CS162
Operating Systems and Systems Programming
Lecture 13

Address Translation to Virtual Memory

October 15, 2019

Prof. David E. Culler

http://cs162.eecs.Berkeley.edu

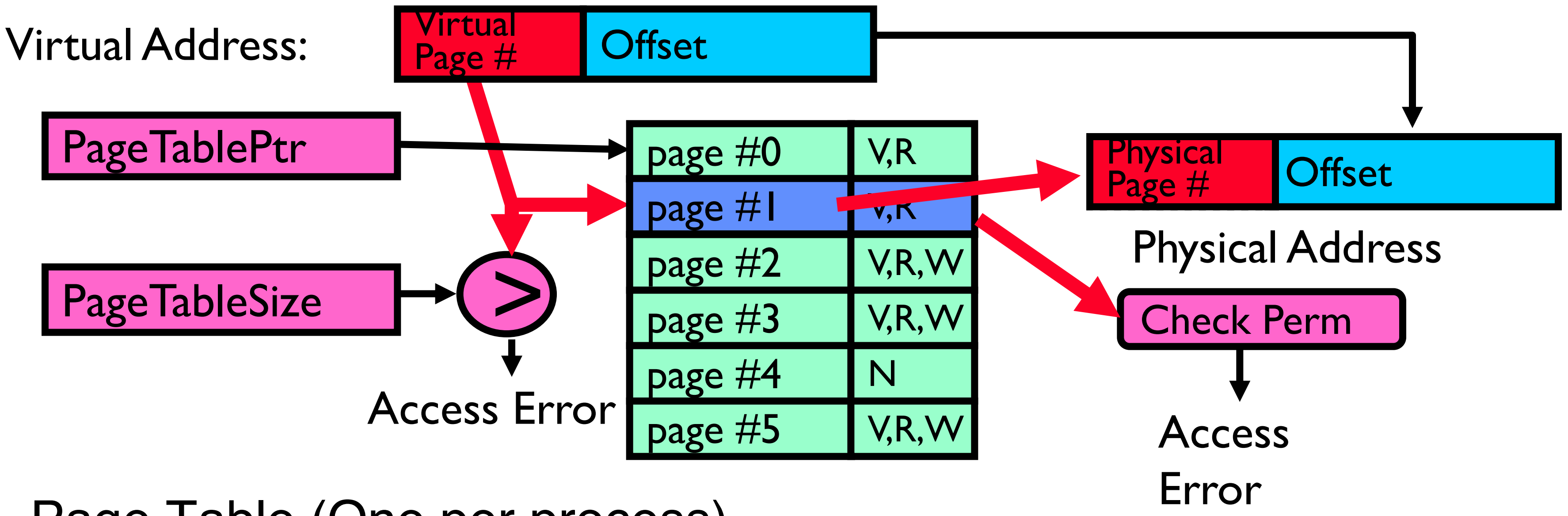
Read: 3Easy Ch19
A&D Ch 9

Post-Blackout Logistics

- Midterm Exam rescheduled for Friday 10/18 6-8 pm
  – Individuals have been assigned to rooms
  – Alternate midterm in process (piazza & surveys)
- HW 3 submission extended to 10/25
- Project 2 released in substantially reduced form
  – No LMFQ scheduler in PINTOS, python notebook study instead
  – Design doc extended, Single Checkpoint

- Opportunity to think broadly about system design
  – Infrastructure goal is to be taken for granted (just works)
  – "only as strong as the weakest link" => Reliable system out of unreliable (or not completely reliable) parts
    » E.g., Internet best-effort delivery with end-host reliability (TCP)
    » OS handles most errors in I/O devices unbeknownst to Apps
    » Virtualizes memory to gracefully extend to utilize disks

Recall: Basic Paging

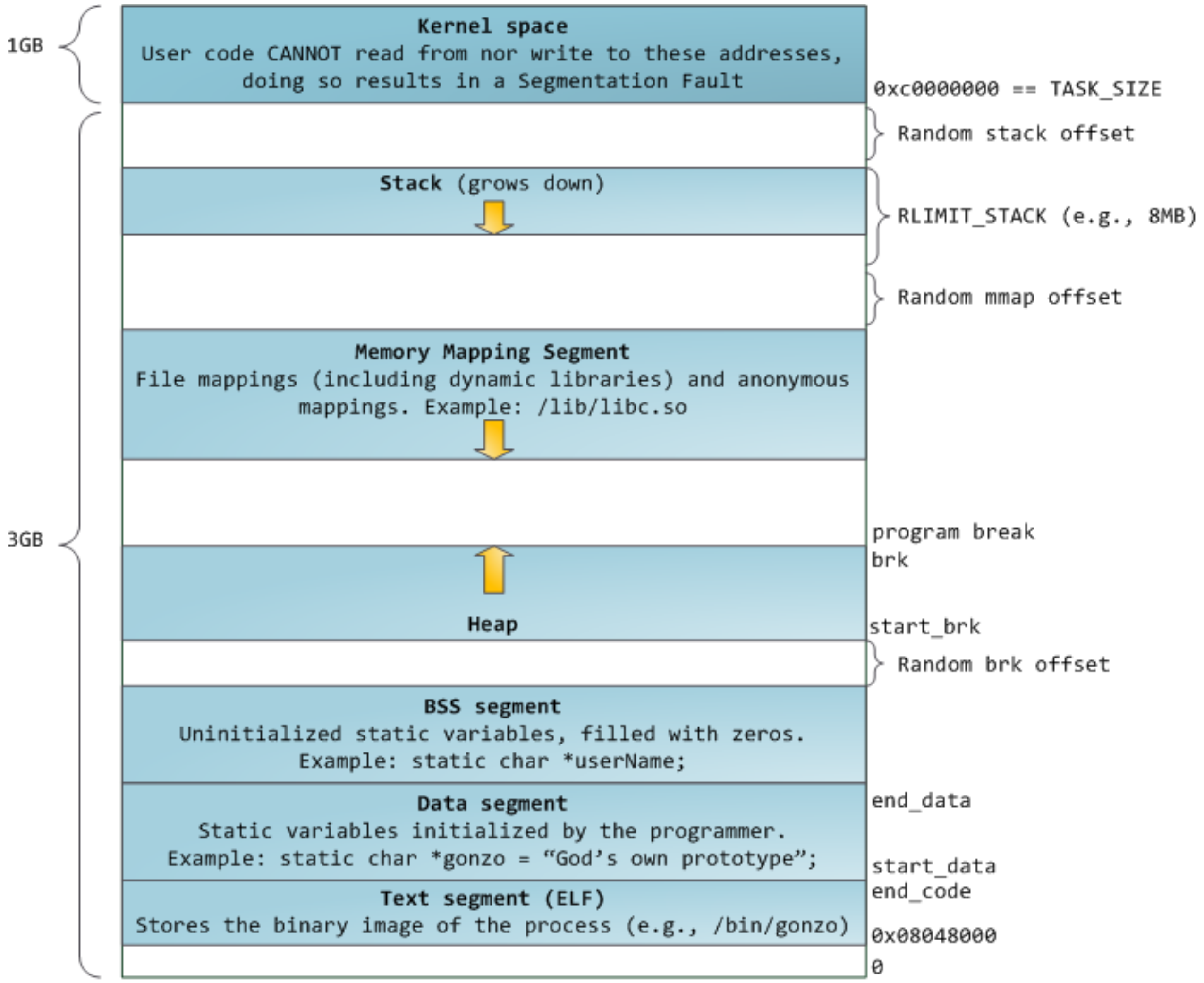


- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

Recall: Page Table Discussion

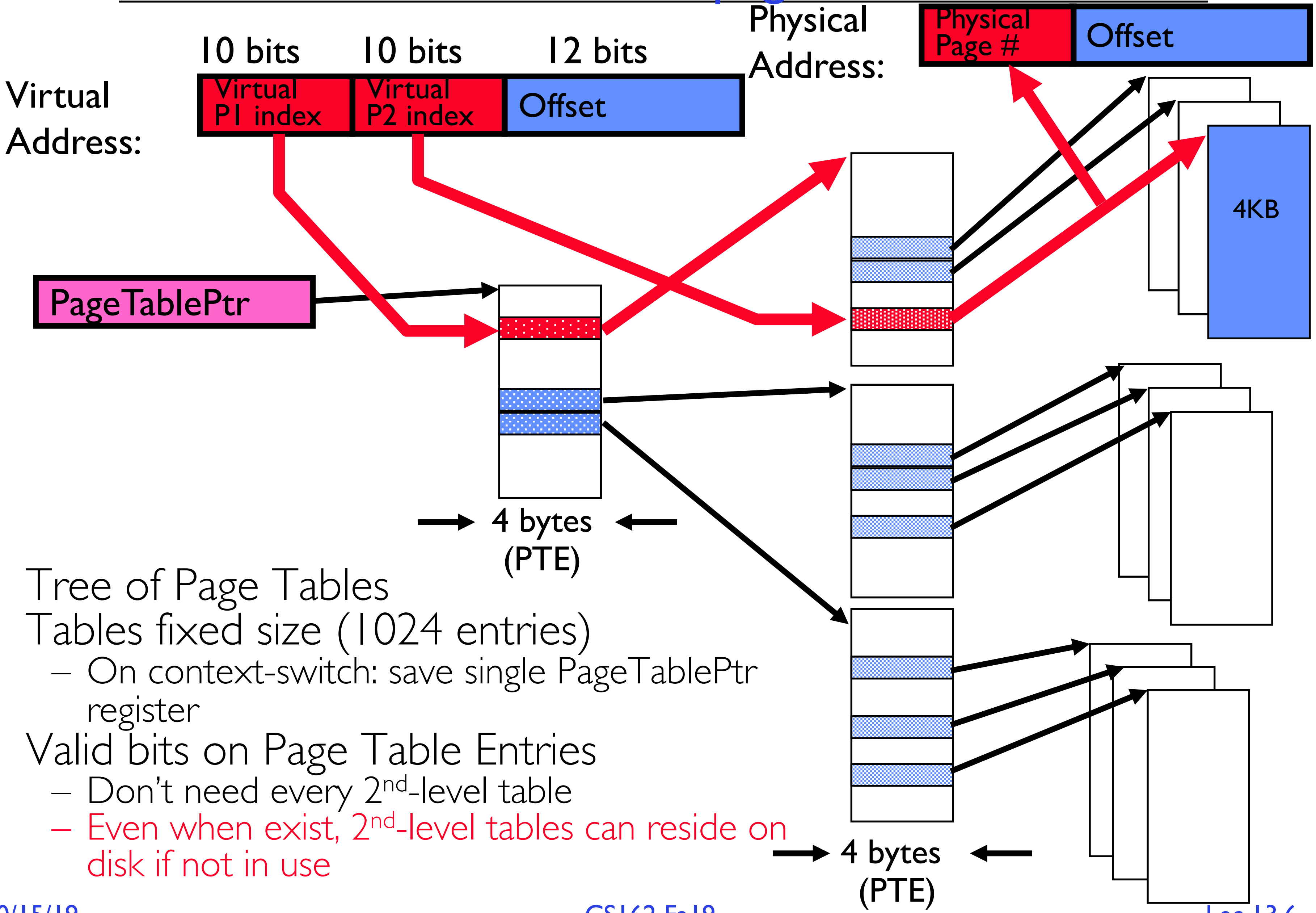
- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - Simple memory allocation
    - Supports sparse address space, Easy to share
  - Con: Size & Cost. What if address space is sparse?
    - E.g., on UNIX, code starts at ~0, stack starts at $(2^{31}-1)$
    - With 4K pages, need a million page table entries!
  - Con: What if table really big?
    - Not all pages used all the time $\Rightarrow$ would be nice to have working set of page table in memory
- How about multi-level paging or combining paging and segmentation?

Recall: Memory Layout for Linux 32-bit


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

http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

Fix for sparse address space: The two-level page table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- 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

What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - "Pointer to" (address of) next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - 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**

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

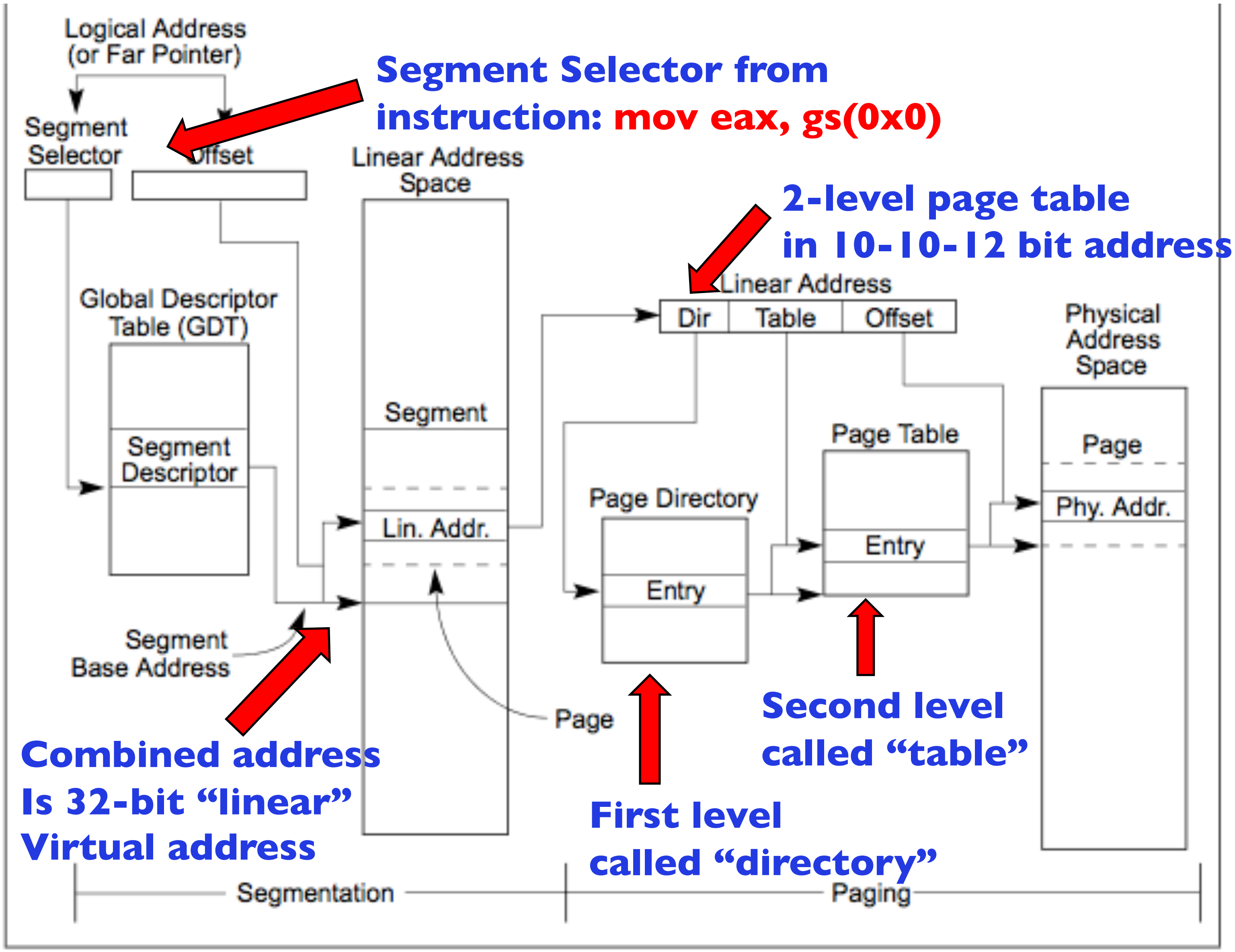
Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory (e.g., on disk)
  - Validity checked first
    - » OS can use other bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - » Address spaces disconnected after child created
  - How to do this cheaply?
    - » Make copy of parent's page tables (point at same memory)
    - » Mark entries in both sets of page tables as read-only
    - » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

Multi-level Translation Analysis

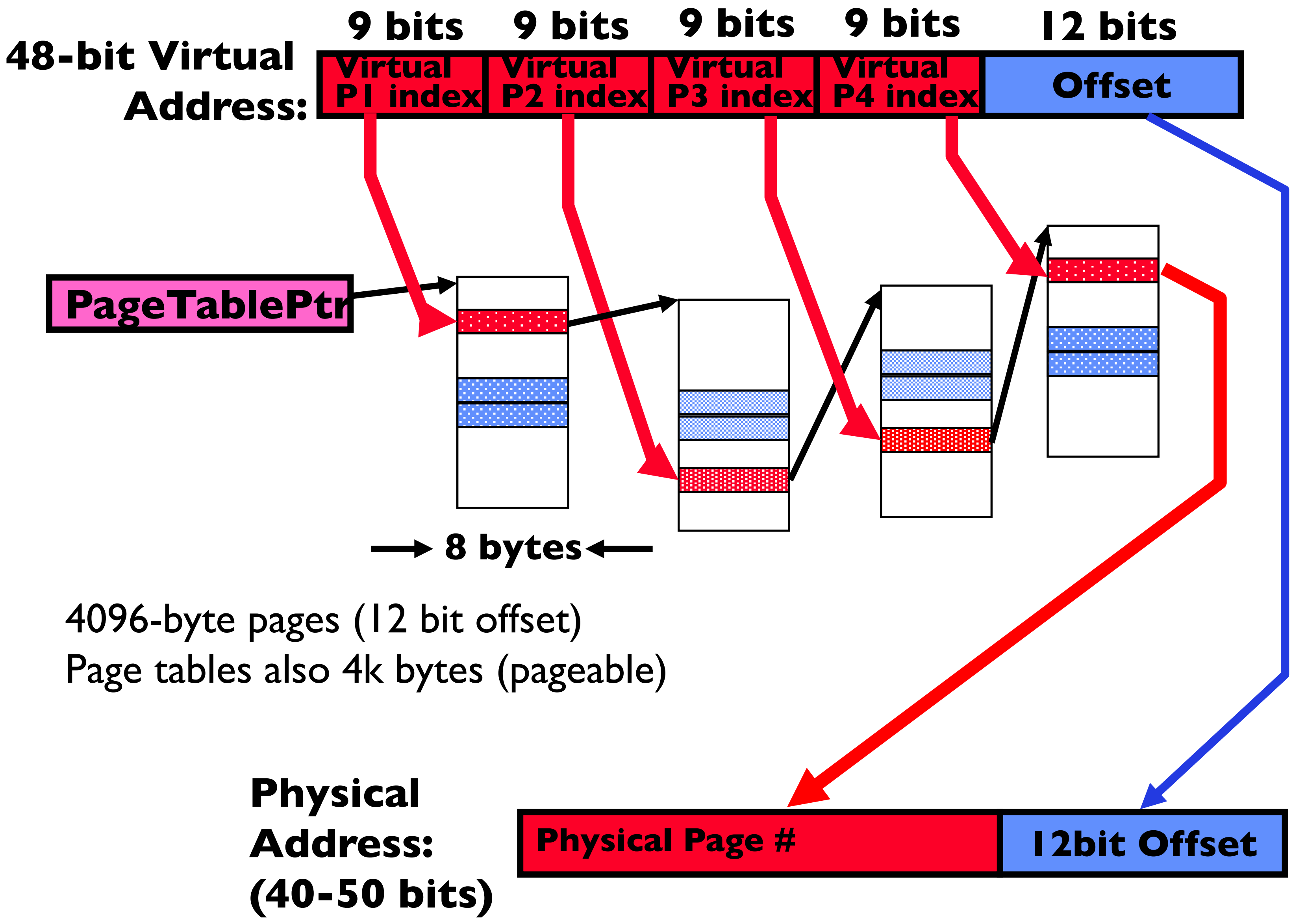
- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other wards, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

Recall: Making it real: X86 Memory model with segmentation (16/32-bit)



Logical Address (or Far Pointer)
Segment Selector
Offset
Segment Selector from instruction: mov eax, gs(0x0)
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
2-level page table in 10-10-12 bit address
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Combined address Is 32-bit "linear" Virtual address
First level called "directory"
Second level called "table"
Segmentation
Paging

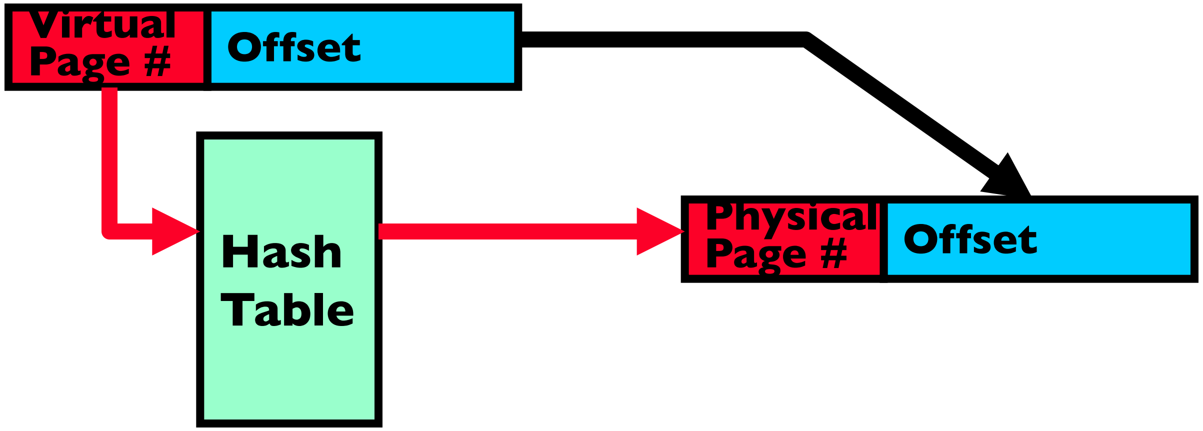
X86_64: Four-level page table!



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

Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least proportional to 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

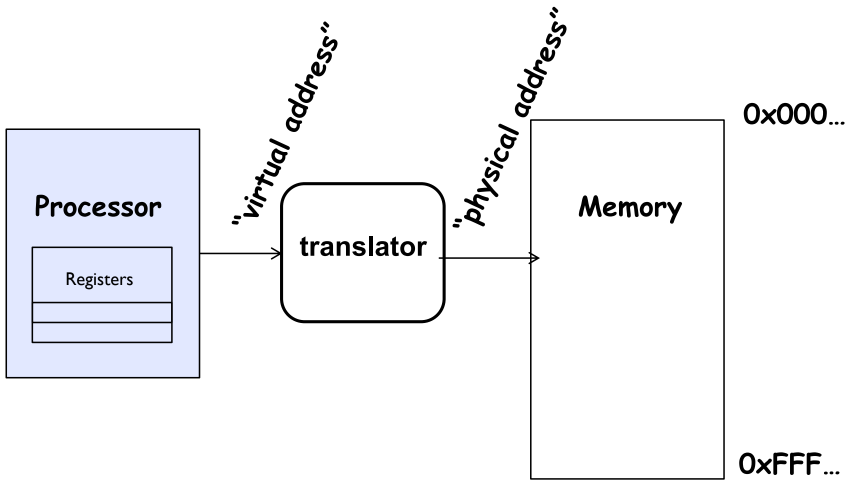
	Advantages	**Disadvantages**
Simple Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory Internal fragmentation
Paged segmentation	Table size ~ # of pages in virtual memory, fast easy allocation	Multiple memory references per page access
Multi-level pages		
Inverted Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

Recall: Dual-Mode Operation

- Can a process modify its own translation tables?
  - NO!
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bit(s) in control register only accessible in Kernel mode
  - Kernel can easily switch to user mode; User program must invoke an exception of some sort to get back to kernel mode (more in moment)
- Note that x86 model actually has more modes:
  - Traditionally, four "rings" representing priority; most OSes use only two:
    - » Ring 0 $\Rightarrow$ Kernel mode, Ring 3 $\Rightarrow$ User mode
  - Newer processors have additional mode for hypervisor ("Ring -1")
- Certain operations restricted to Kernel mode:
  - Including modifying the page table (CR3 in x86), and GDT/LDT
  - Have to transition into Kernel mode before you can change them

Two Critical Issues in Address Translation

- What to do if the translation fails? - a page fault (later)
- How to translate addresses fast enough?
  - Every instruction fetch
  - Plus every load / store
  - EVERY MEMORY REFERENCE !
  - More than one translation for EVERY instruction



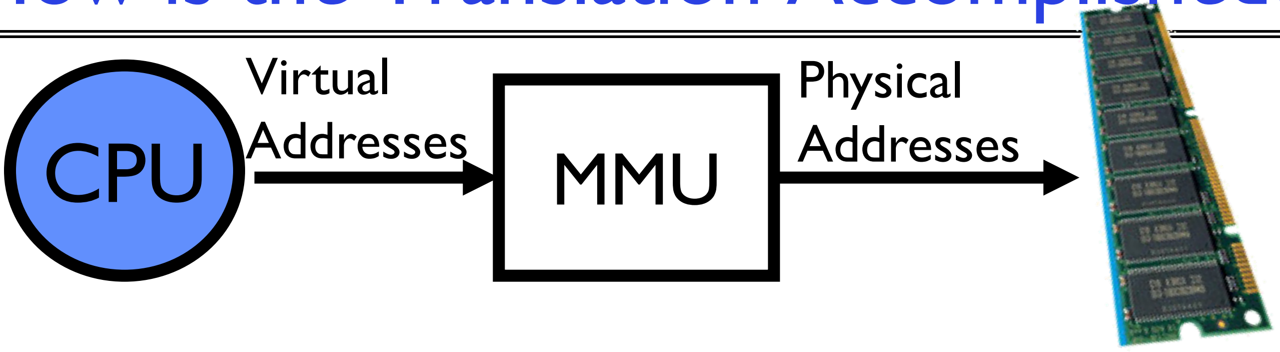
Review: What is in a PTE?

- 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 same format previous slide (10, 10, 12-bit offset)
  - 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

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

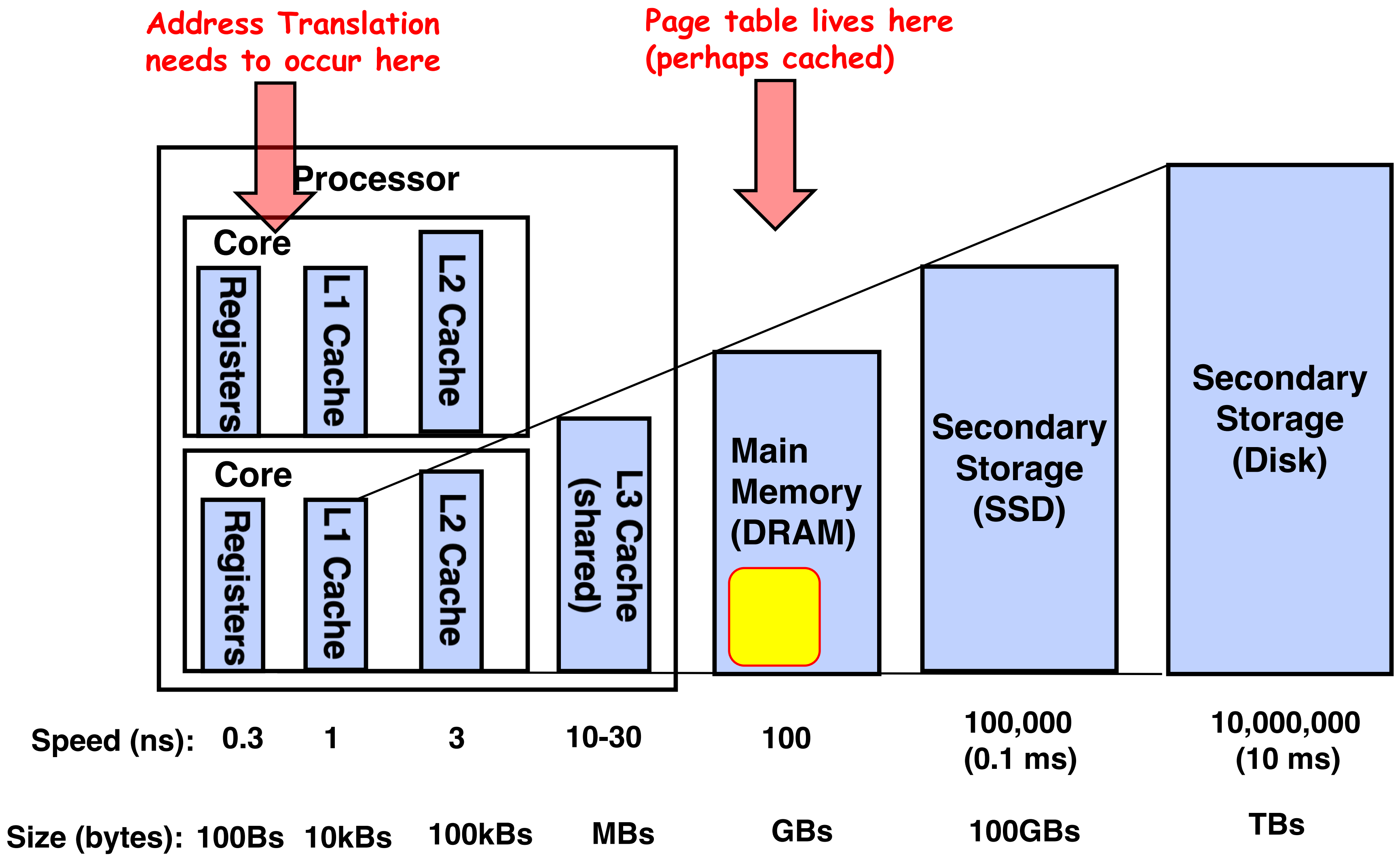
How is the Translation Accomplished?



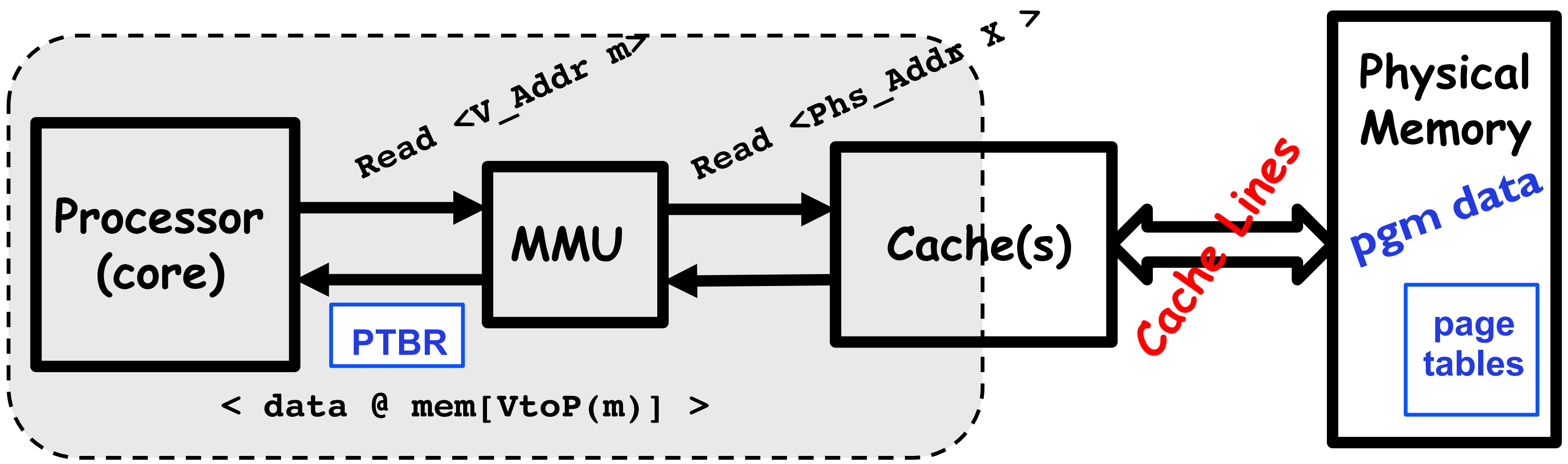
- What does the MMU need to do to translate an address?
- 1-level Page Table
  - Read PTE from memory, check valid, merge address
  - Set "accessed" bit in PTE, Set "dirty bit" on write
- 2-level Page Table
  - Read and check first level
  - Read, check, and update PTE
- N-level Page Table …
- MMU does *page table Tree Traversal* to translate each address
- How can we make this go REALLY fast?
  - Fraction of a processor cycle

Recall: Memory Hierarchy

- Large memories are slow, only small memory is fast

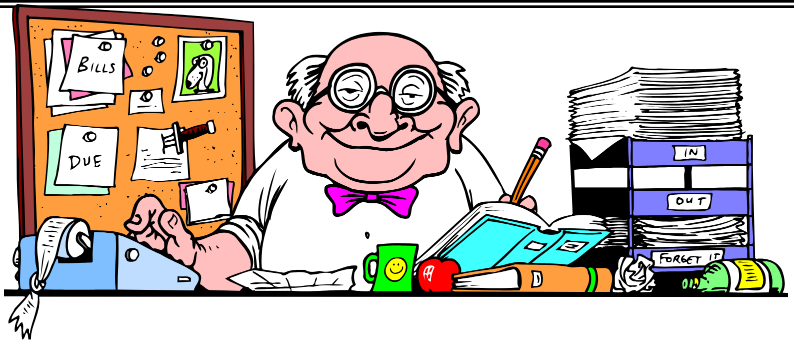


Where and What is the MMU ?



- The processor requests READ Virt-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

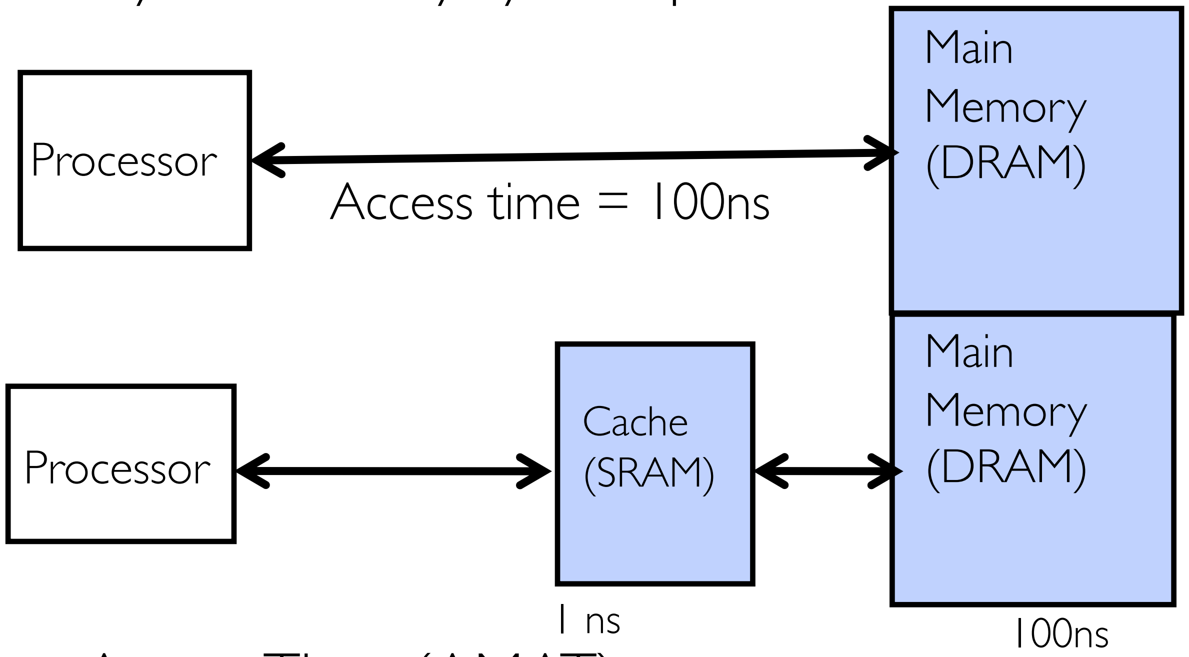
Recall: CS61c Caching Concept



- 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…
- Good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Key measure: Average Access time =
  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

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

- Caching is the key to memory system performance



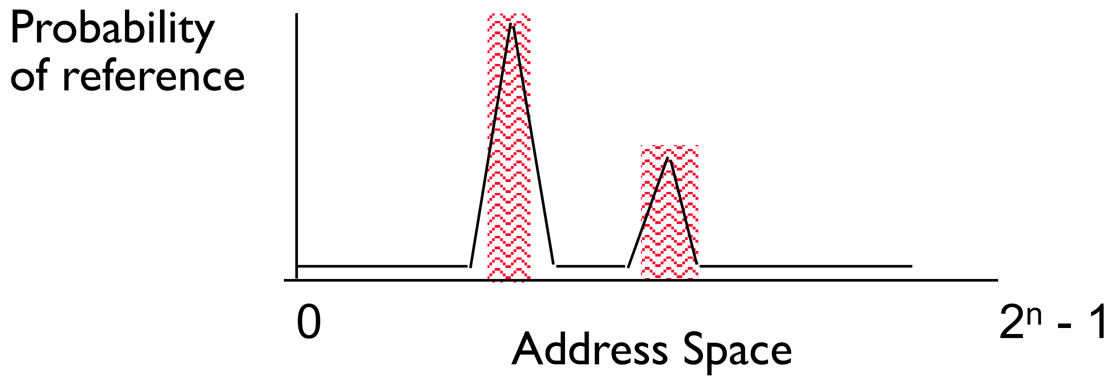
Average Memory Access Time (AMAT)
= (Hit Rate x HitTime) + (Miss Rate x MissTime)
Where HitRate + MissRate = 1

HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101)=11.1 ns

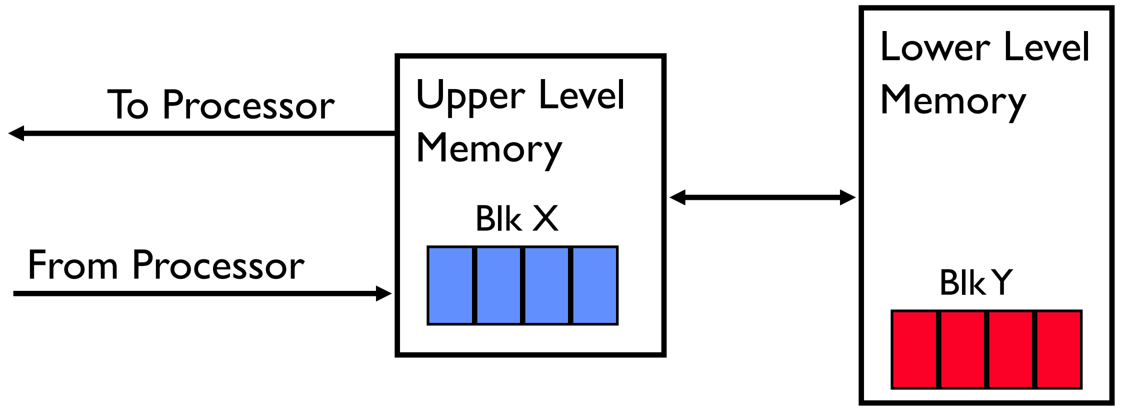
HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101)=2.01 ns

MissTime_{L1} includes $\text{HitTime}_{L1}+\text{MissPenalty}_{L1} \equiv \text{HitTime}_{L1} + \text{AMAT}_{L2}$

Recall: Why Does Caching Help? Locality!

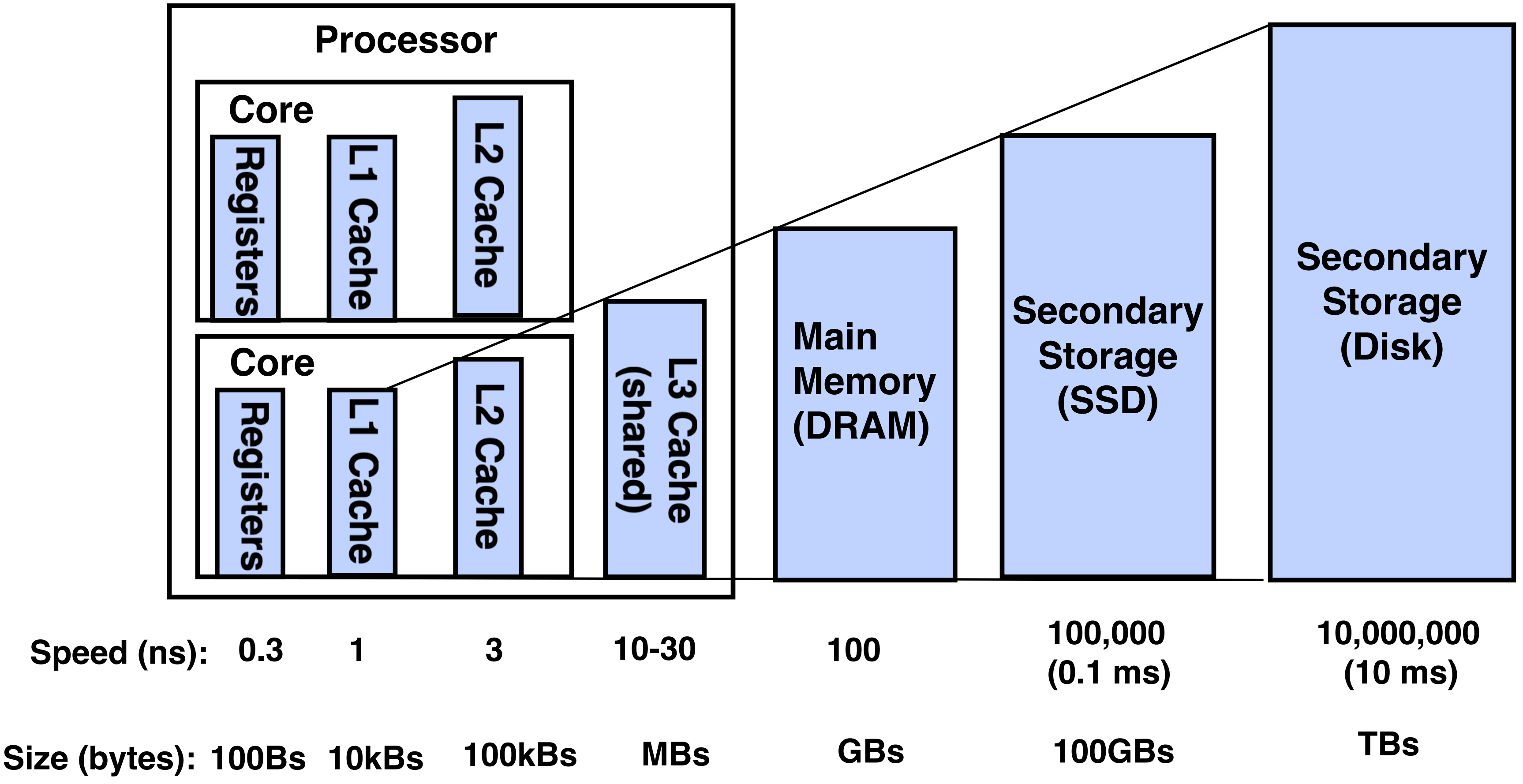


- Temporal Locality (Locality in Time):
  - – Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - – Move contiguous blocks to the upper levels



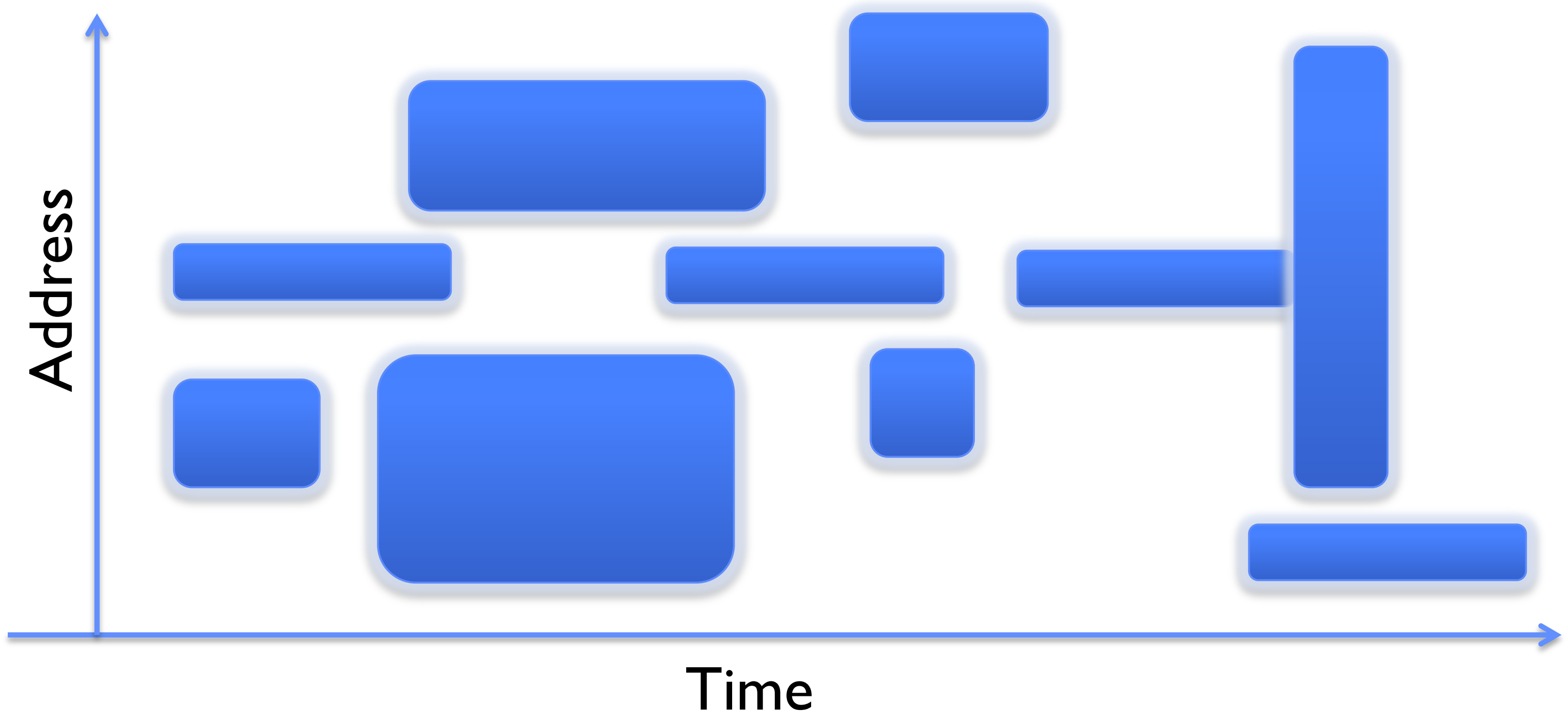
Recall: Memory Hierarchy

- 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



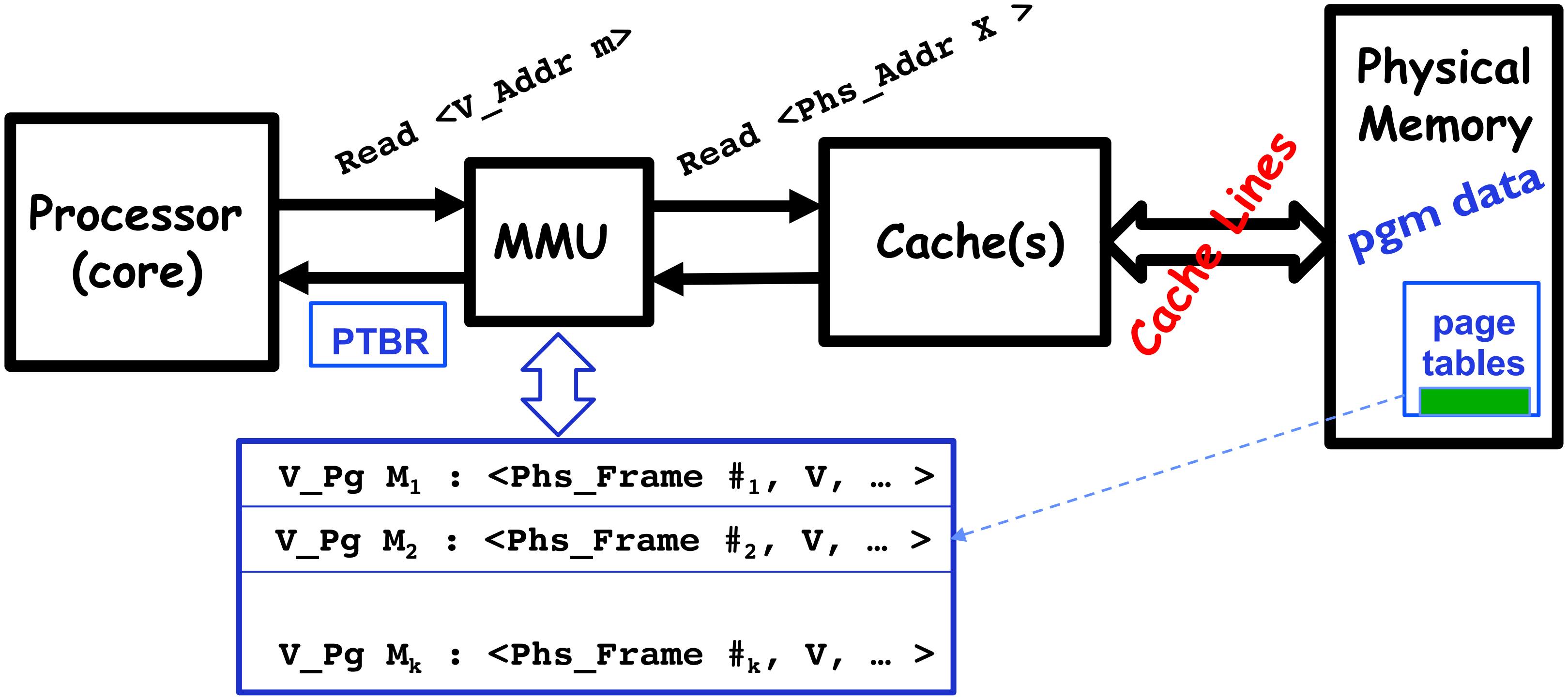
Working Set Model

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



How do we make Address Translation Fast?

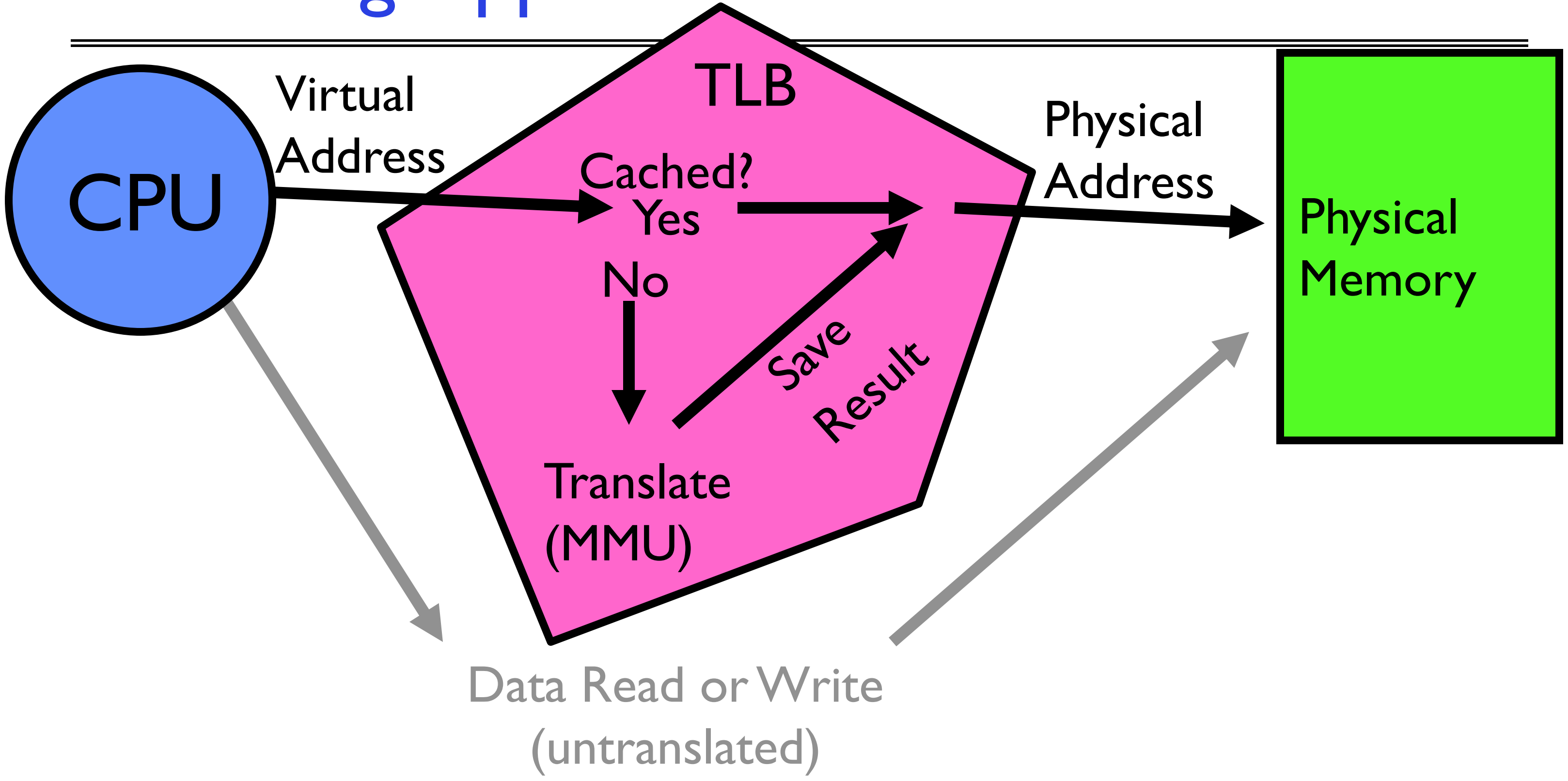
- Cache results of recent translations !
  - Different from a traditional cache
  - Cache Page Table Entries using Virtual Page # as the key



Translation Look-Aside Buffer

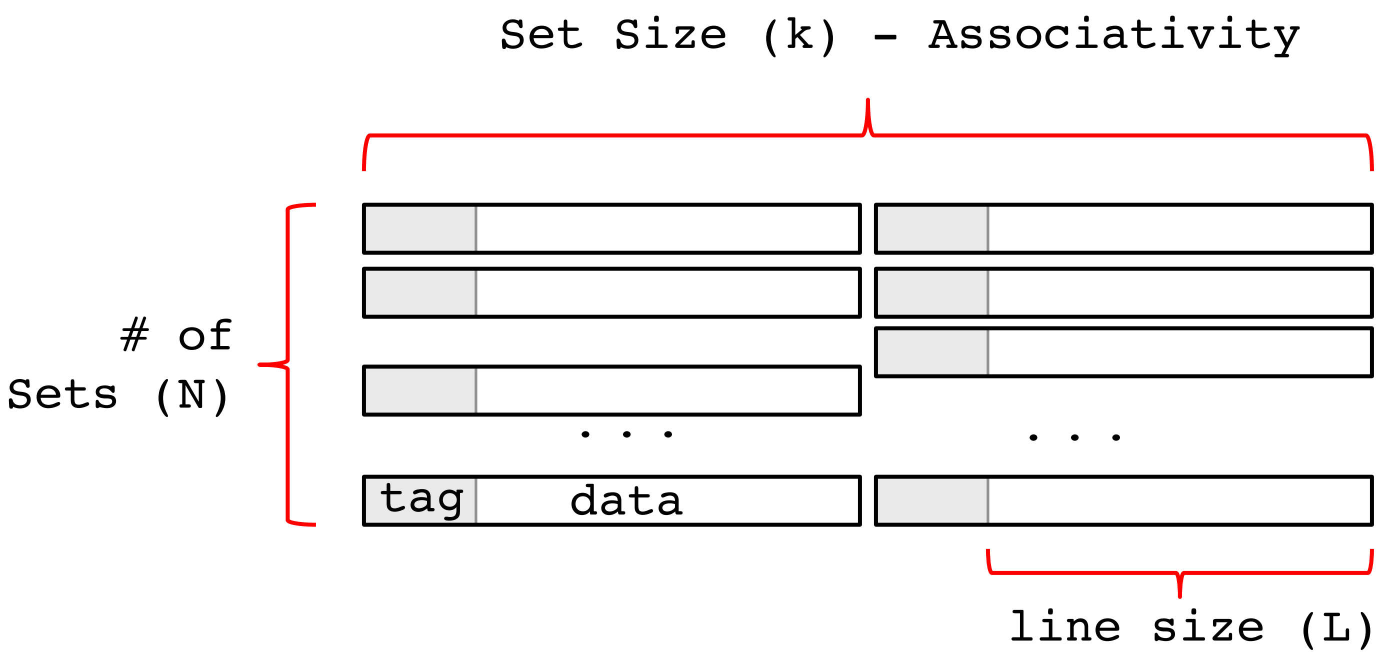
- Record recent Virtual Page # to Physical Frame # translation
- If present, have the physical address without reading any of the page tables !!!
  - Even if the translation involved multiple levels
  - Caches the end-to-end result
- Was invented by Sir Maurice Wilkes – *prior to caches*
- People realized "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 the same page (since accesses sequential, loops, function)
  - Stack accesses have definite locality of reference
  - Data accesses ?? …

What kind of Cache for TLB?



- Remember all those cache design parameters and trade-offs?
- Amount of Data = N * L * K
- Tag is portion of address that identifies line (w/o line offset)
- Write Policy (write-thru, write-back), Eviction Policy (LRU, …)

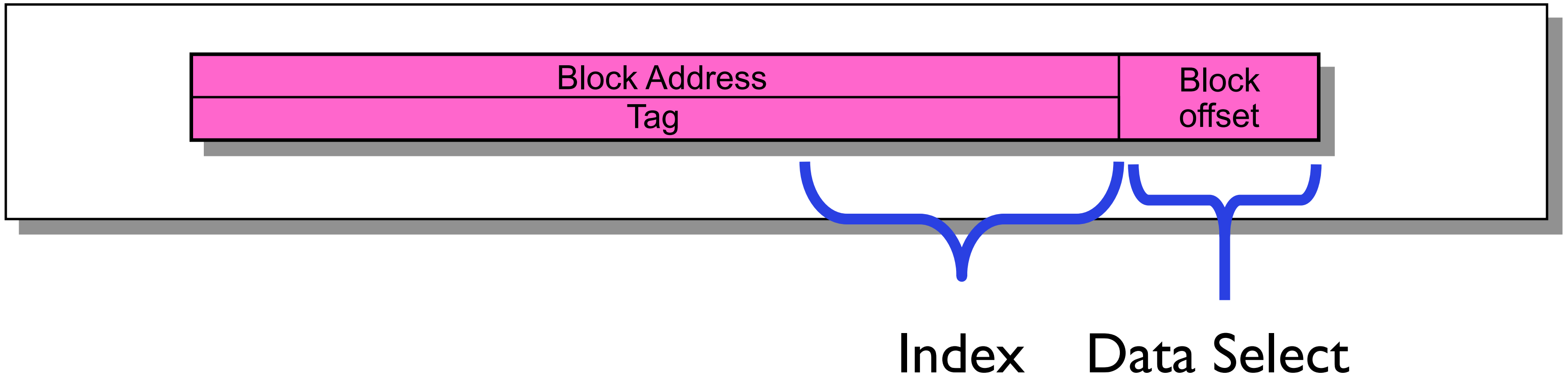
How might organization of TLB differ from that of a conventional instruction or data cache?

- Let's do some review …

4 C's: Summary of 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
- 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

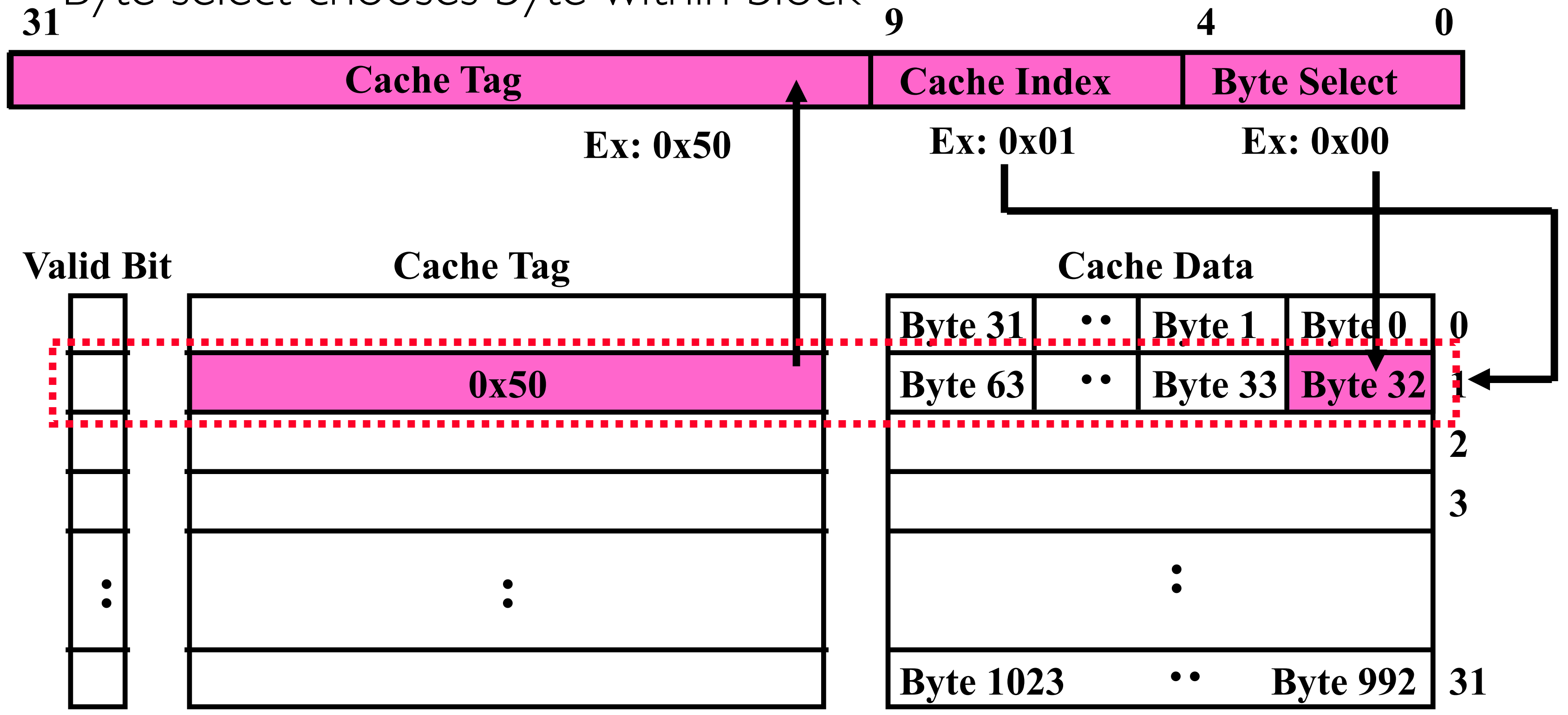
How is a Block found in a Cache?



- Block is minimum quantum of caching
  – Data select field used to select data within block
  – Many caching applications don't have data select field
- Tag used to identify actual the block (what address?)
  – If no candidates match, then declare cache miss
- Index Used to Lookup Candidates in Cache
  – Index identifies the set of possibilities (check tag)

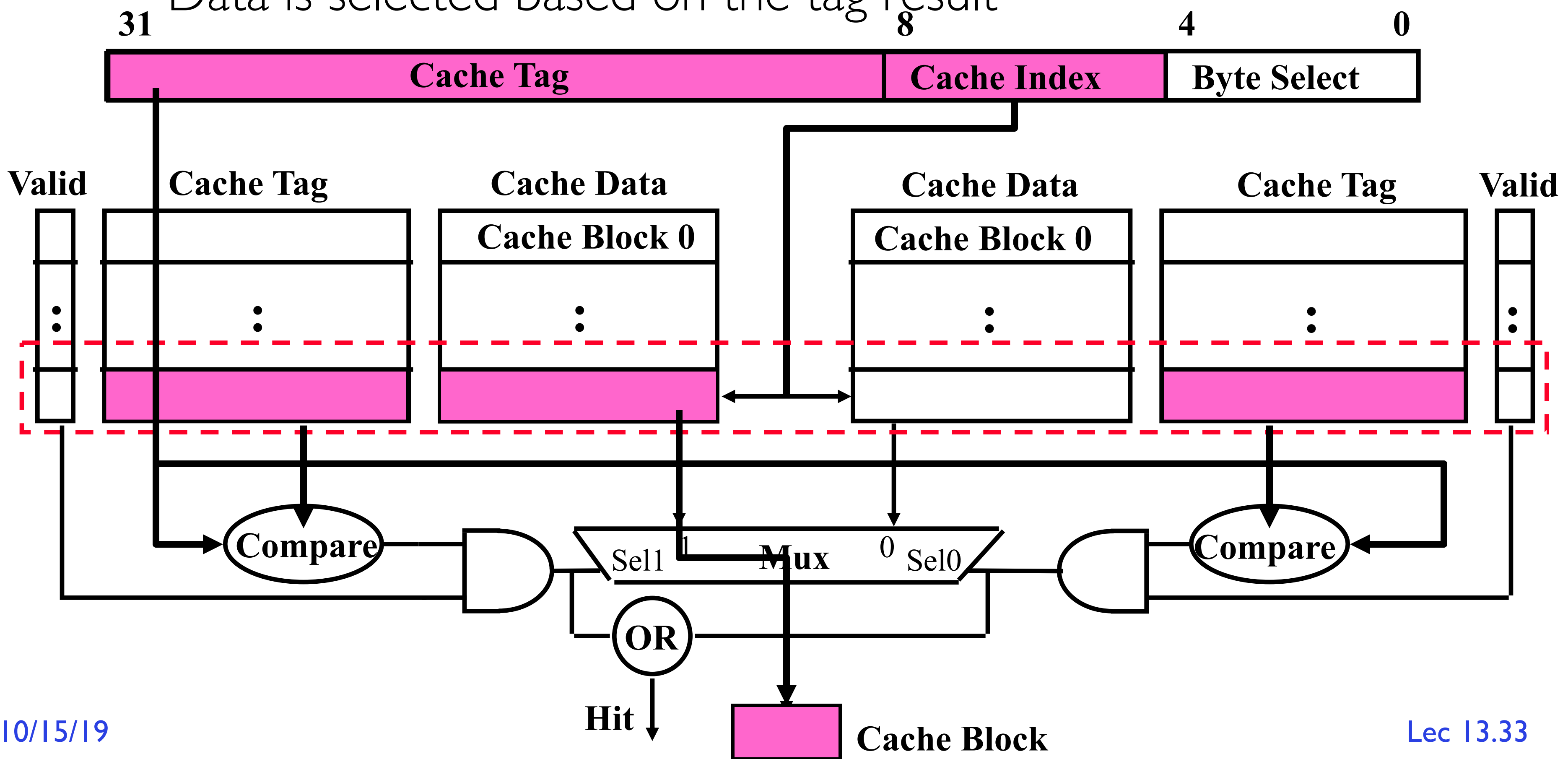
Review: Direct Mapped Cache

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



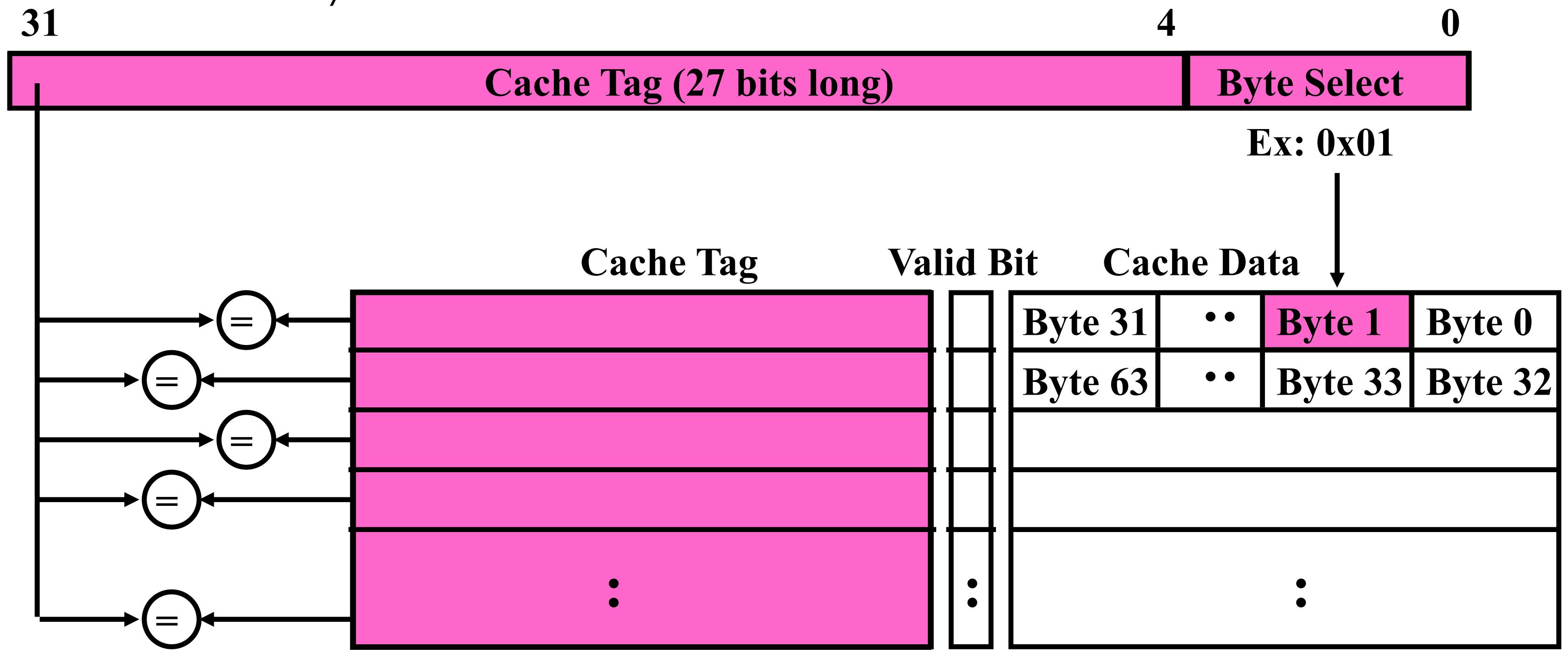
Review: Set Associative Cache

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



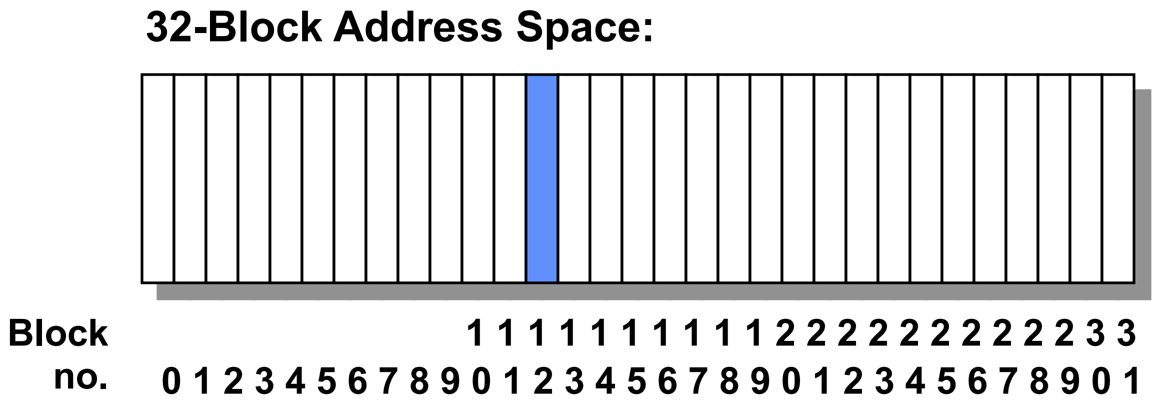
Review: Fully Associative Cache

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



Where does a Block Get Placed in a Cache?

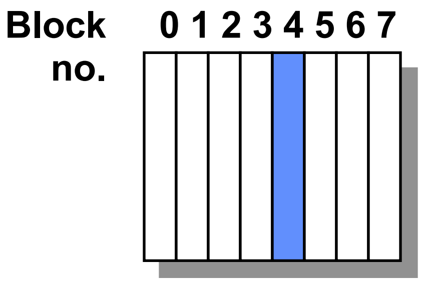
- Example: Block 12 placed in 8 block cache

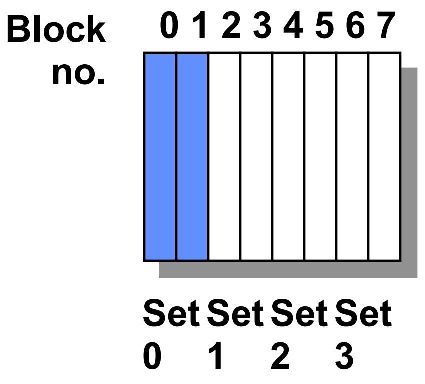


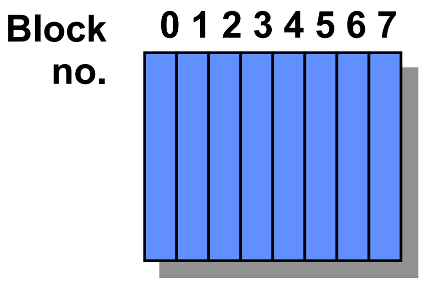
Direct mapped: block 12 can go only into block 4 (12 mod 8)

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?

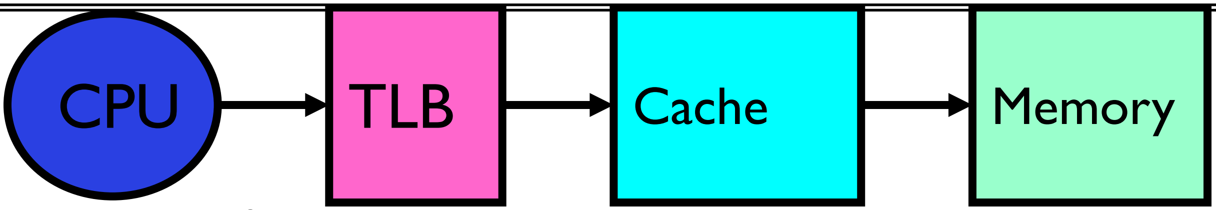
- Write through: The information is written to both the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?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

Questions about caches ?

- How does operating system behavior affect cache performance?
- Switching threads?
- Switching contexts?
- Cache design? What addresses are used?
- What does our understanding of caches tell us about TLB organization?

Break

What TLB Organization Makes Sense?



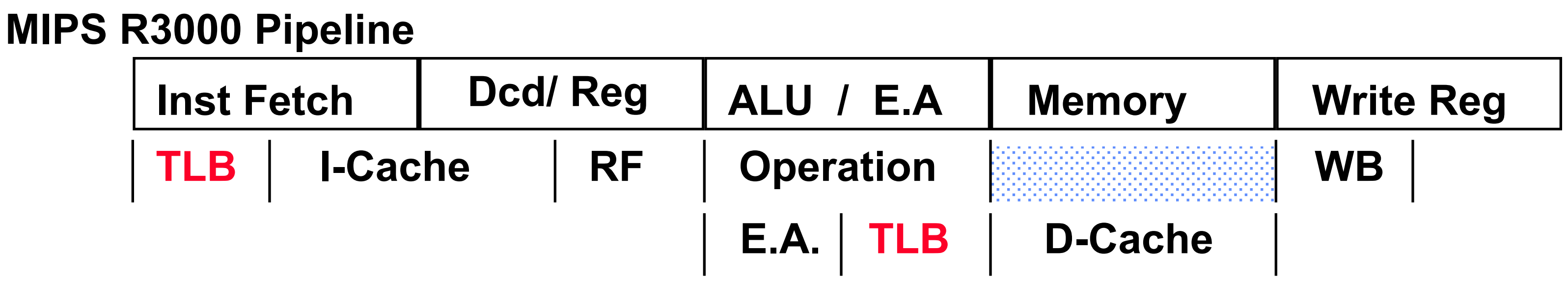
- Needs to be really fast
  - Critical path of memory access
    - » In simplest view: before the cache
    - » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely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

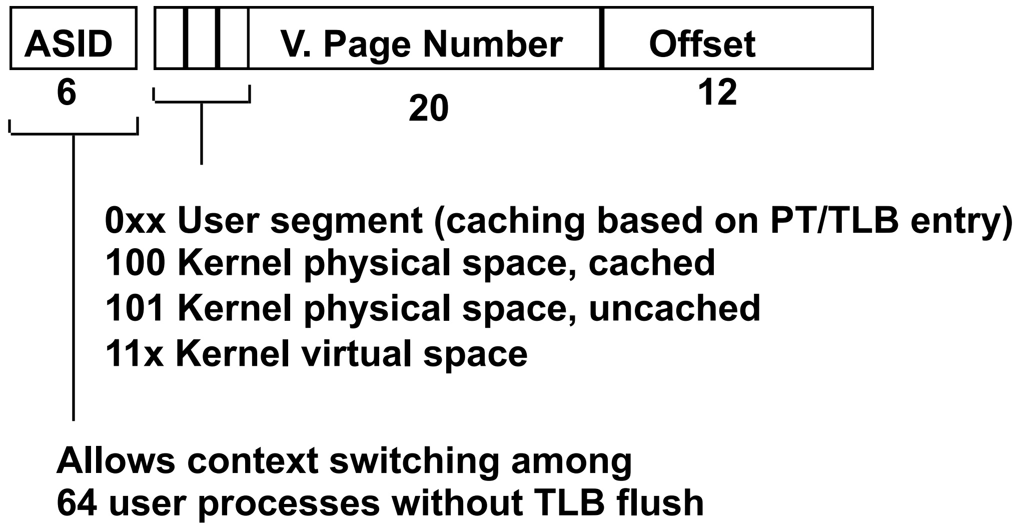
Example: R3000 pipeline includes TLB "stages"



TLB

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

Virtual Address Space


ASID
V. Page Number
Offset
6
20
12
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

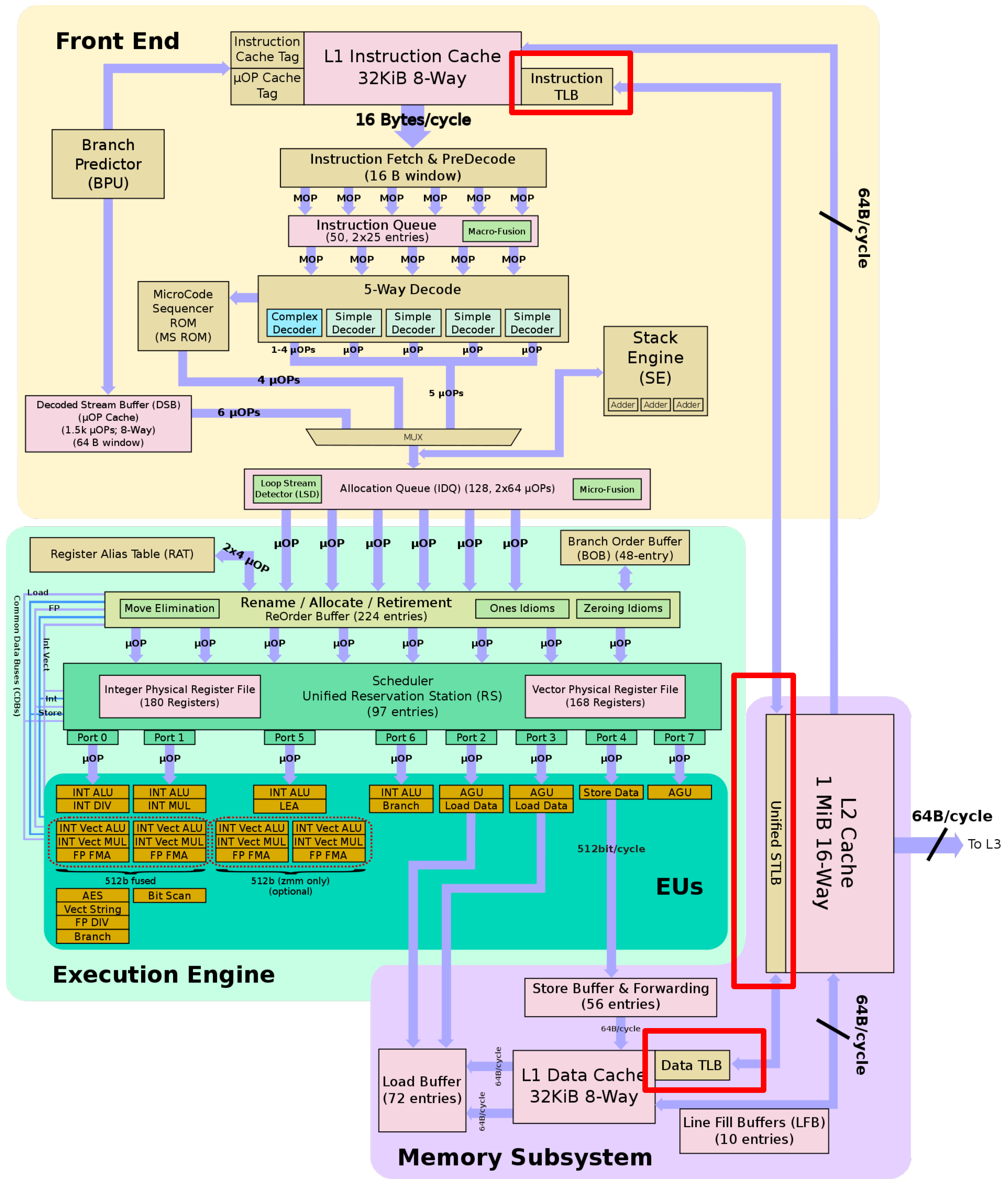
Example: Pentium-M TLBs (2003)

- Four different TLBs
  - Instruction TLB for 4K pages
    - » 128 entries, 4-way set associative
  - Instruction TLB for large pages
    - » 2 entries, fully associative
  - Data TLB for 4K pages
    - » 128 entries, 4-way set associative
  - Data TLB for large pages
    - » 8 entries, 4-way set associative
- All TLBs use LRU replacement policy
- Why different TLBs for instruction, data, and page sizes?

Intel Nahelem (2008)

- L1 DTLB
  - 64 entries for 4 K pages and
  - 32 entries for 2/4 M pages,
- L1 ITLB
  - 128 entries for 4 K pages using 4-way associativity and
  - 14 fully associative entries for 2/4 MiB pages
- unified 512-entry L2 TLB for 4 KiB pages, 4-way associative.

Current Intel x86 (Skylake, Cascade Lake)


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP MOP MOP MOP MOP MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MOP MOP MOP MOP MOP
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Simple Decoder
1-4 µOPs µOP µOP µOP µOP
4 µOPs
5 µOPs
Stack Engine (SE)
Adder Adder Adder
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
2x4 µOP
µOP µOP µOP µOP µOP µOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Common Data Buses (CDBs)
Int Vect
Int
Store
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
µOP µOP µOP µOP µOP µOP µOP µOP
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0 Port 1 Port 5 Port 6 Port 2 Port 3 Port 4 Port 7
µOP µOP µOP µOP µOP µOP µOP µOP
INT ALU INT DIV
INT ALU INT MUL
INT ALU LEA
INT ALU Branch
AGU Load Data
AGU Load Data
Store Data
AGU
INT Vect ALU INT Vect MUL FP FMA
INT Vect ALU INT Vect MUL FP FMA
INT Vect ALU INT Vect MUL FP FMA
INT Vect ALU INT Vect MUL FP FMA
512b fused
512b (zmm only) (optional)
AES Vect String FP DIV Branch
Bit Scan
512bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 1 MiB 16-Way
64B/cycle
To L3
64B/cycle
Store Buffer & Forwarding (56 entries)
64B/cycle
64B/cycle
64B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

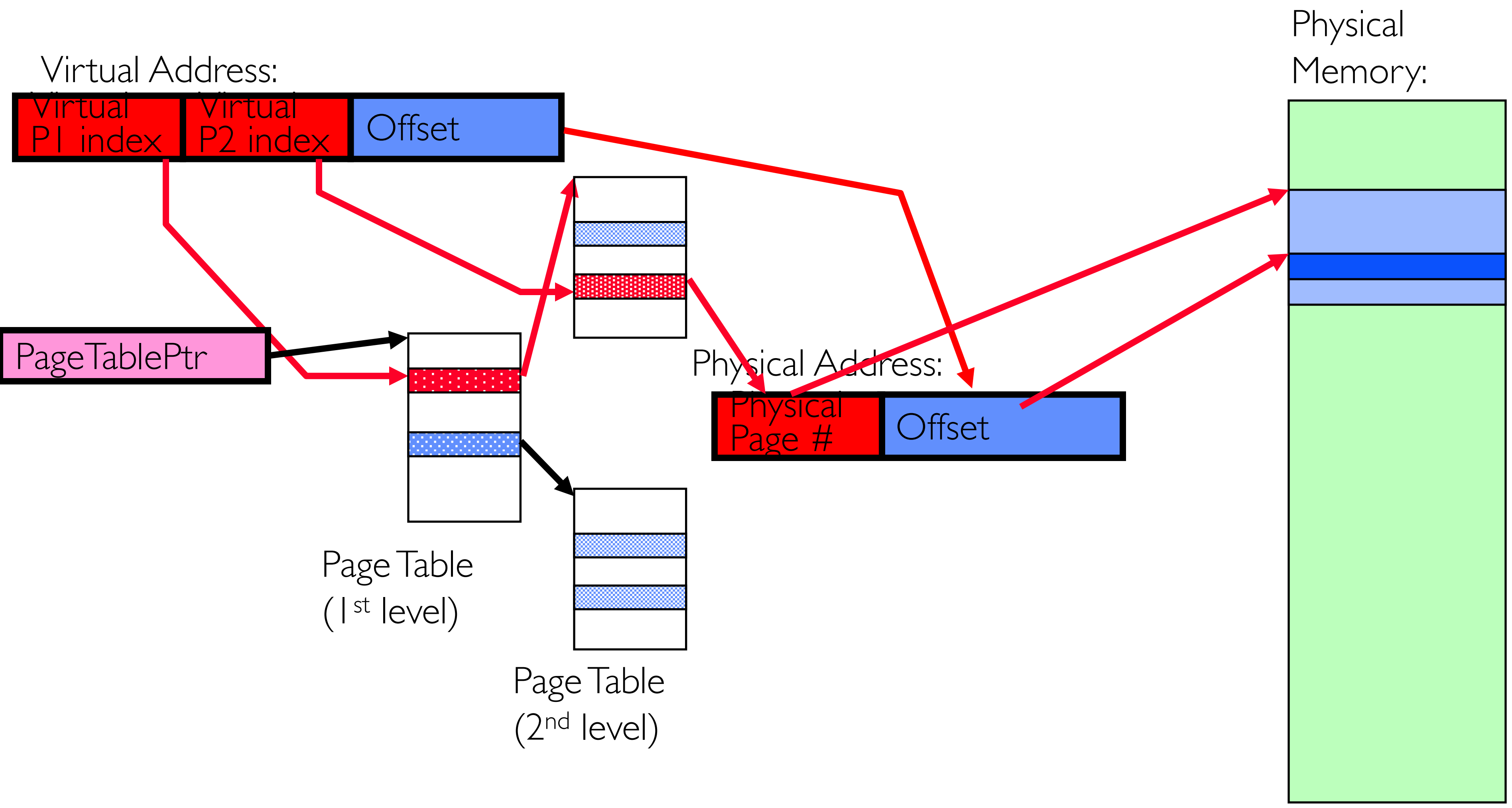
Current Example: Memory Hierarchy

- 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:

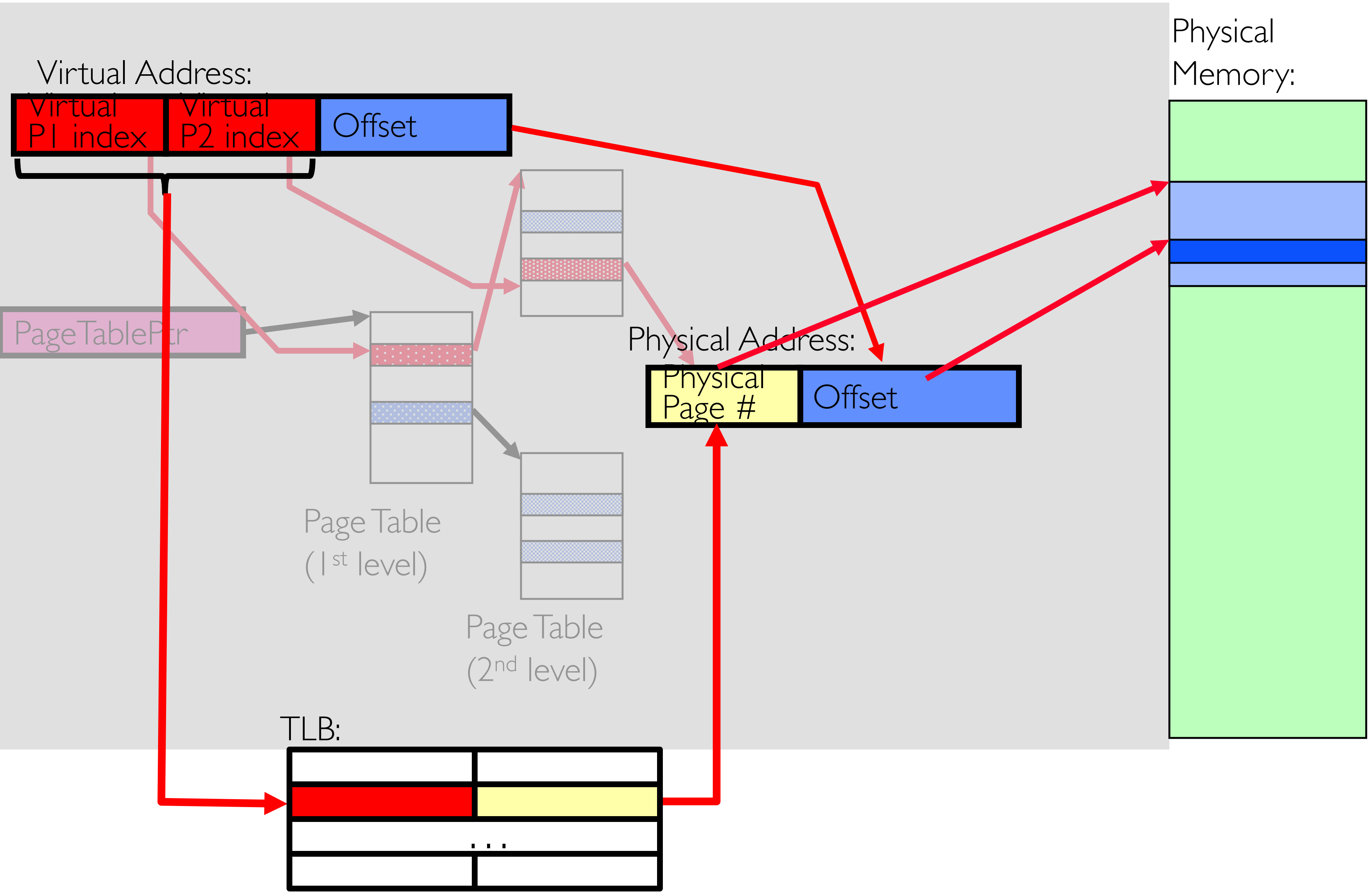
What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  – Address Space just changed, so TLB entries no longer valid!
- Options?
  – Invalidate 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"

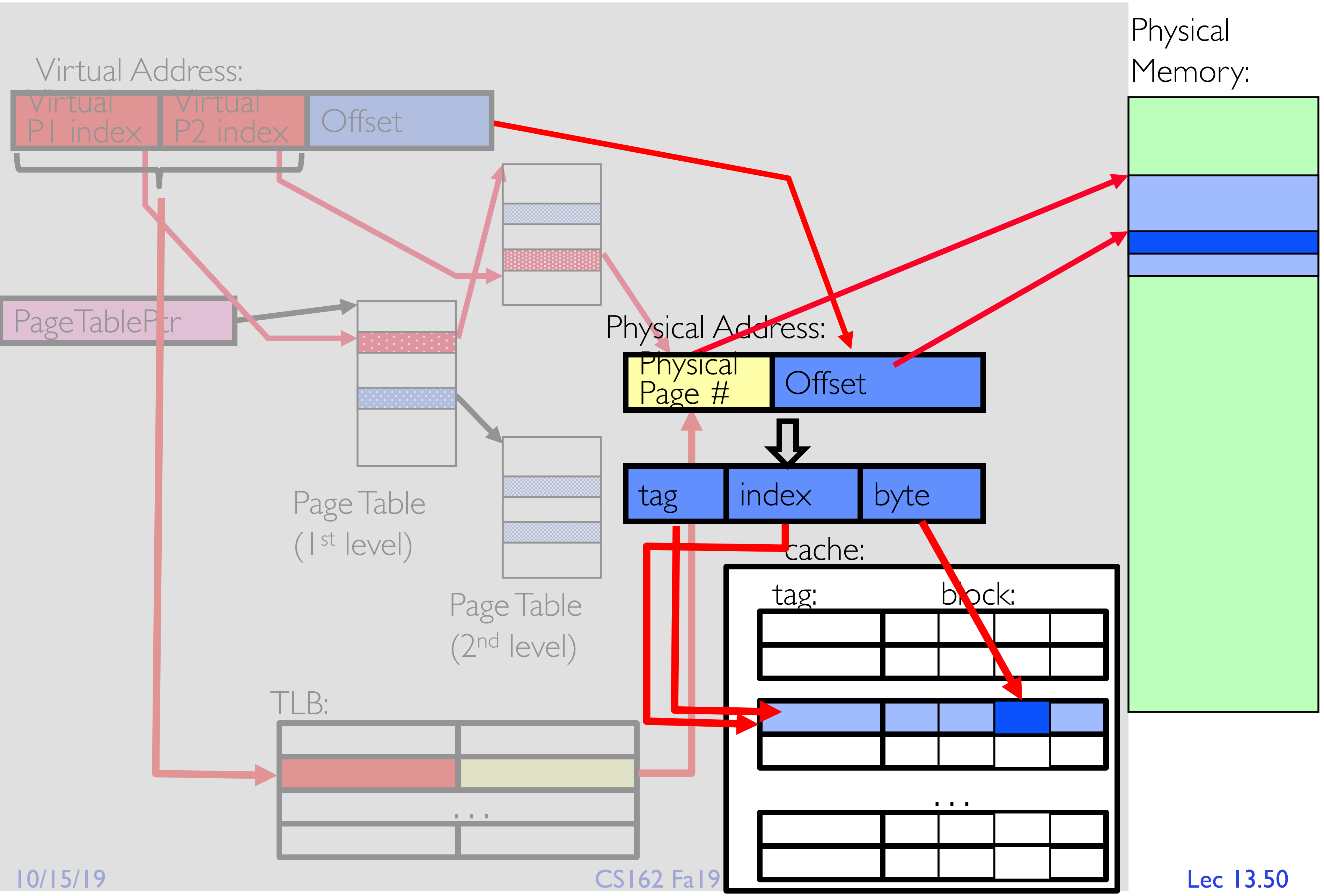
Putting Everything Together: Address Translation



Putting Everything Together: TLB

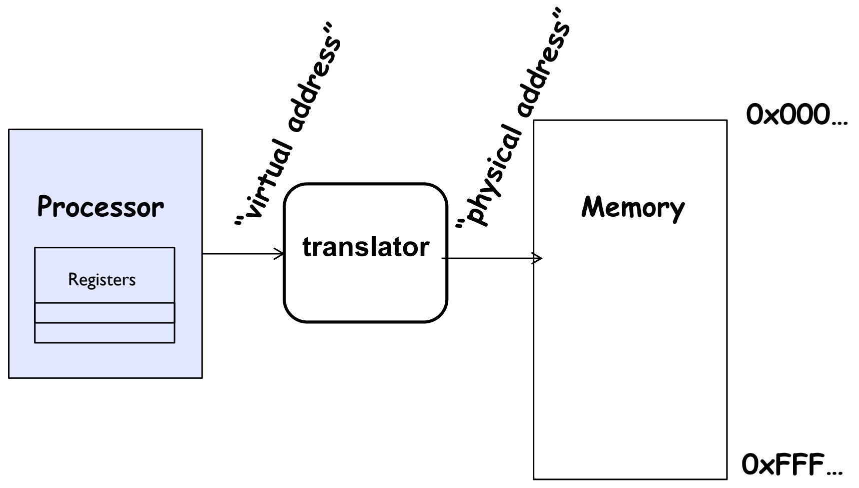


Putting Everything Together: Cache



Two Critical Issues in Address Translation

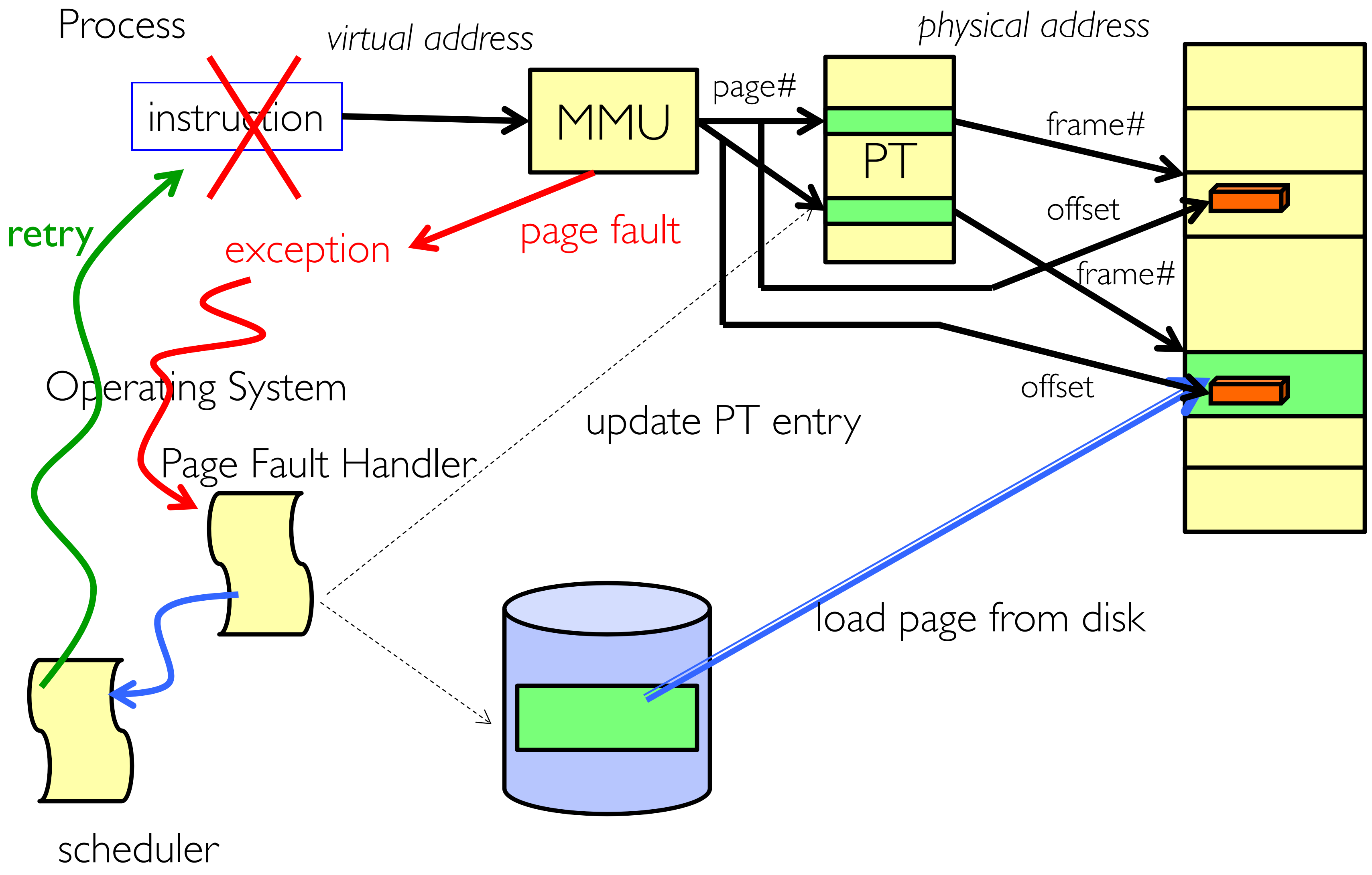
- What to do if the translation fails? - a page fault
- How to translate addresses fast enough?
  - Every instruction fetch
  - Plus every load / store
  - EVERY MEMORY REFERENCE !
  - More than one translation for EVERY instruction



Page Fault

- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Priv. 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 instruction
- 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

Next Up: What happens when …

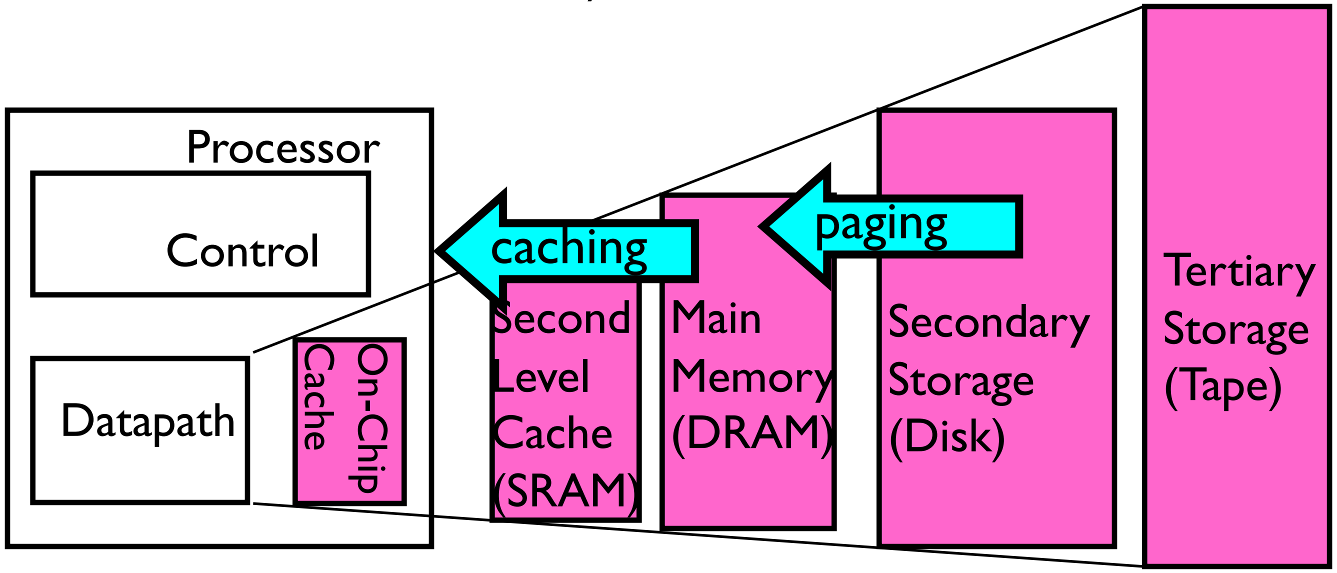


Inversion of the Hardware / Software Boundary

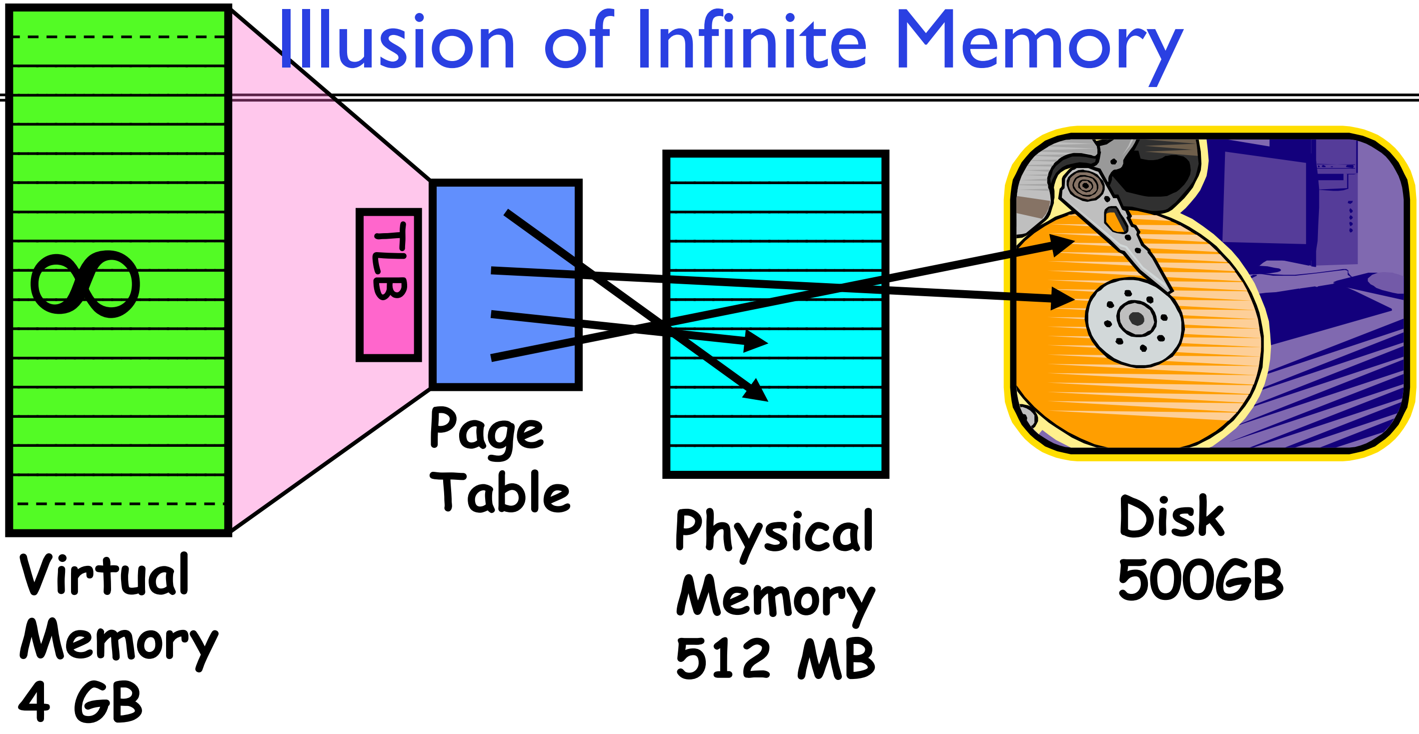
- In order for an instruction to complete …
- It requires the intervention of operating system software
- Receive the page fault, remedy the situation
  - Load the page, create the page, copy-on-write
  - Update the PTE entry so the translation will succeed
- Restart (or resume) the instruction
  - This is one of the huge simplifications in RISC instructions sets
  - Can be very complex when instruction modify state (x86)

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



- 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
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - » Performance issue, not correctness issue

Demand Paging as a form of Caching

- Must ask:
  - What is block size?
    - » 1 page
  - What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
    - » Fully associative: arbitrary virtual→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… (kinda 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!

Review: What is in a PTE?

- 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:
  - 2-level page tabler (10, 10, 12-bit offset)
  - 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

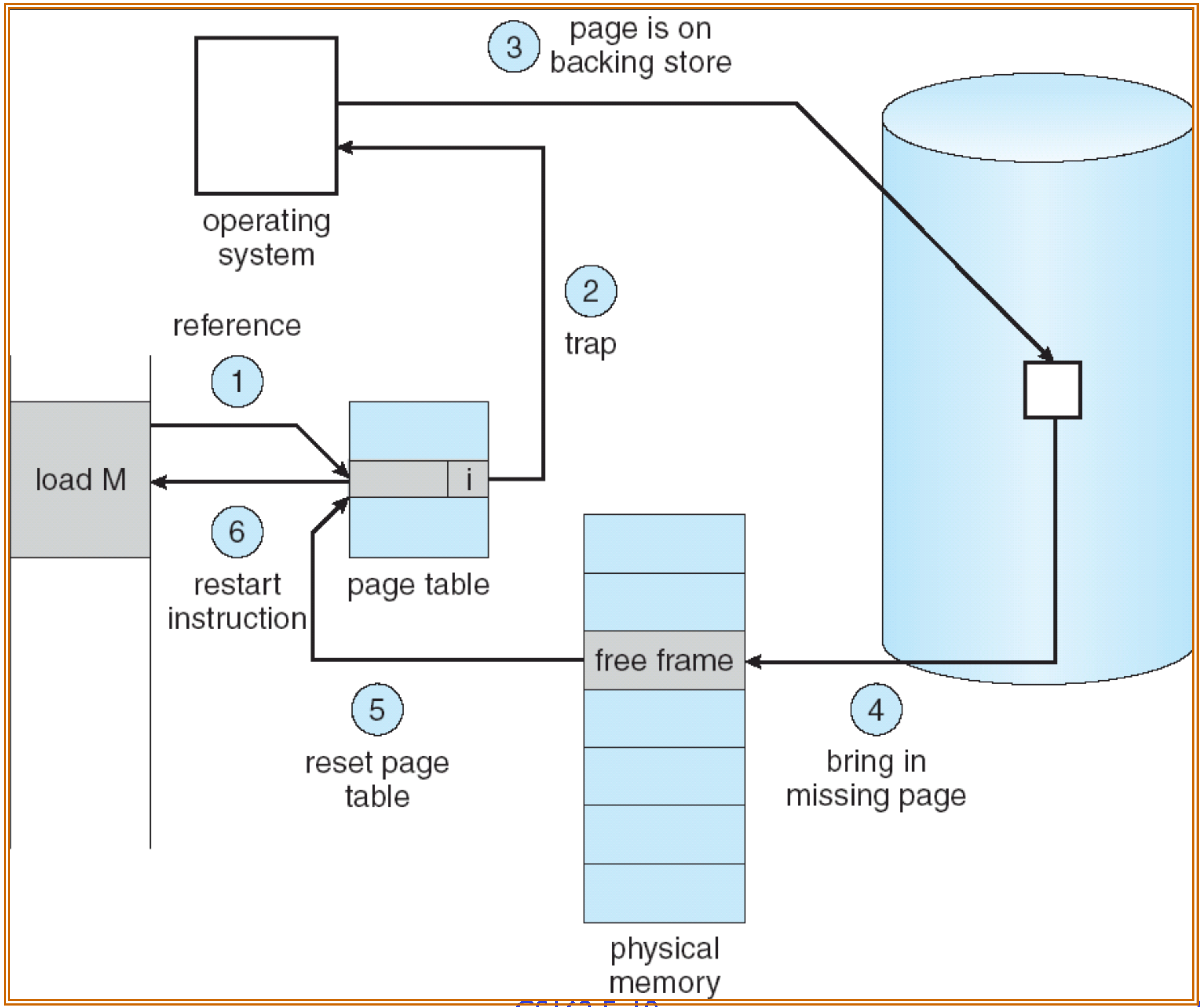
P:	Present (same as "valid" bit in other architectures)
W:	Writeable
U:	User accessible
PWT:	Page write transparent: external cache write-through
PCD:	Page cache disabled (page cannot be cached)
A:	Accessed: page has been accessed recently
D:	Dirty (PTE only): page has been modified recently
L:	L=1⇒4MB 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 (or other) 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"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread is continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Cache

Summary: Steps in Handling a Page Fault



Where are places that caching arises in OSes?

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

- Which pages to keep in memory?
  - All-important "Policy" aspect of virtual memory
  - Will spend a bit more time on this in upcoming lectures

Impact of caches on Operating Systems

- Indirect - dealing with cache effects
  - Maintaining the correctness of various caches
  - E.g., TLB consistency:
    - » With PT across context switches ?
    - » Across updates to the 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 interleaving of threads (small quantum) may degrade cache performance
    - » Increase average memory access time (AMAT) !!!
- Designing operating system data structures for cache performance

Summary (1/3)

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted Page Table
  - Use of hash-table to hold translation entries
  - Size of page table ~ size of physical memory rather than size of virtual memory

Summary (2/3)

- 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 (3/3)

- "Translation Lookaside Buffer" (TLB)
  - Small number of PTEs and optional process IDs (< 512)
  - 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
  - TLB is logically in front of cache (need to overlap with cache access)
- On Page Fault, OS can take actions to resolve the situation
  - Demand paging, automatic memory management
  - Make copy of existing page for process
  - On process start, don't have to load much of executable into memory
  - Rarely used code and data may never get paged in
- Need to handle the exception carefully