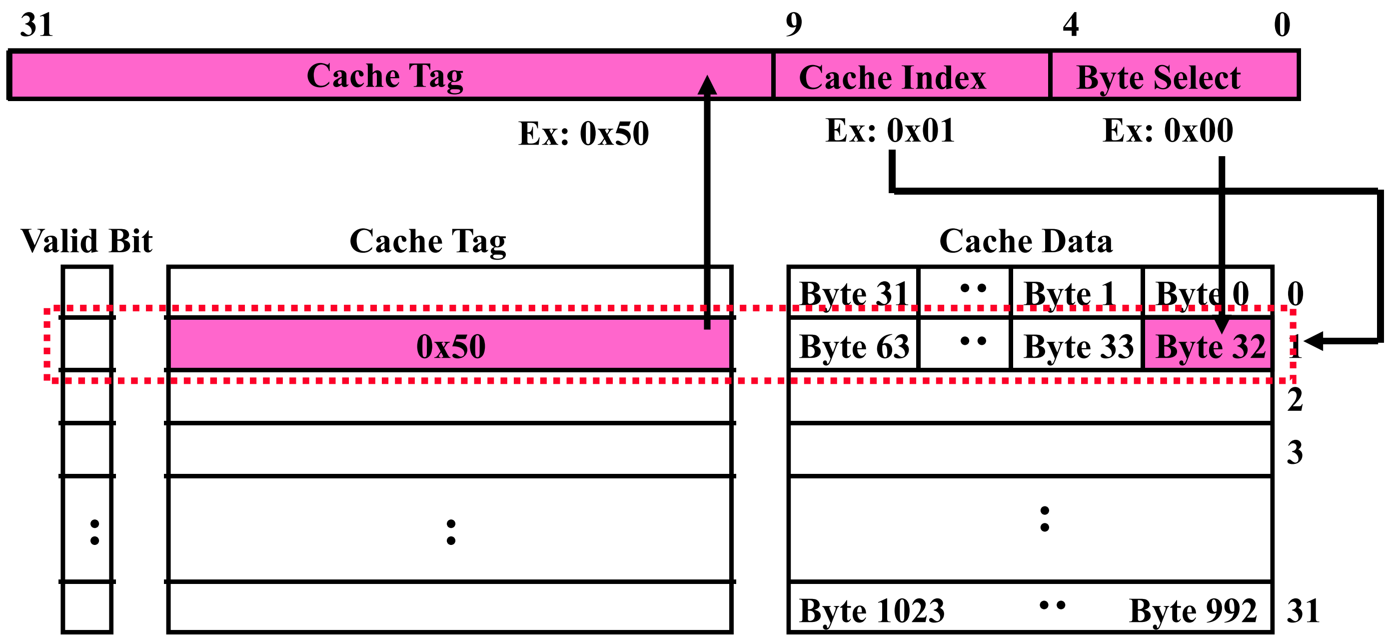
How is a Block found in a Cache?



- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field
- 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

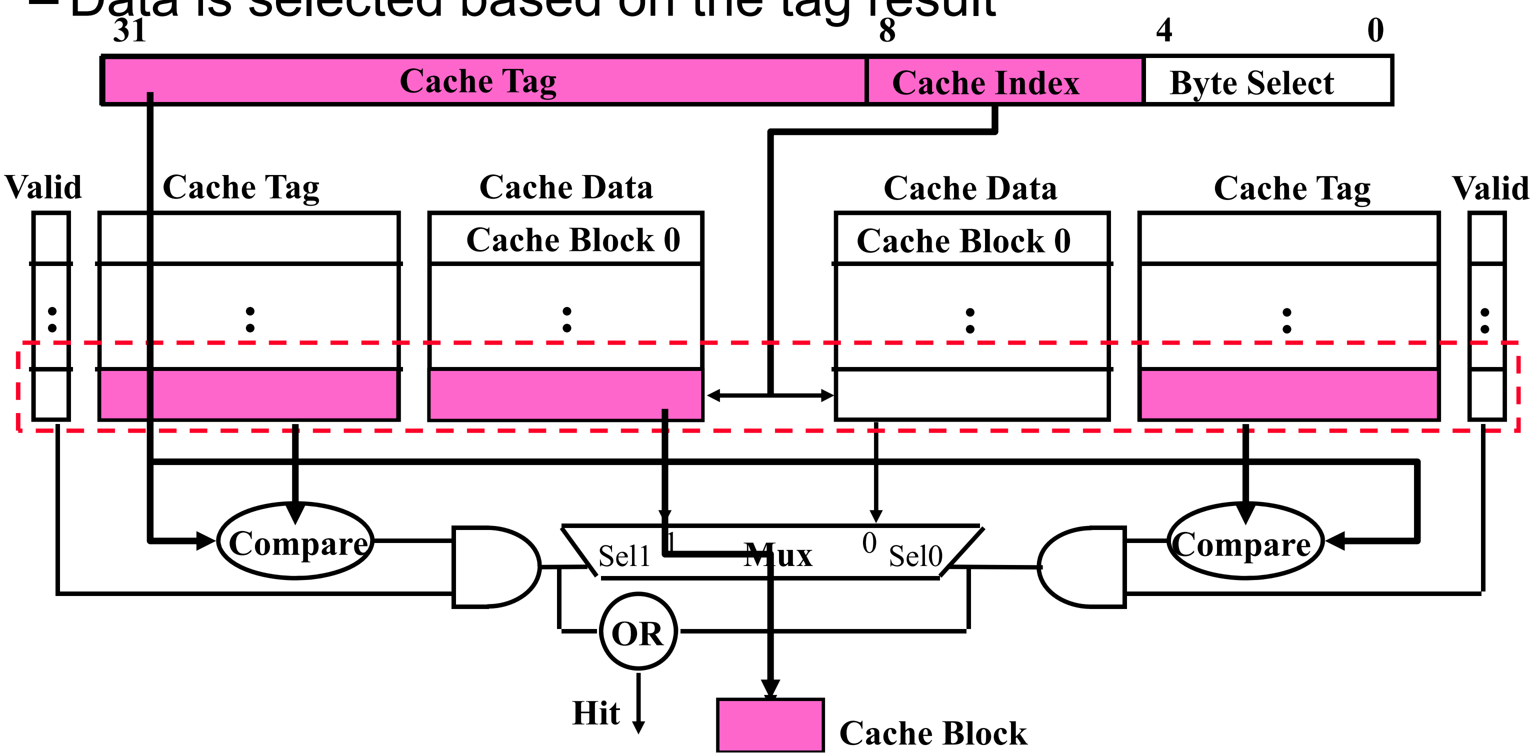
Review: Direct Mapped Cache

- 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



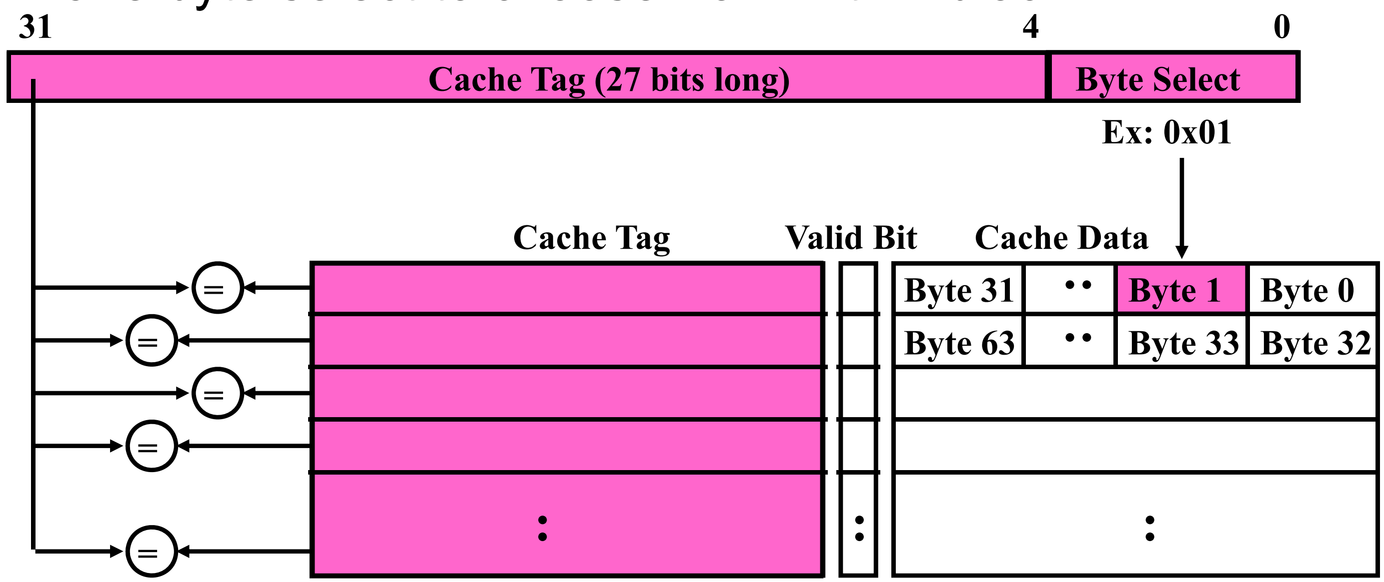
Review: Set Associative Cache

- 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



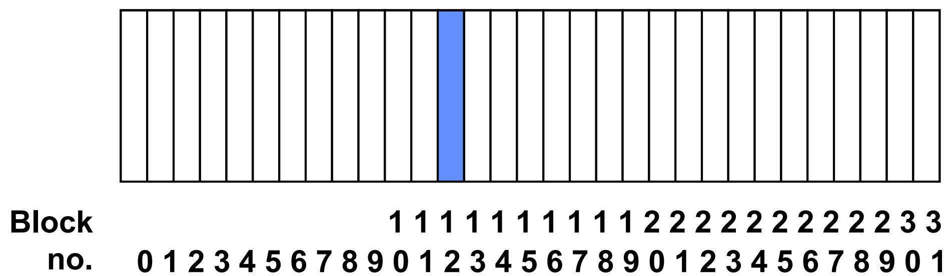
Administrivia

- Midterm 2: Thursday 8pm-10pm
  – You are responsible material up to and including today's lecture
  – Two sheets of notes: handwritten, double-sided
- Next week after exam: some extra credit for attending
  – Decided not to do it this week since people are really crazy busy
- Midterm 2 is on π Day!!!
  – 40 digits sufficient to calculate circumference of visible universe to atomic dimensions: https://www.jpl.nasa.gov/edu/news/2016/3/16/how-many-decimals-of-pi-do-we-really-need/
  – Here are 40 decimal places: 3.1415926535897932384626433832795028841971
- Best formula for PI is from Ramanujan:
  – $\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$
  – Google announced back in 2019 (3/14/19) that Emma Haruka Iwao had just calculated pi to 31,415,926,535,897 digits (new record…)

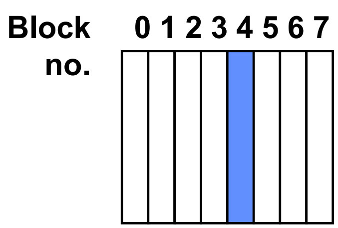
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

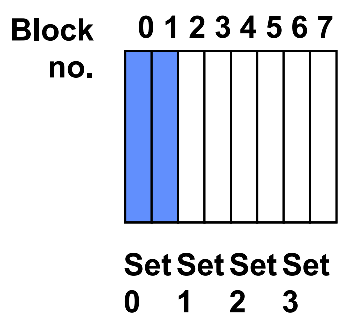
32-Block Address Space:



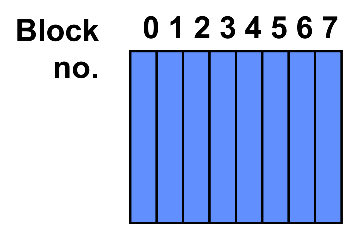
Direct mapped:
block 12 can go only into block 4 (12 mod 8)



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



Fully associative:
block 12 can go anywhere



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%

Review: What happens on a write?

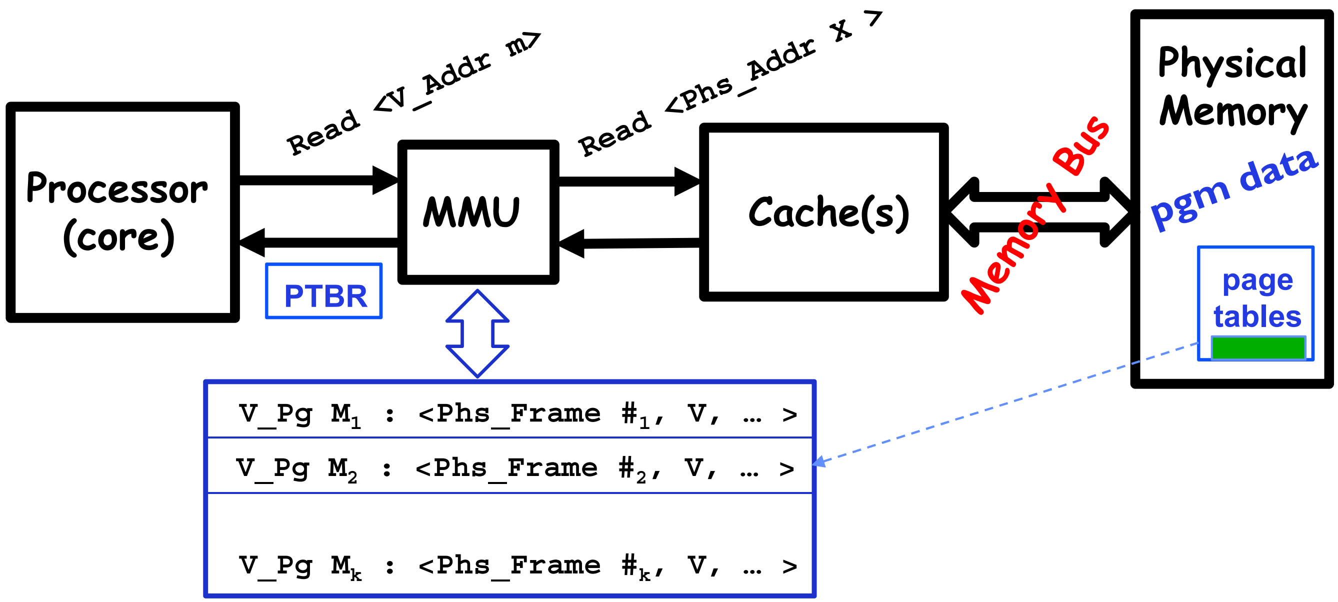
- Write through: The information is written to both the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- 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
      Read miss may require writeback of dirty data

A Summary on 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 unless you prefetch
  - Solution: Prefetch values before use
  - Note: If you run "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution 1: increase cache size
  - "Solution 2": Change (e.g. reduce) associativity to focus misses in a few places?!
    - » Consider fully-associative cache of size n: access pattern 0, 1, … n-1, n, 0, 1, …
    - » Contrast with direct mapped of size n: Fewer misses!
- 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

How do we make Address Translation Fast?

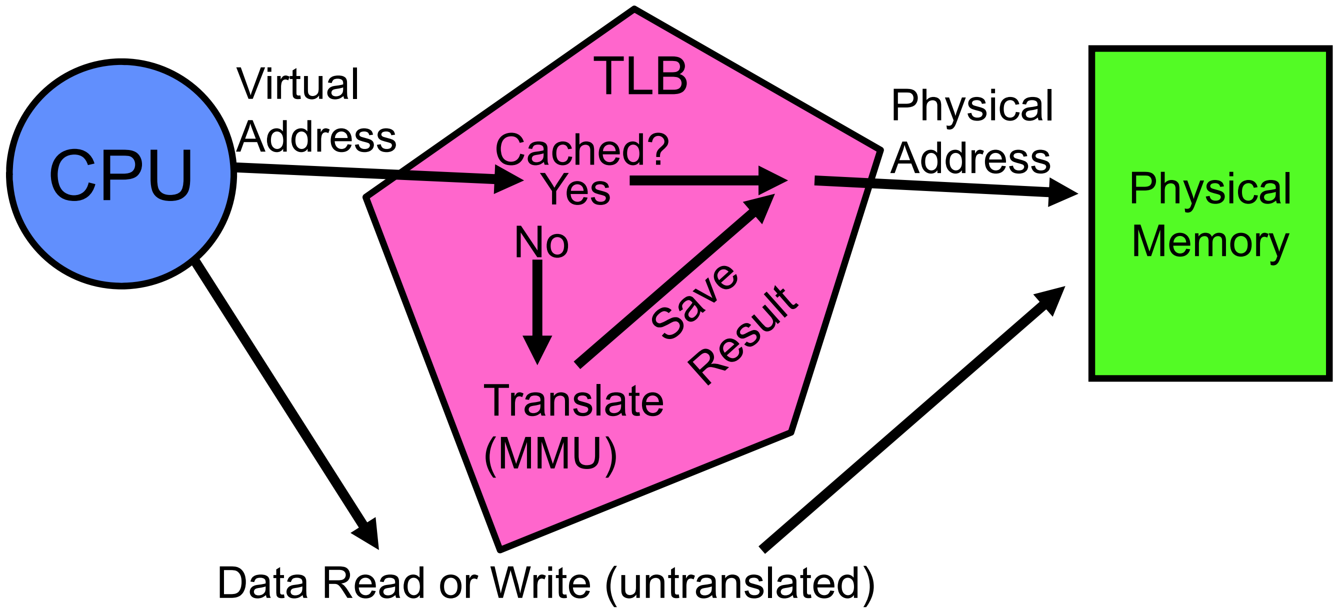
- Cache results of recent translations !
  - Different from a traditional cache
  - Cache Page Table Entries using Virtual Page # as the key



Translation Look-Aside Buffer

- Record recent Virtual Page # to Physical Frame # translation
- If present, have the physical address without reading any page tables !!!
  – Even if the translation involved multiple levels
  – Caches the end-to-end result
- Was invented by Sir Maurice Wilkes – *prior to caches*
  – When you come up with a new concept, you get to name it!
  – People realized "if it's good for page tables, why not the rest of the data in memory?"
- On a *TLB miss*, the page tables may be cached, so only go to memory when both miss

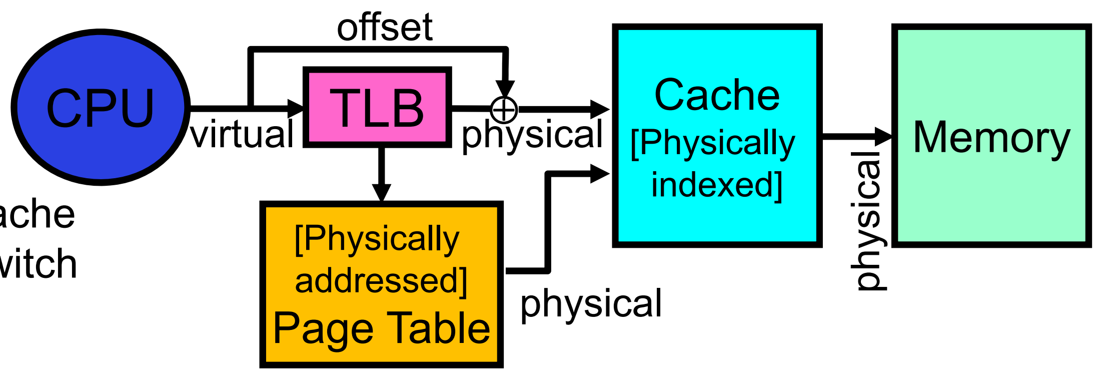
Caching Applied to Address Translation



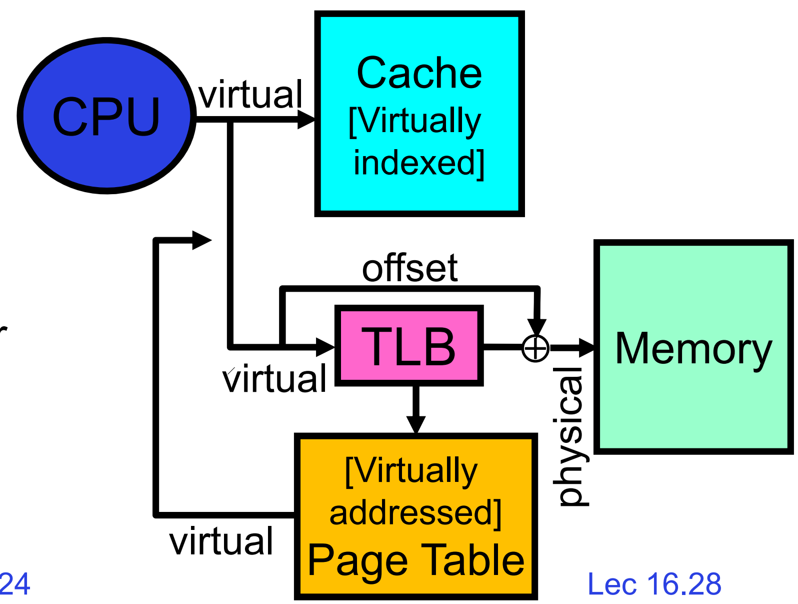
- Question is one of page locality: does it exist?
  – Instruction accesses spend a lot of time on same page (accesses are sequential)
  – Stack accesses have definite locality of reference
  – Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  – Sure: multiple levels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

Physically-Indexed vs Virtually-Indexed Caches

- Physically-Indexed, Physically-Tagged
  - Address handed to cache after translation
  - Page Table in physical memory (so that it can be cached)
  - Benefits:
    - » Every piece of data has single place in cache
    - » Cache can stay unchanged on context switch
  - Challenges:
    - » TLB is in critical path of lookup!
  - Pretty Common today (e.g. x86 processors)

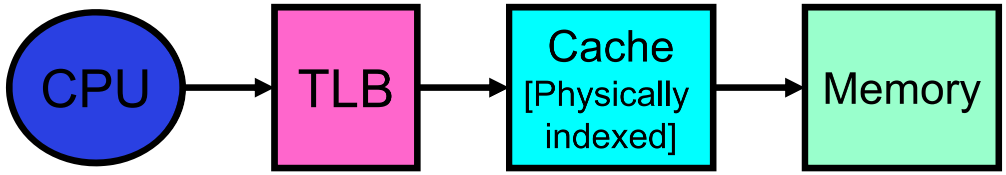


- Virtually-Indexed, Virtually-Tagged or Physically-Tagged
  - Address handed to cache before translation
  - Page Table in virtual memory (so that it can be cached); Only last level of Page Table points to physical memory.
  - Benefits:
    - » TLB not in critical path of lookup, so system can be faster
  - Challenges:
    - » Same data could be mapped in multiple places of cache
    - » May need to flush cache on context switch



- We will stick with Physically Indexed Caches for now!

What TLB Organization Makes Sense?



- For Physically Indexed/Tagged, 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 MissTime extremely high! (Page Table traversal)
  - Cost of Conflict (Miss Time) is high
  - Hit Time – dictated by clock cycle
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of virtual page number 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

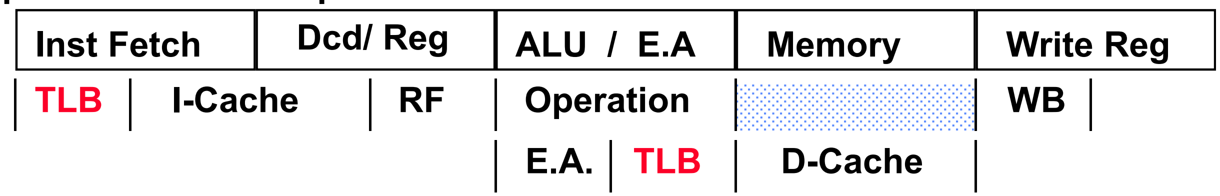
TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries (larger now)
  - Not very big, can support higher associativity
- Small TLBs 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"
- 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

Making physically-indexed caches fast: Fit into Pipeline!

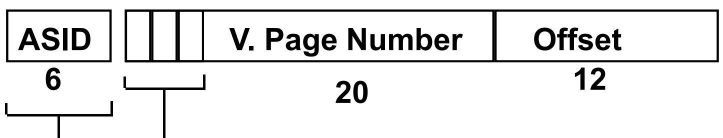
Example: MIPS R3000 Pipeline



TLB

64 entry, on-chip, fully associative, software TLB fault handler

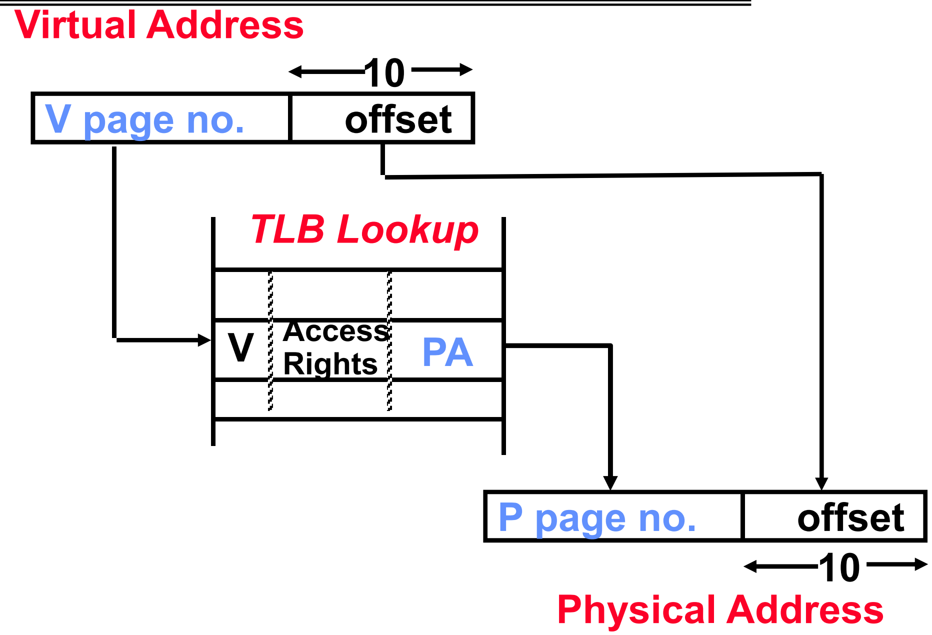
Virtual Address Space



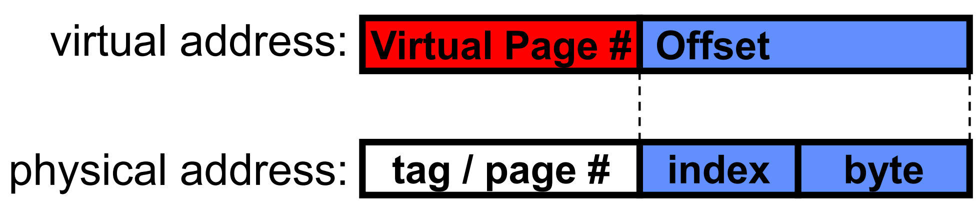
0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

Further reducing translation time for physically-indexed caches

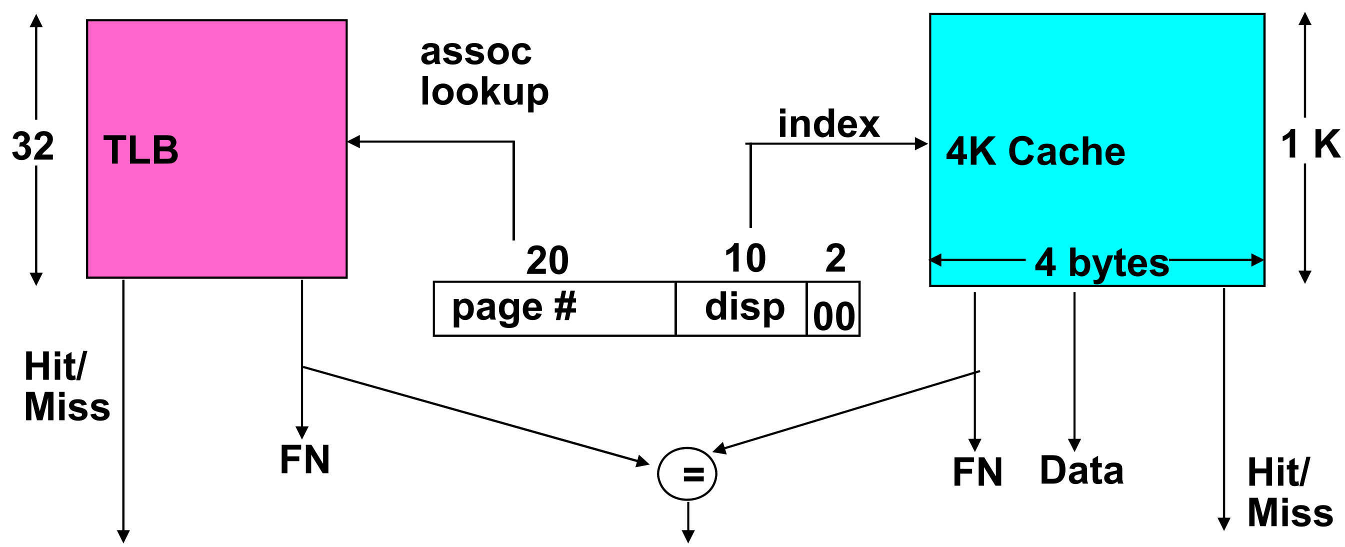


- As described, TLB lookup is in serial with cache lookup
  - Consequently, speed of TLB can impact speed of access to cache
- Machines with TLBs go one step further: overlap TLB lookup with cache access
  - Works because offset available early
  - Offset in virtual address exactly covers the “cache index” and “byte select”
  - Thus can select the cached byte(s) in parallel to perform address translation



Overlapping Cache and TLB access

- Here is how this might work with a 4K cache:

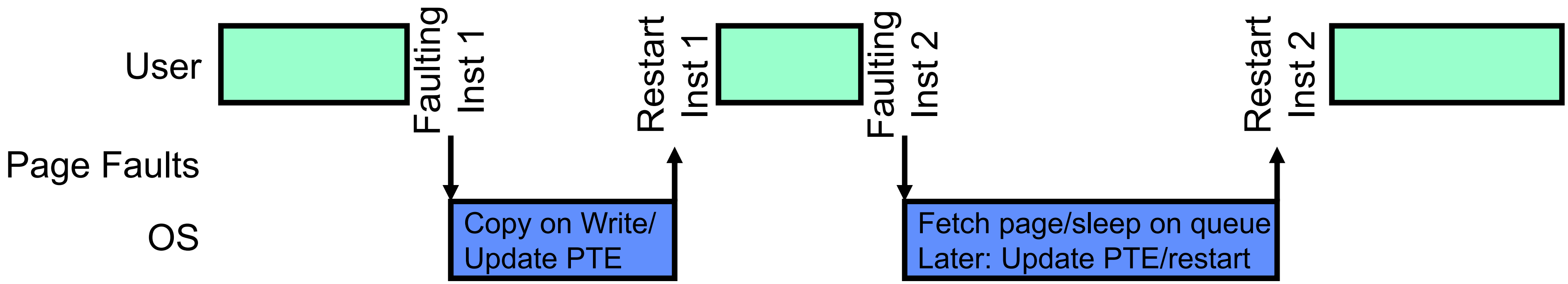


- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- As discussed earlier, Virtual Caches would make this faster
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

What Actually Happens on a TLB Miss?

- Hardware traversed page tables (x86, many others):
  - 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
- 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: Page fault



- How to transparently restart faulting instructions?
  - (Consider load or store that gets 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?

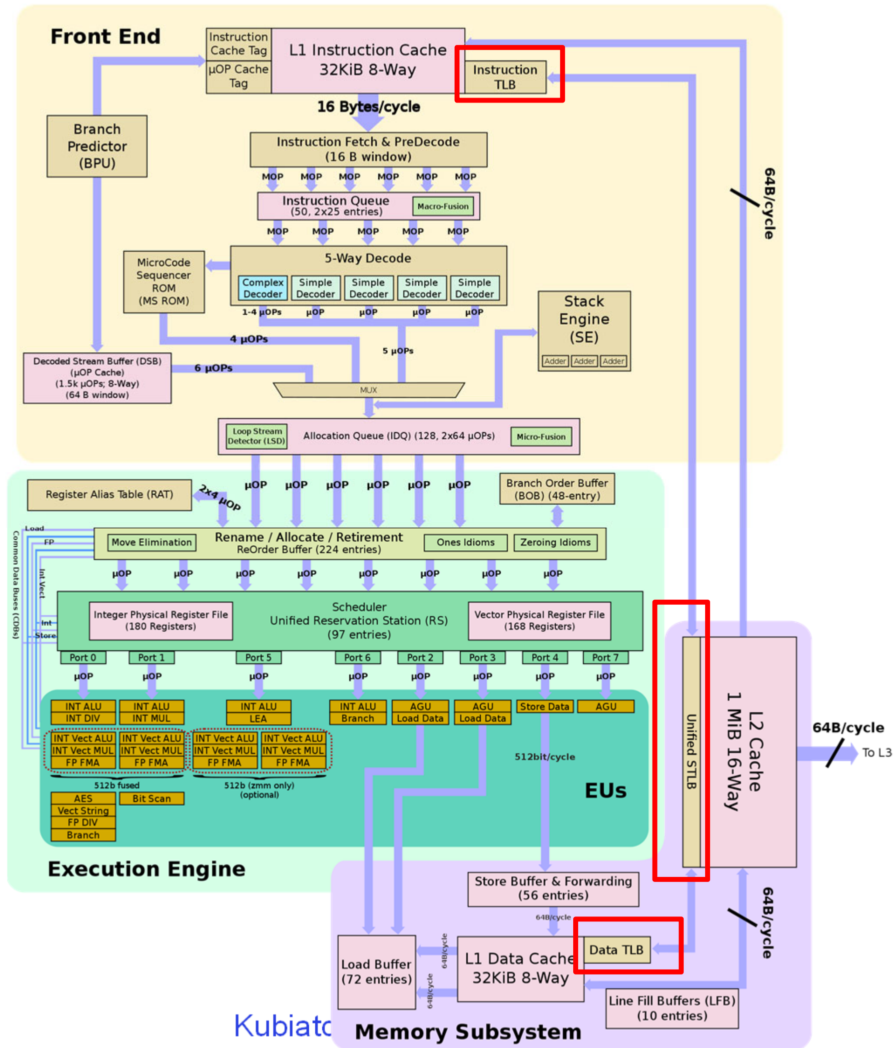
Consider weird things that can happen

- What if an instruction has side effects?
  - Options:
    - » Unwind side-effects (easy to restart)
    - » Finish off side-effects (messy!)
  - Example 1: `mov (sp)+,10`
    - » What if page fault occurs when write to stack pointer?
    - » Did `sp` get incremented before or after the page fault?
  - Example 2: `strcpy (r1), (r2)`
    - » Source and destination overlap: can't unwind in principle!
    - » IBM S/370 and VAX solution: execute twice – once read-only
- What about "RISC" processors?
  - For instance delayed branches?
    - » Example:
      ```
      bne somewhere
      ld r1,(sp)
      ```
    - » Restart after page fault: need two PCs, PC and nPC!
  - Delayed exceptions:
    - » Example:
      ```
      div r1, r2, r3
      ld r1, (sp)
      ```
    - » What if takes many cycles to discover divide by zero, but load has already caused page fault?

Precise Exceptions

- Precise $\Rightarrow$ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - x86 takes this position
- Imprecise $\Rightarrow$ system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Recent Intel x86 (Skylake, Cascade Lake)



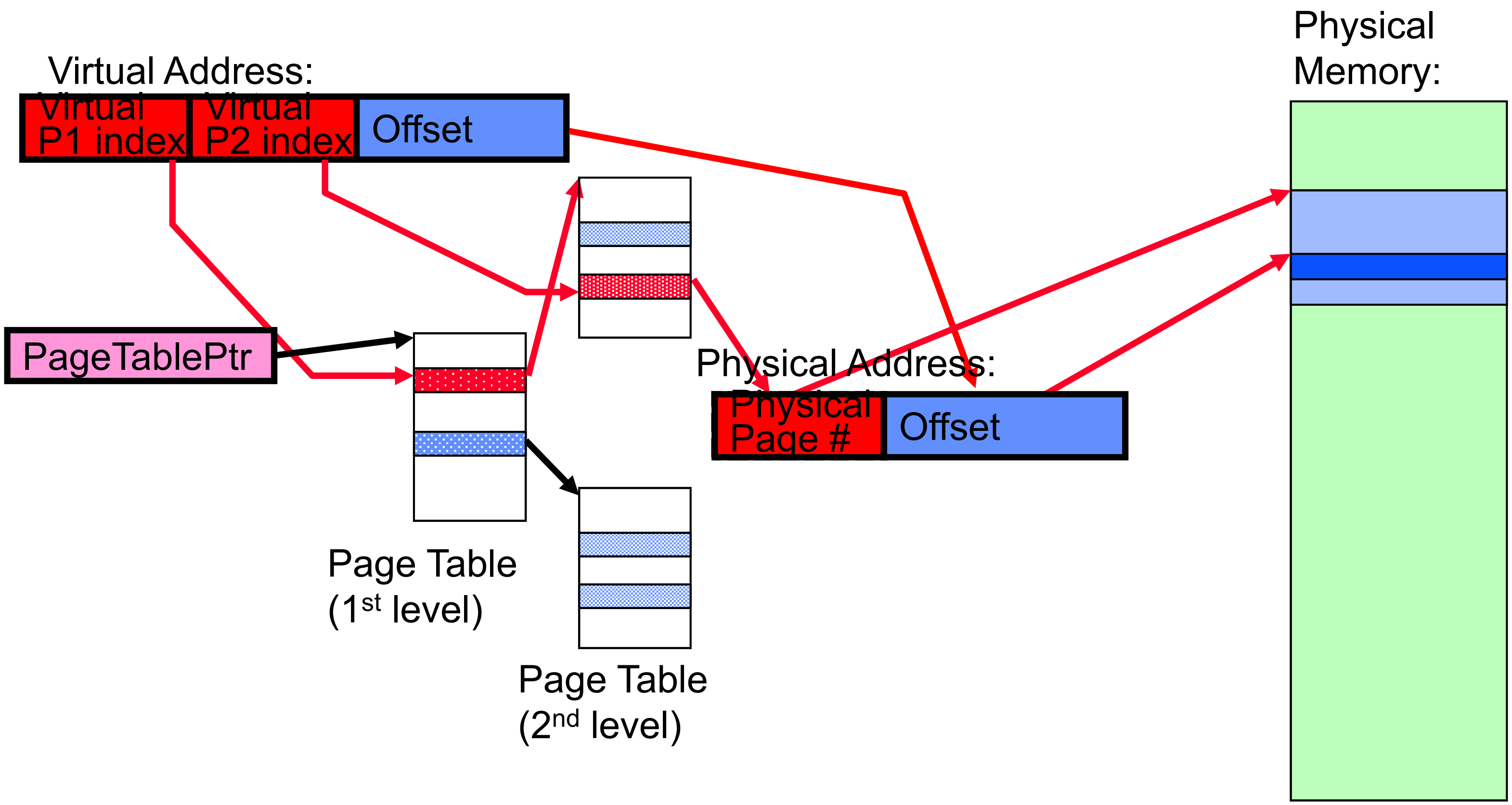
Recent Example: Memory Hierarchy

- Caches (all 64 B line size)
  - L1 I-Cache: 32 KiB/core, 8-way set assoc.
  - L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  - L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  - L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency
- TLB
  - L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    - » 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  - L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    - » 32 entries; 4-way set associative, 2 MiB / 4 MiB page translations:
    - » 4 entries; 4-way associative, 1G page translations:
  - L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    - » 16 entries; 4-way set associative, 1 GiB page translations:

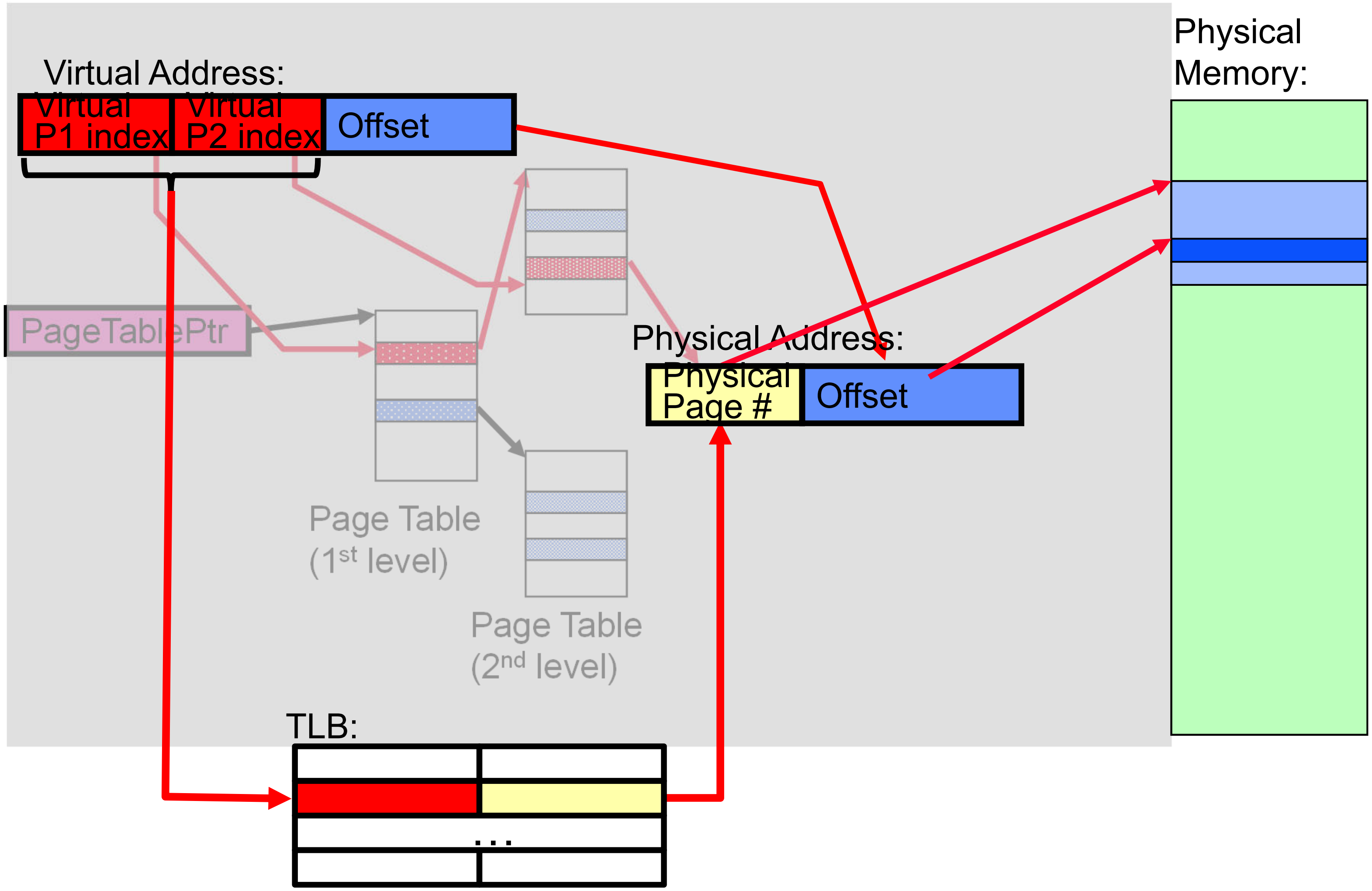
What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate ("Flush") 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"
- Aside: with Virtually-Indexed, Virtually-Tagged cache, need to flush cache!
  - Everyone has their own version of the address "0" and can't distinguish them
  - This is one advantage of Virtually-Indexed, Physically-Tagged caches..

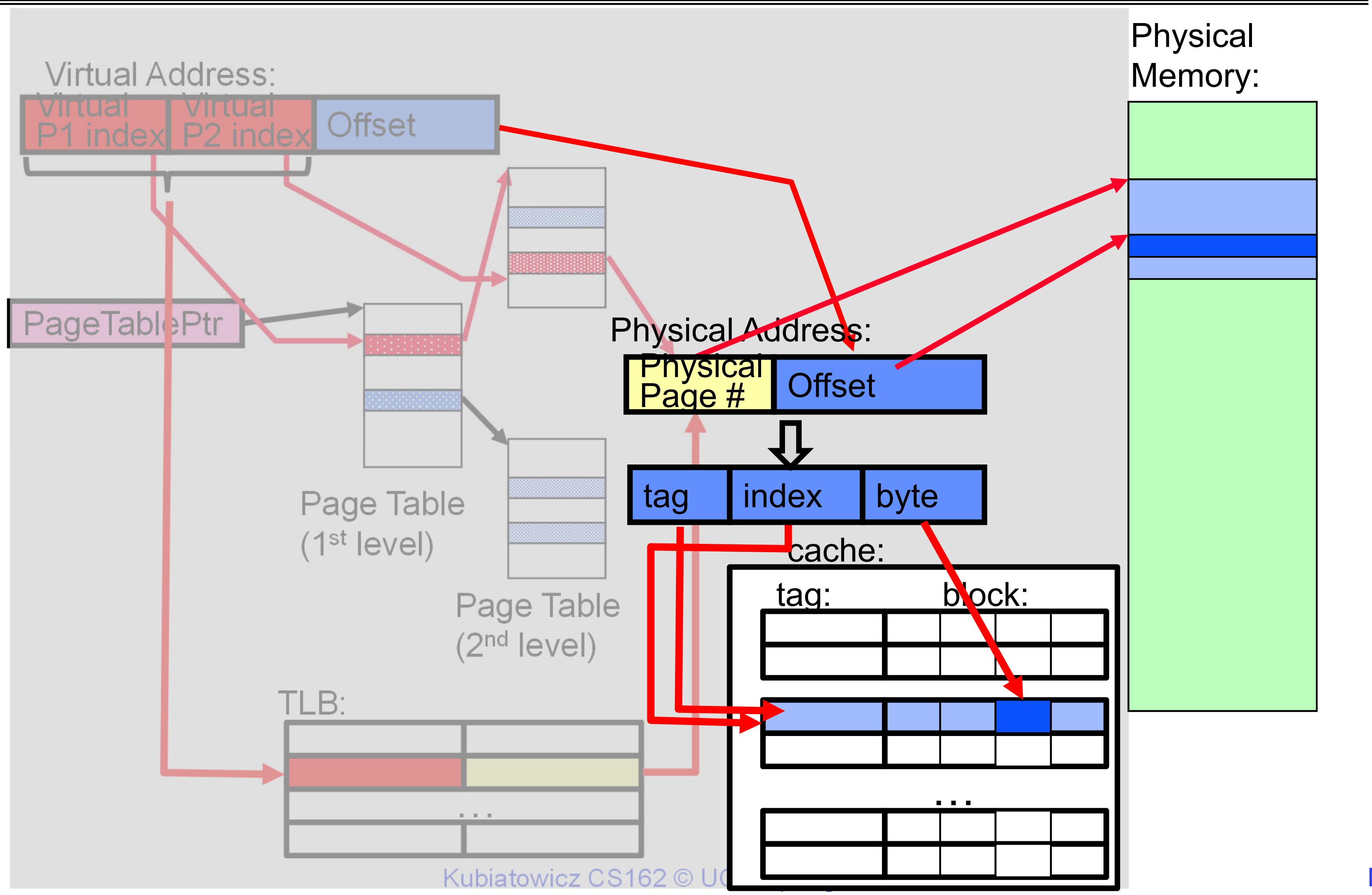
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: Cache

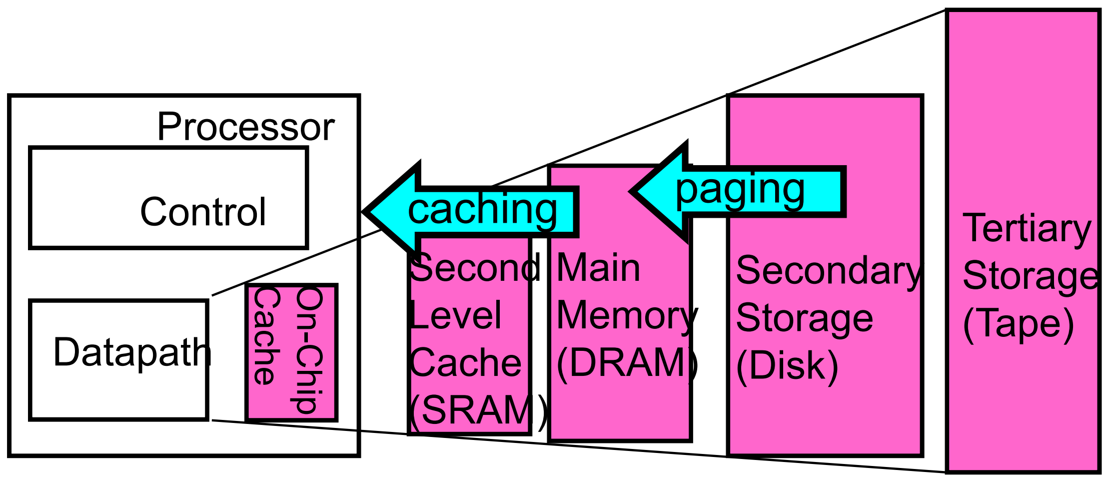


Page Fault Handling

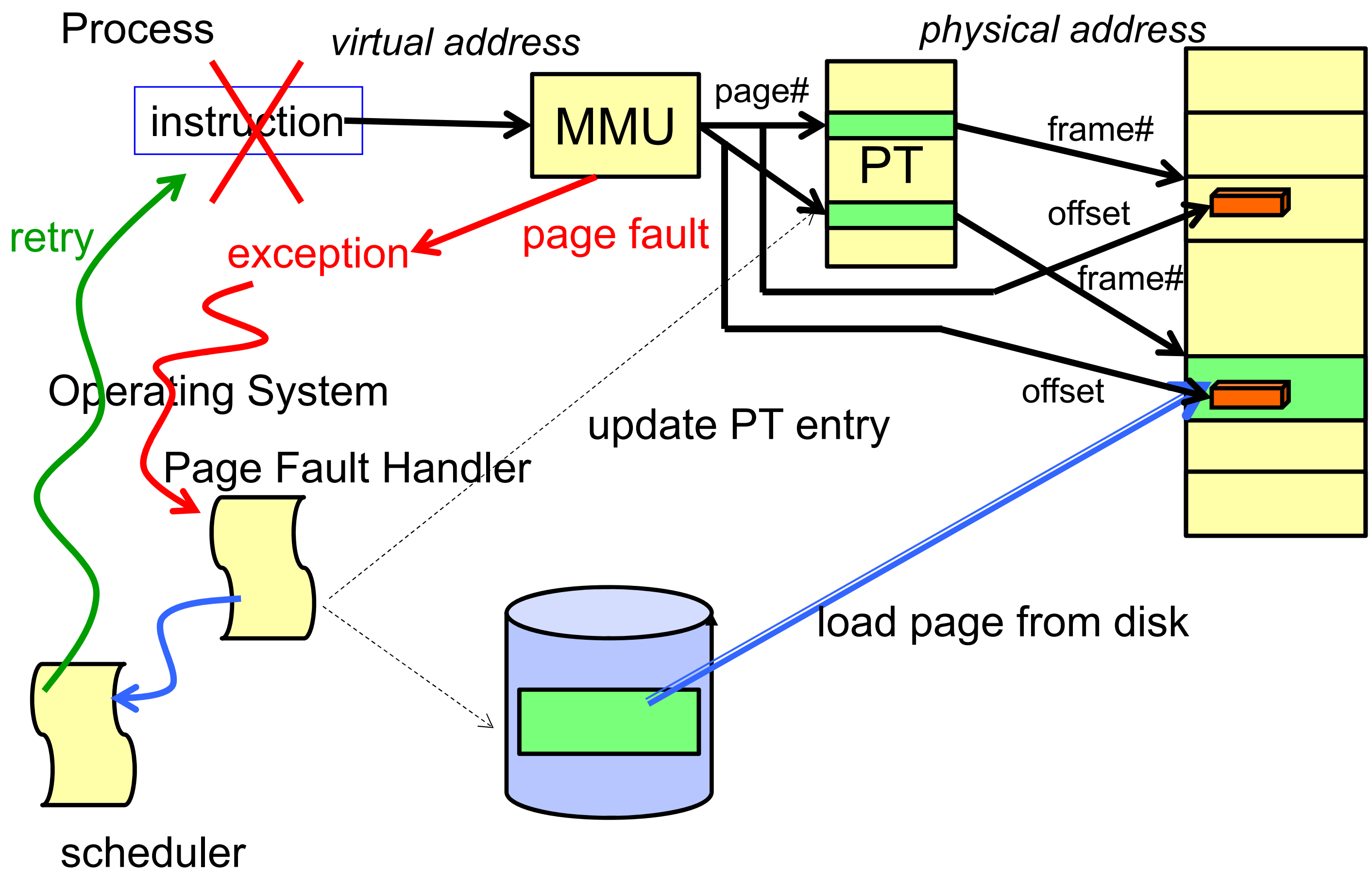
- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Privilege Level Violation, Access violation, or does not exist
  - Causes an Fault / Trap
    - » Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically terminate the process
- Other Page Faults engage operating system to fix the situation and retry the instruction
  - Allocate an additional stack page, or
  - Make the page accessible – (Copy on Write),
  - Bring page in from secondary storage to memory – demand paging
- Fundamental inversion of the hardware / software boundary
  - Need to execute software to allow hardware to proceed!

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



Page Fault ⇒ Demand Paging



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

Summary (2/2)

- "Translation Lookaside Buffer" (TLB)
  - Small number of PTEs and optional process IDs (< 512)
  - Often Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
- Demand Paging: Treating the DRAM as a cache on disk
  - Page table tracks which pages are in memory
  - Any attempt to access a page that is not in memory generates a page fault, which causes OS to bring missing page into memory