

CS162
Operating Systems and
Systems Programming
Lecture 17

Memory 4: TLBs and Demand Paging Policies

March 19th, 2026

Prof. John Kubiatowicz

<http://cs162.eecs.Berkeley.edu>

Recall 61C: Average Memory Access Time

- Used to compute access time probabilistically:

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

$$\text{Hit Rate}_{L1} + \text{Miss Rate}_{L1} = 1$$

Hit Time_{L1} = Time to get value from L1 cache.

Miss Time_{L1} = $\text{Hit Time}_{L1} + \text{Miss Penalty}_{L1}$

Miss Penalty_{L1} = AVG Time to get value from lower level (DRAM)

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

- What about more levels of hierarchy?

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

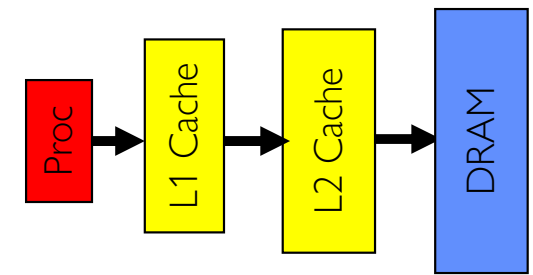
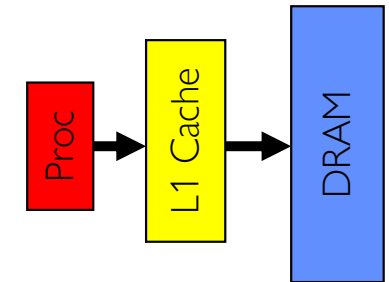
$$\begin{aligned} \text{Miss Penalty}_{L1} &= \text{AVG time to get value from lower level (L2)} \\ &= \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2} \end{aligned}$$

Miss Penalty_{L2} = Average Time to fetch from below L2 (DRAM)

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

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

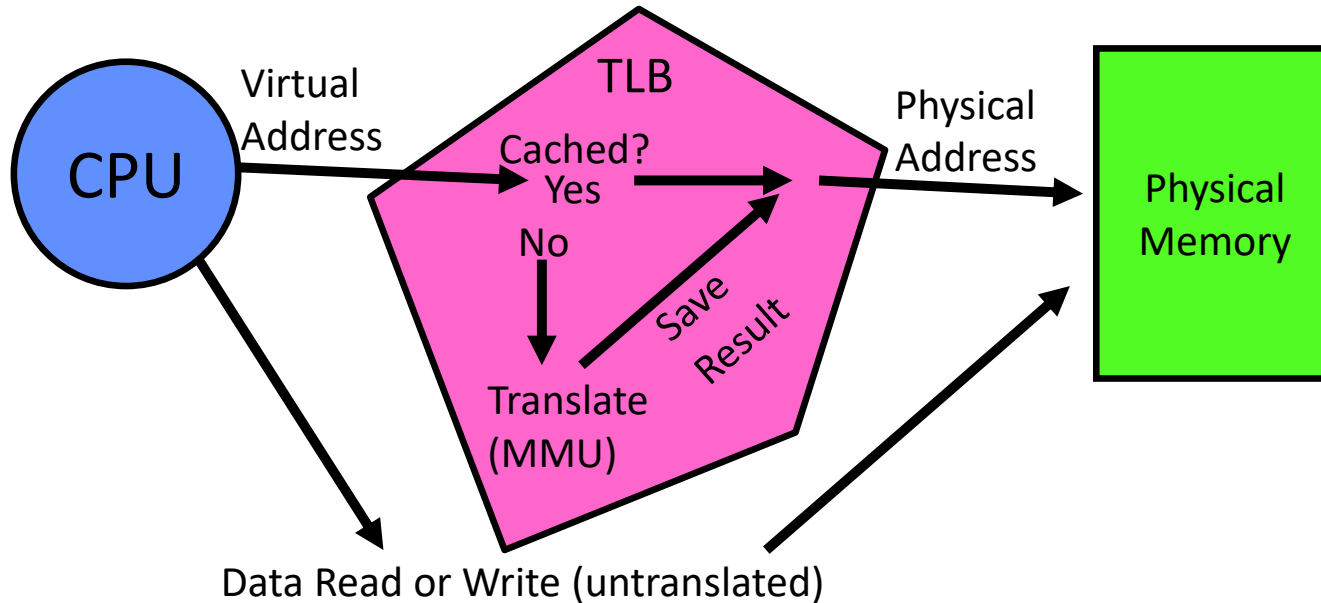
- And so on ... (can do this recursively for more levels!)



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

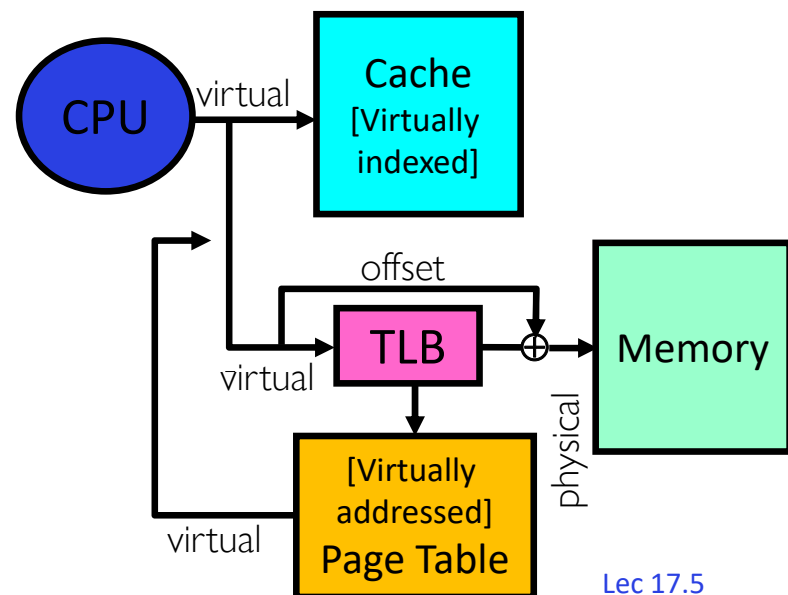
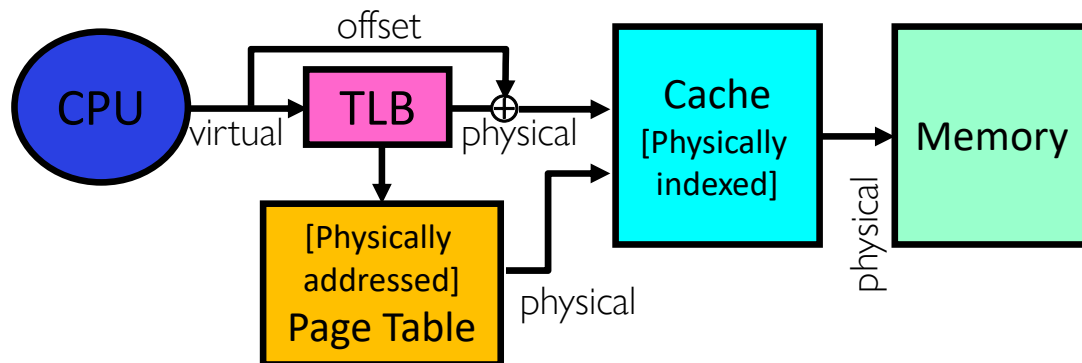
- **Physically-Indexed, Physically-Tagged**

- 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)

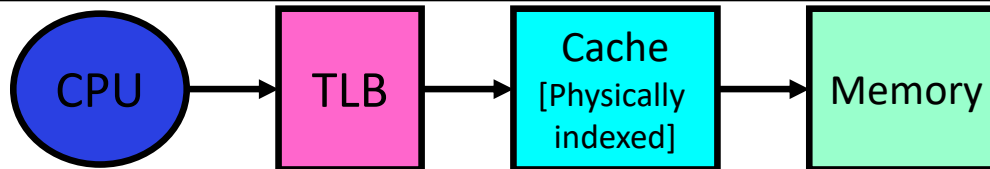
- **Virtually-Indexed, Virtually-Tagged or Physically-Tagged**

- 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!

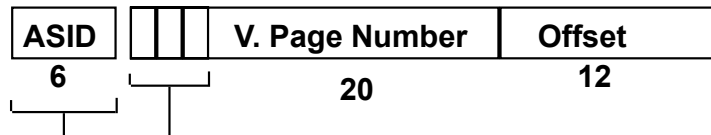
Example: MIPS R3000 Pipeline

Inst Fetch		Dcd/ Reg		ALU / E.A		Memory		Write Reg	
TLB	I-Cache	RF	Operation				WB		
			E.A.	TLB	D-Cache				

TLB

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

Virtual Address Space

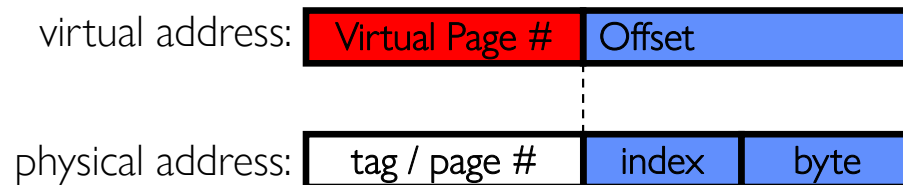
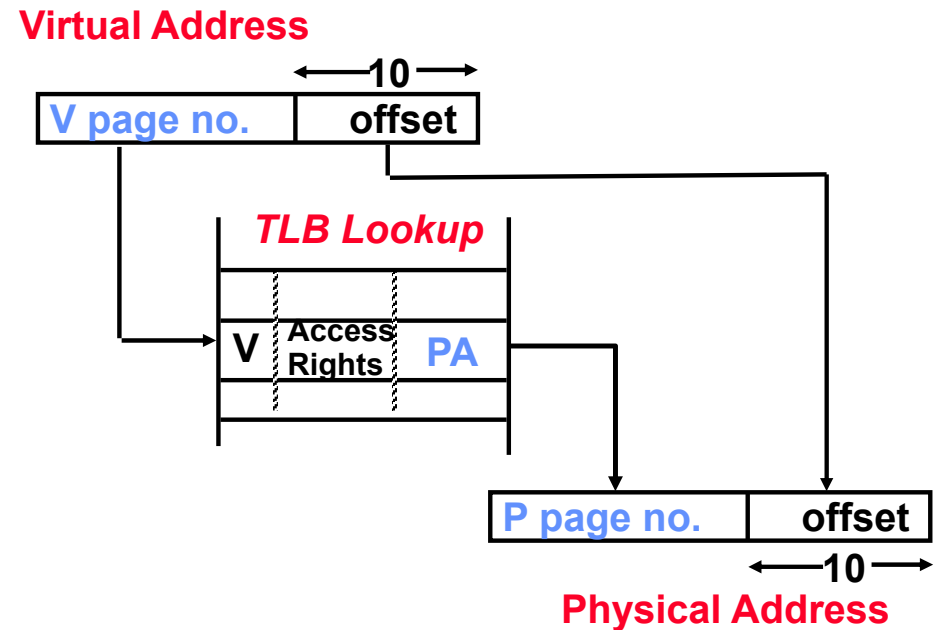


- 0xx User segment (caching based on PT/TLB entry)
- 100 Kernel physical space, cached
- 101 Kernel physical space, uncached
- 11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

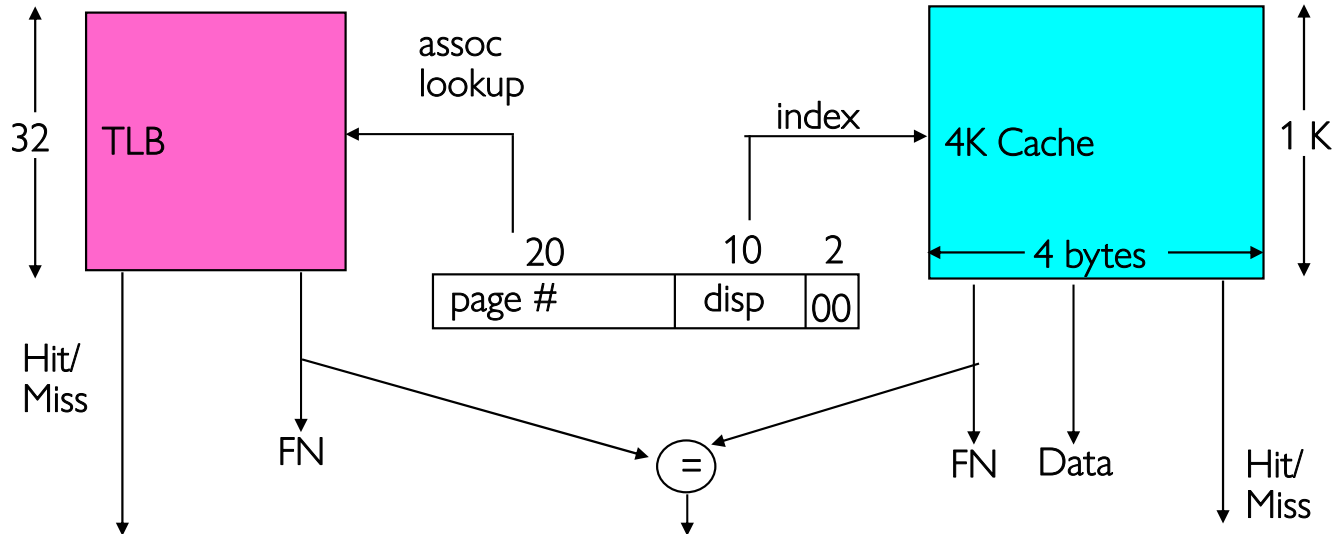
Further reducing translation time for physically-indexed caches

- 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:

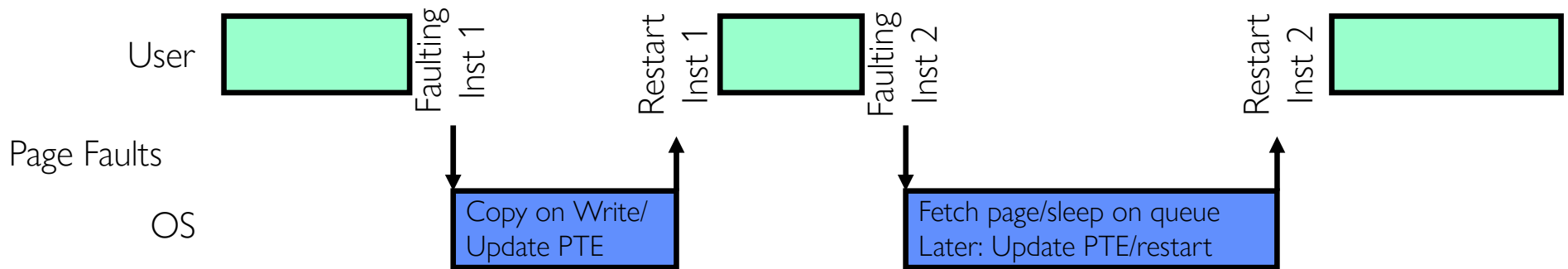


- What if cache size is increased to 8KB?
 - Overlap not complete
 - Need to do something else. See CS152/252
- As discussed earlier, Virtual Caches would make this faster
 - Tags in cache are virtual addresses
 - Translation only happens on cache misses

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 writing 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?

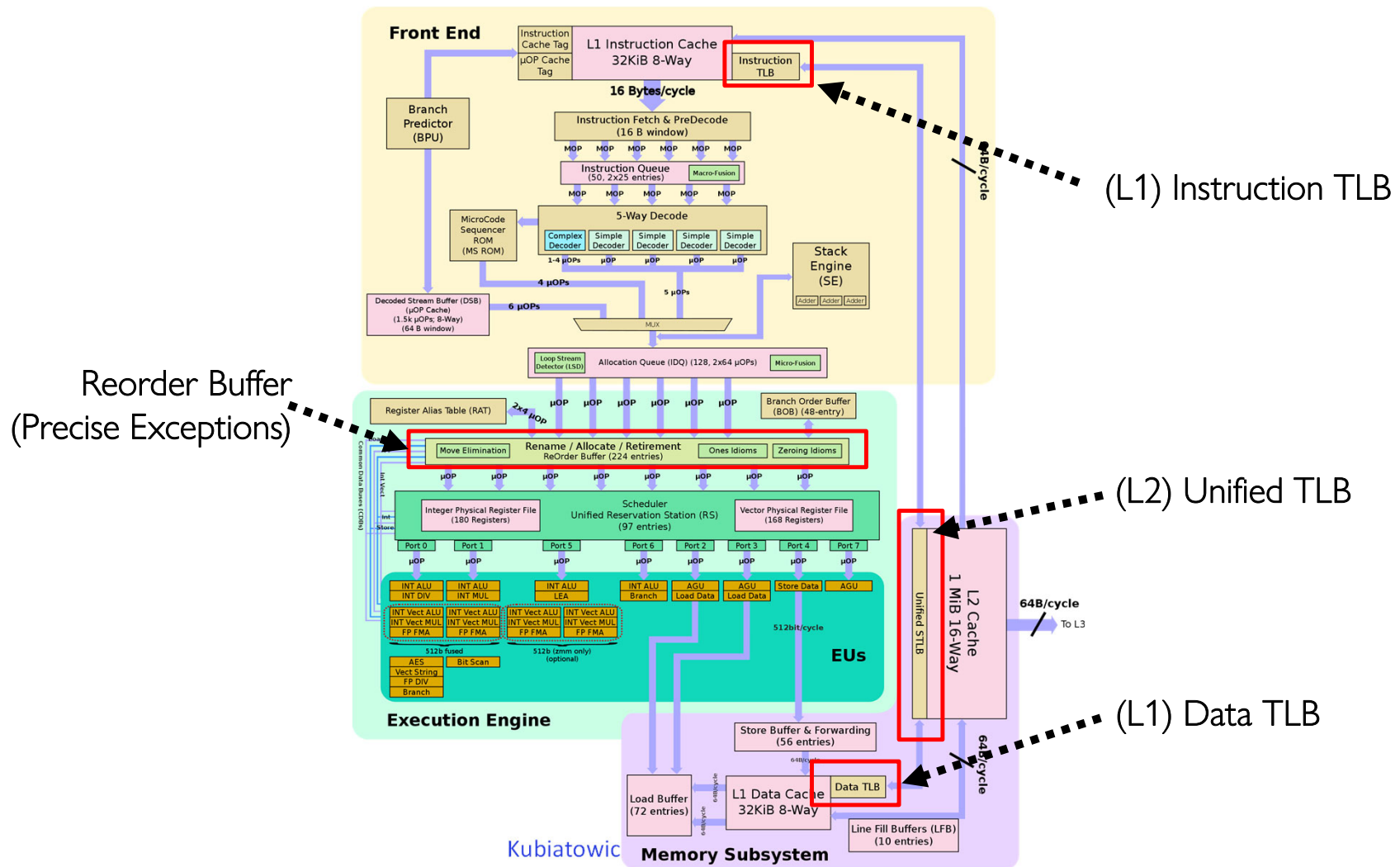
Precise Exceptions

- 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**

Administrivia

- Midterm 2 on 3/31
 - First Tuesday after Spring Break
- Project 2 in full swing
 - Stay on top of this one. Don't wait until last moment to get pieces together
- Homework 4 Starting on Saturday
- Make sure to fill out survey!
 - We really want to hear how you think we are doing
 - Also, will get a chance to suggest topics for the special topics lecture

Recent Intel x86 (Skylake, Cascade Lake)



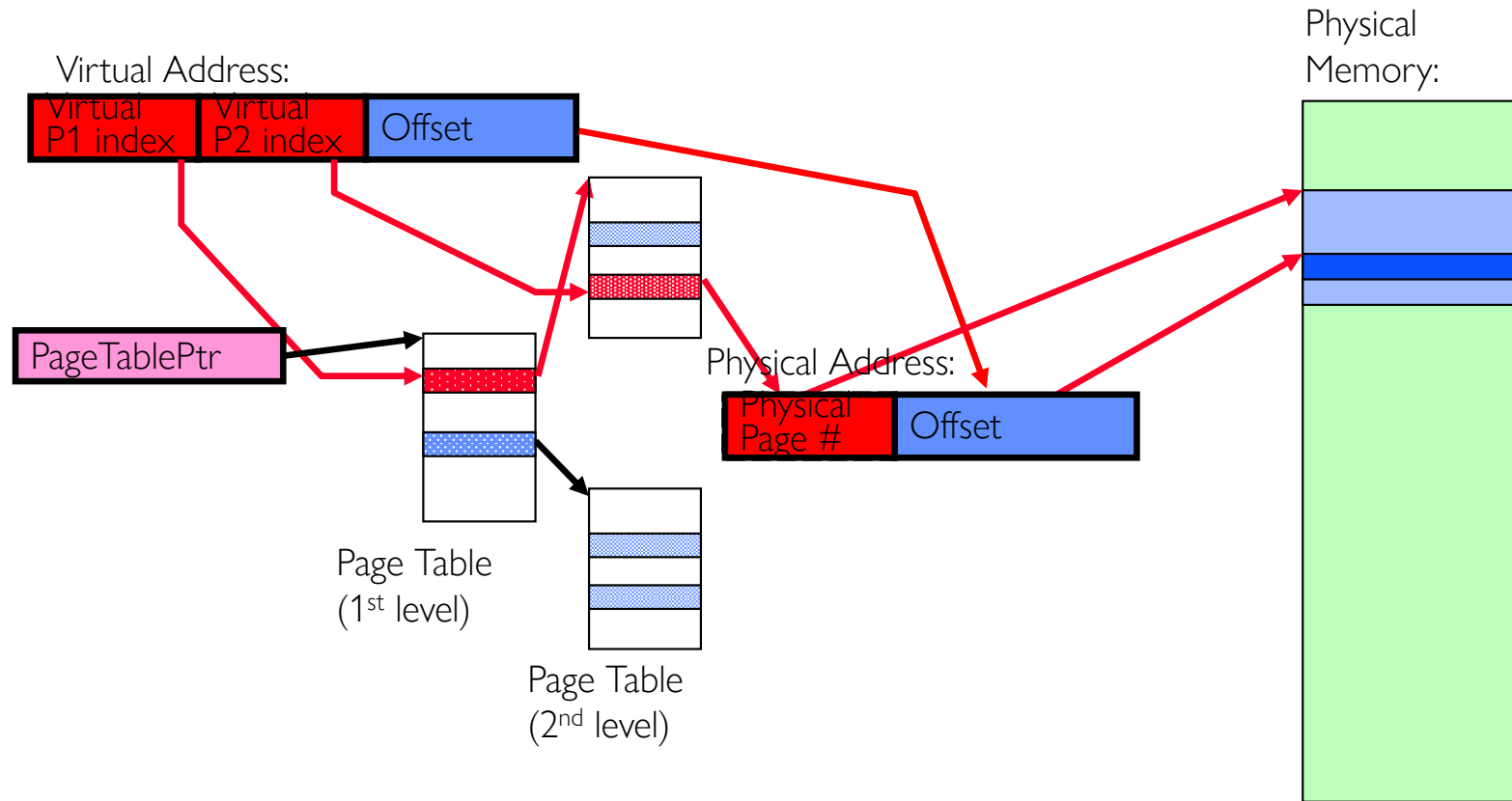
Recent Example: Memory Hierarchy

- 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:

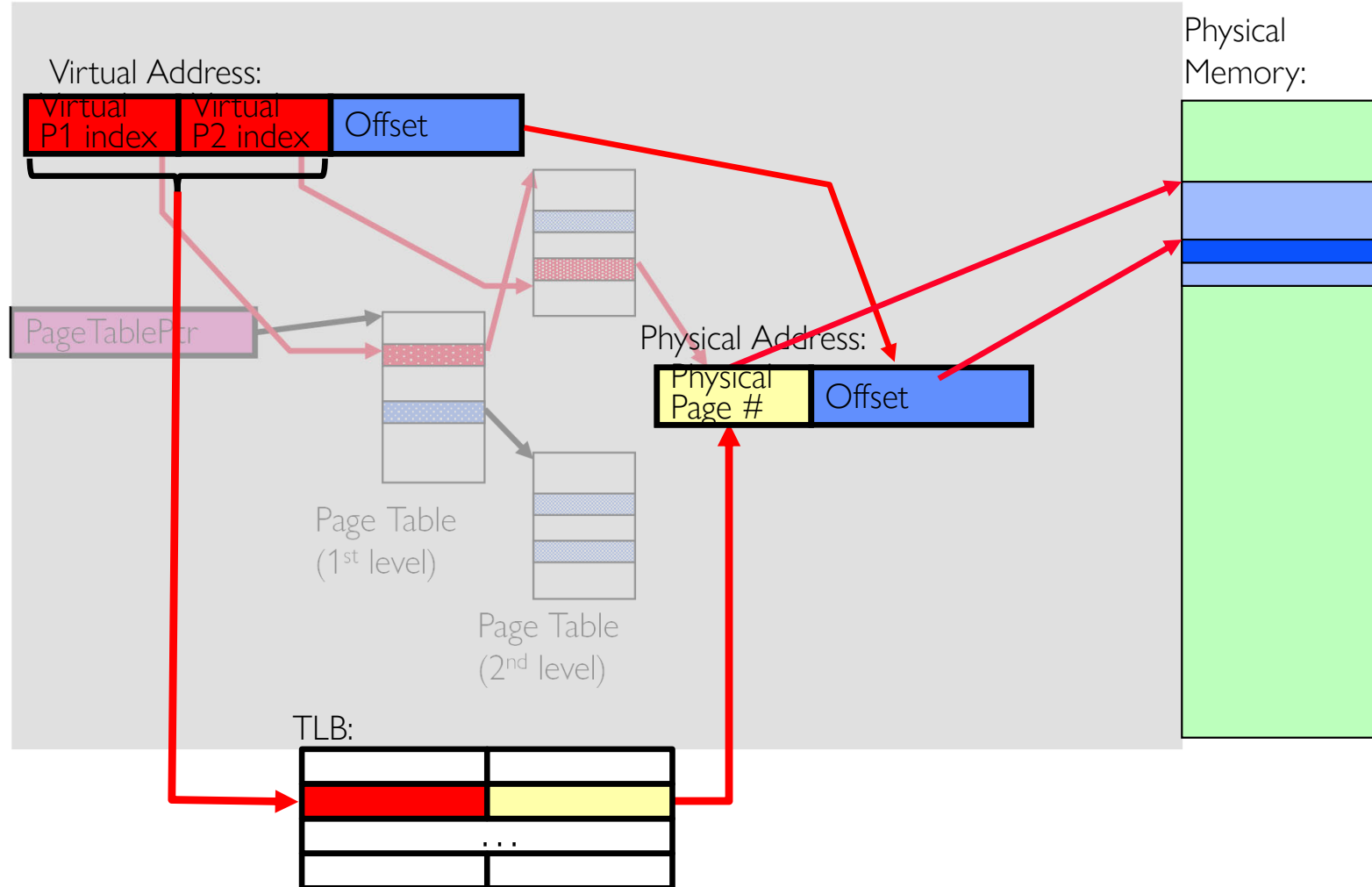
What happens on a Context Switch?

- 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..

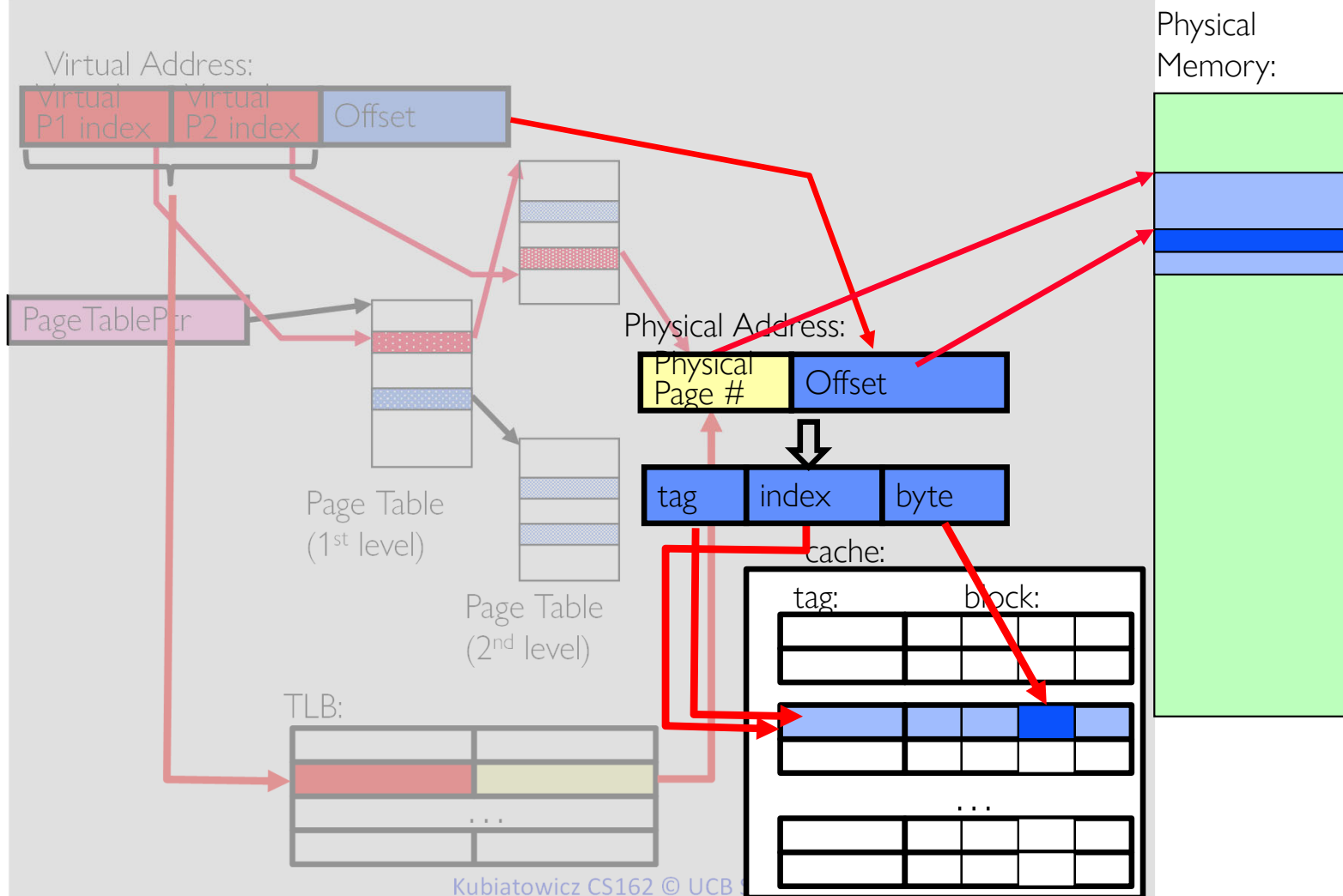
Putting Everything Together: Address Translation



Putting Everything Together: TLB



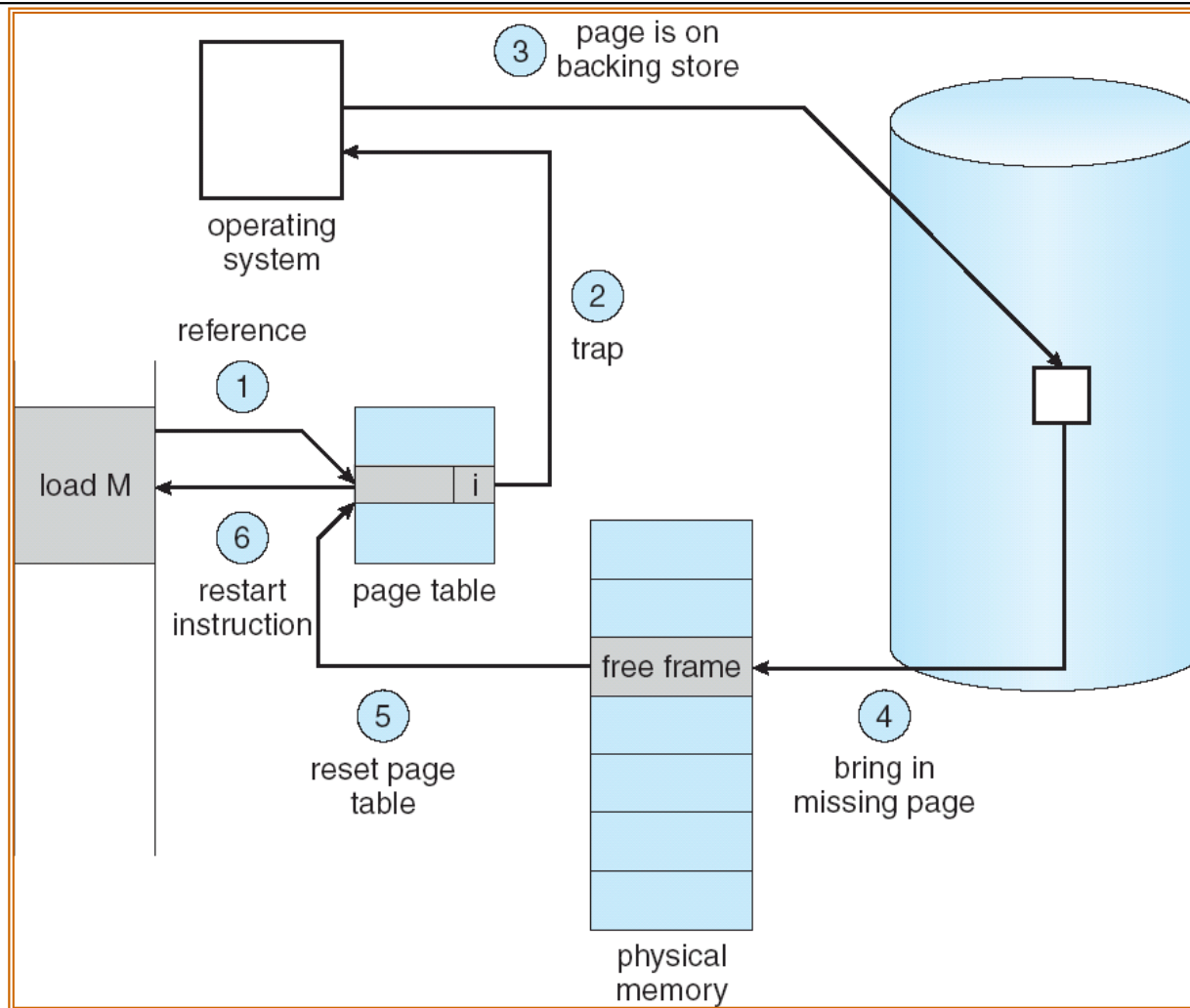
Putting Everything Together: Cache



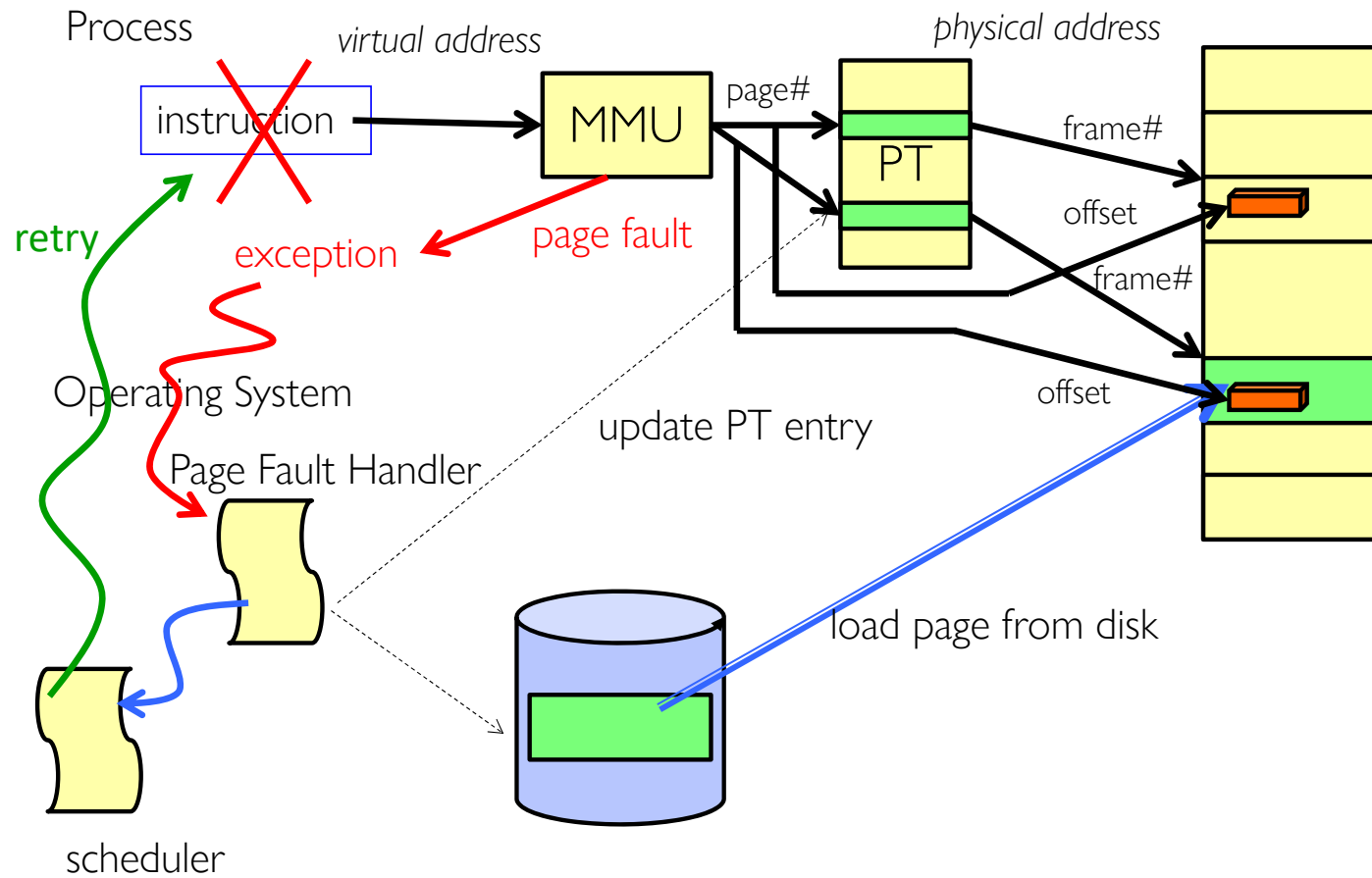
Page Fault Handling

- 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!

Steps in Handling a Page Fault

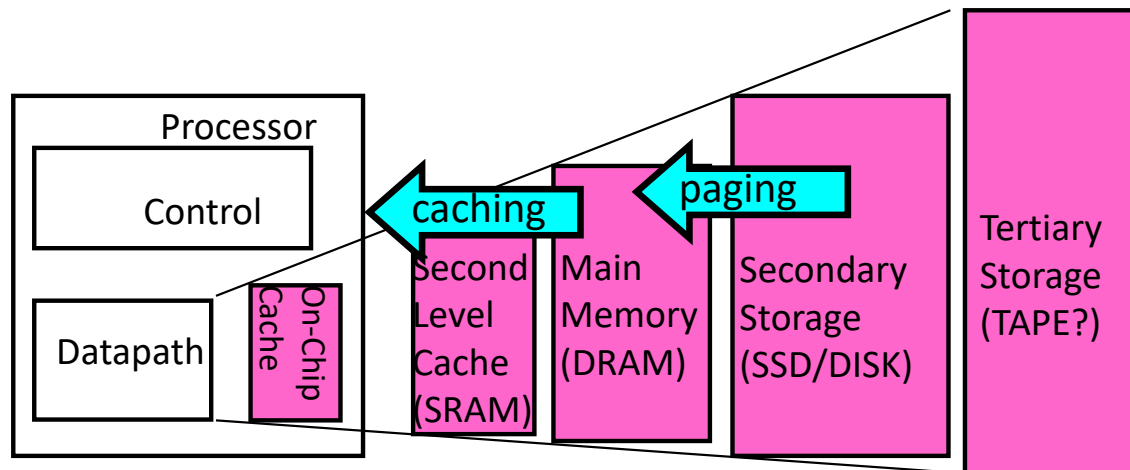


Page Fault \Rightarrow Demand Paging

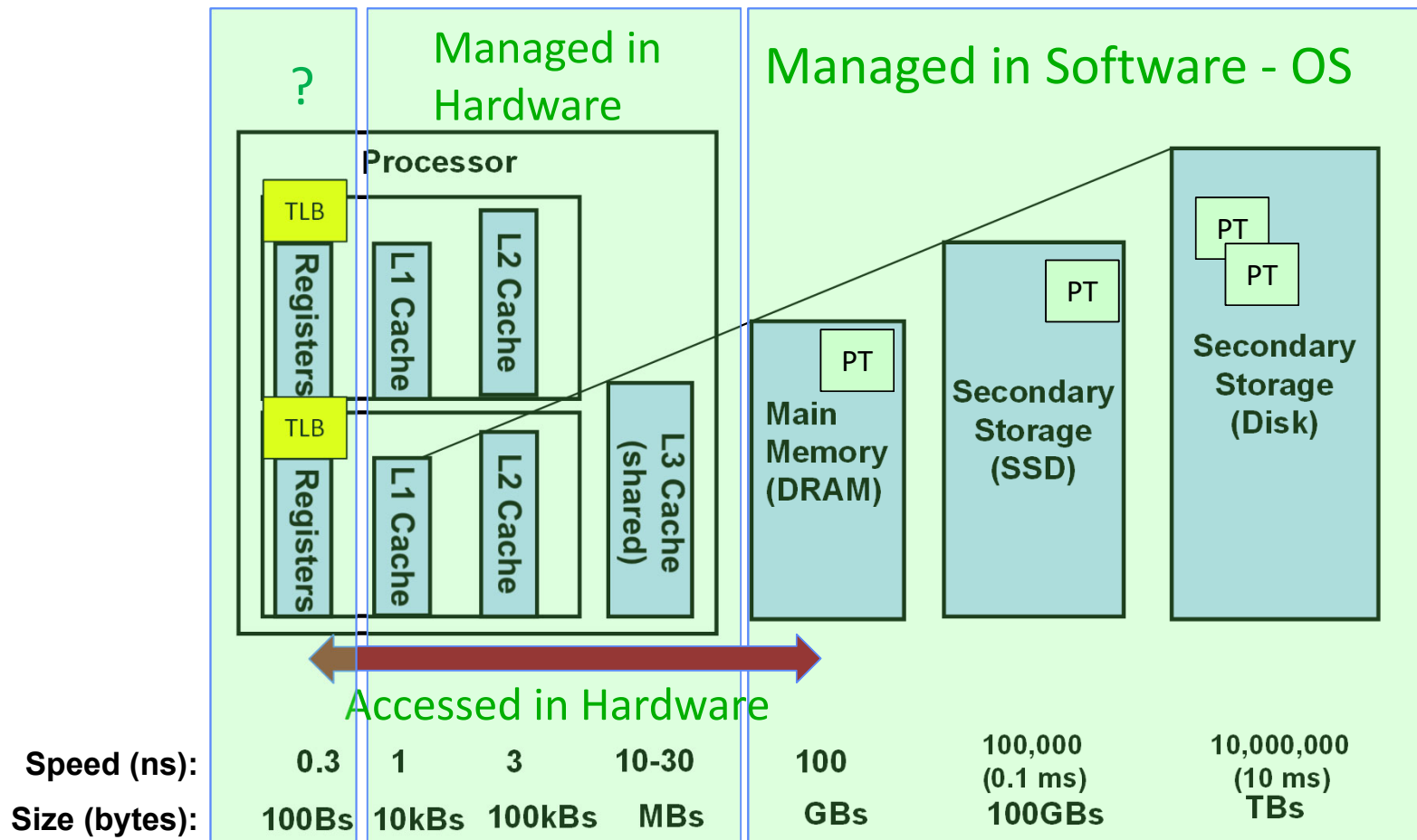


Demand Paging

- 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



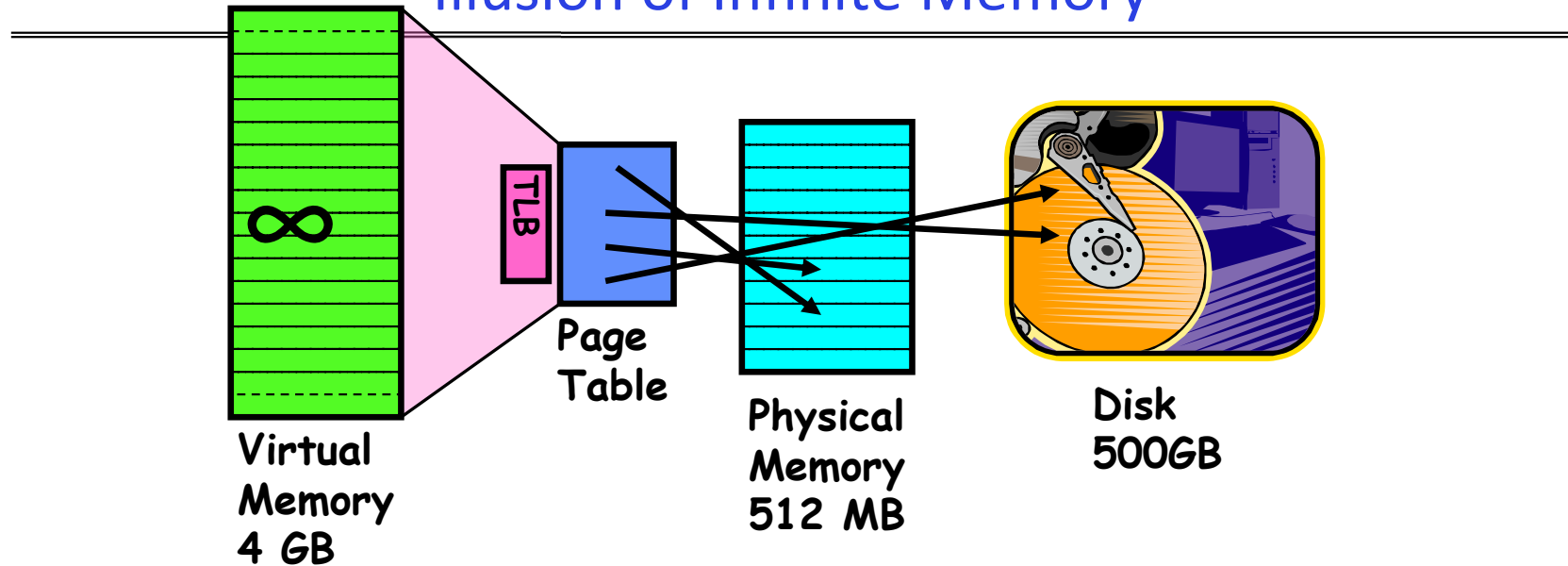
Management & Access to the Memory Hierarchy



Demand Paging as Caching, ...

- What “block size”? - 1 page (e.g, 4 KB)
- What “organization” ie. direct-mapped, set-assoc., fully-associative?
 - Fully associative since arbitrary virtual → physical mapping
- How do we locate a page?
 - 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!

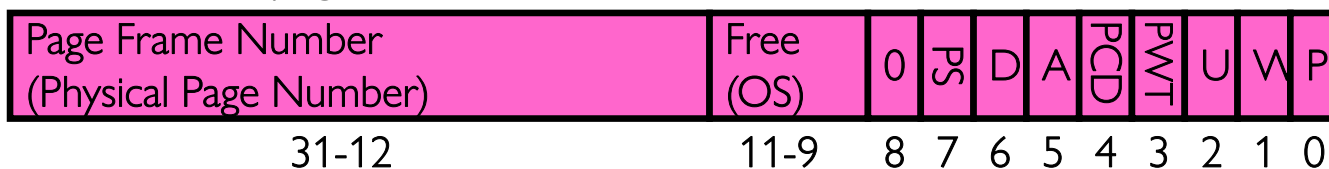
Illusion of Infinite Memory



- Disk is larger than physical memory \Rightarrow
 - 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

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”



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

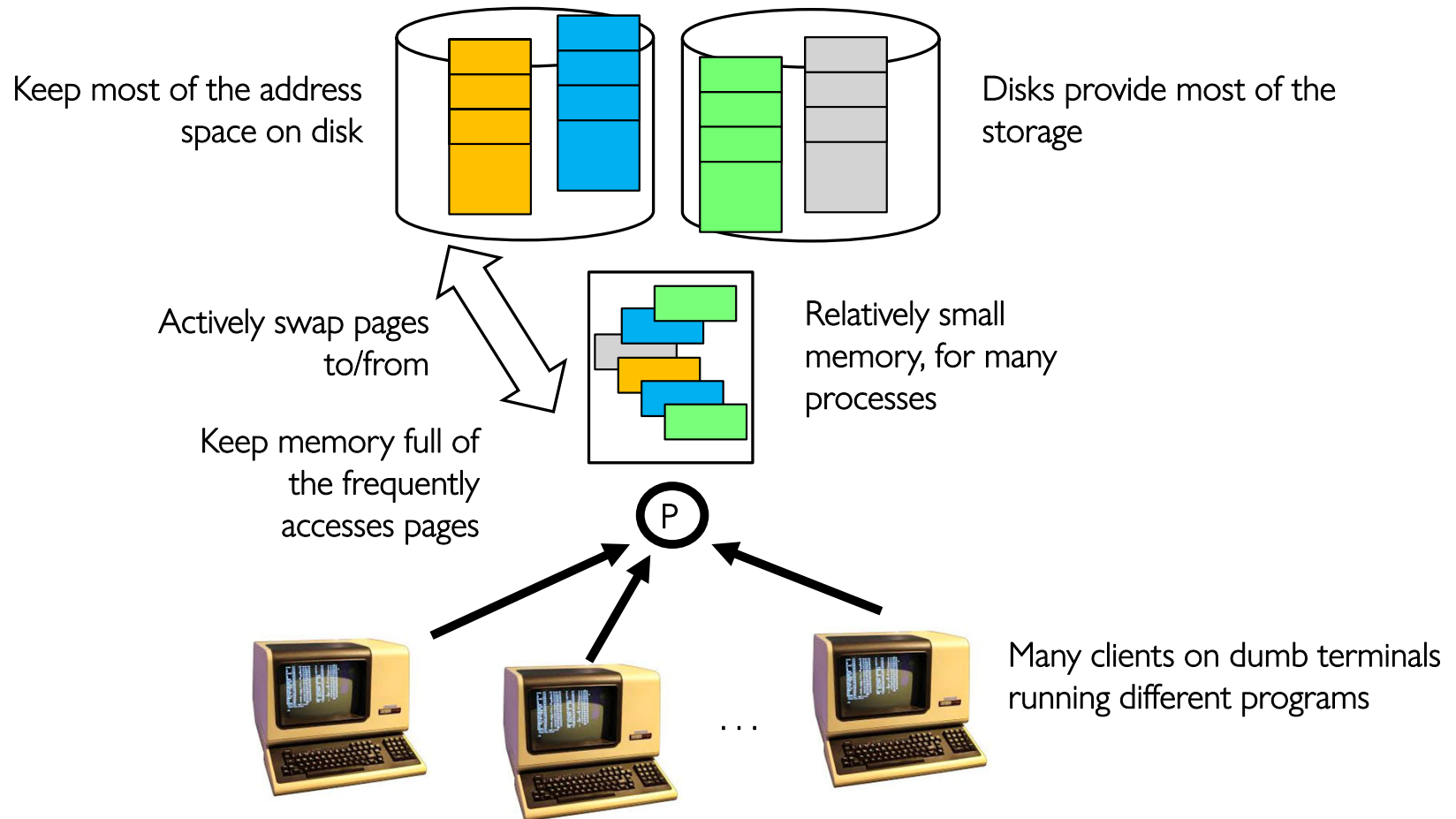
PS: Page Size: PS=1 ⇒ 4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

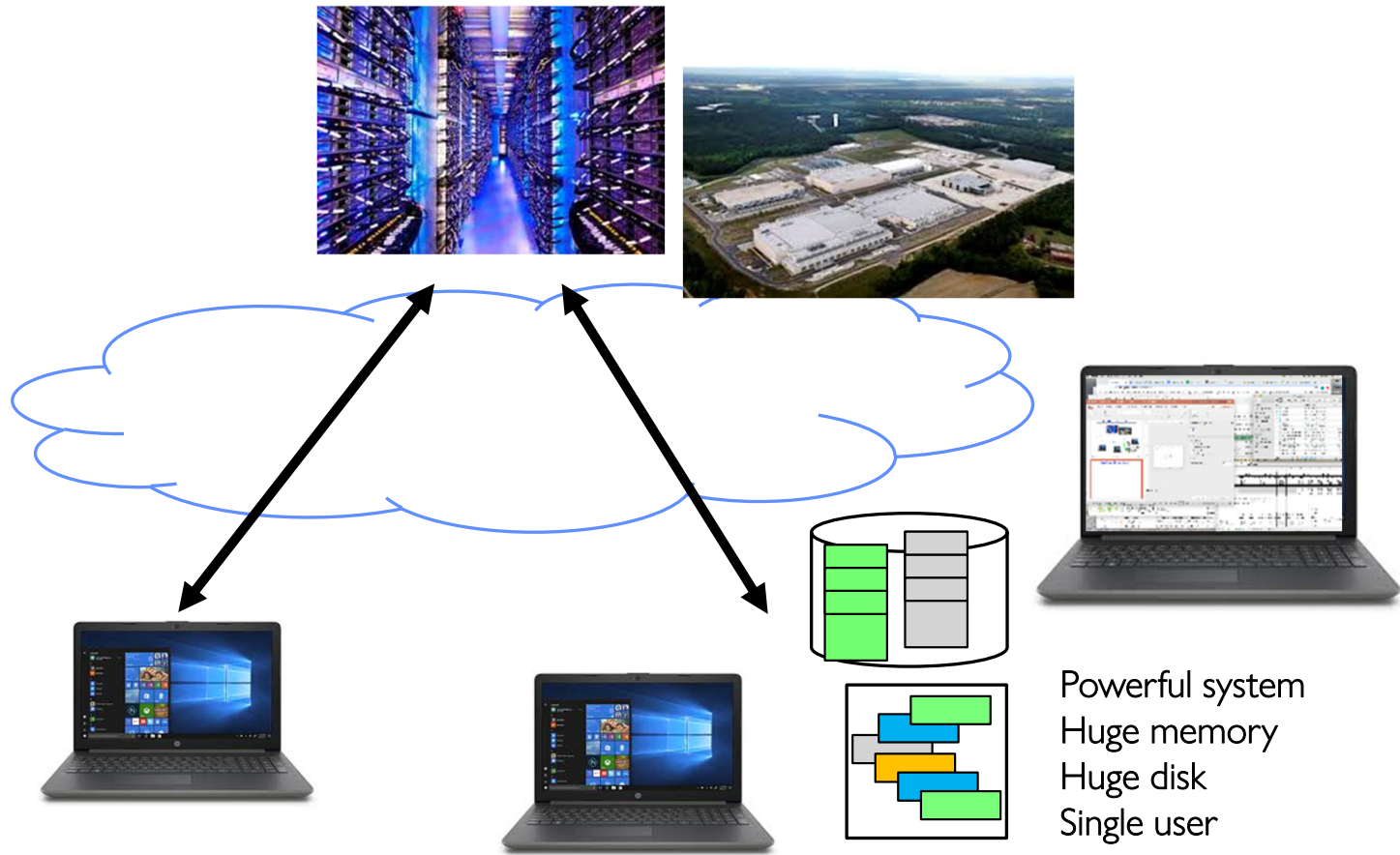
- PTE makes demand paging implementable
 - Valid \Rightarrow Page in memory, PTE points at physical page
 - Not Valid \Rightarrow Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
 - Memory Management Unit (MMU) traps to OS
 - » Resulting trap is a “Page Fault”
 - 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 continued!
 - While pulling pages off disk for one process, OS runs another process from ready queue
 - » Suspended process sits on wait queue

Cache

Origins of Paging



Very Different Situation Today



A Picture on one machine

```
Processes: 407 total, 2 running, 405 sleeping, 2135 threads                               22:10:39
Load Avg: 1.26, 1.26, 0.98 CPU usage: 1.35% user, 1.59% sys, 97.5% idle
SharedLibs: 292M resident, 54M data, 43M linkedit.
MemRegions: 155071 total, 4489M resident, 124M private, 1891M shared.
PhysMem: 13G used (3518M wired), 2718M unused.
VM: 1819G vsize, 1372M framework vsize, 68020510(0) swapins, 71200340(0) swapouts.
Networks: packets: 40629441/21G in, 21395374/7747M out.
Disks: 17026780/555G read, 15757470/638G written.

PID  COMMAND      %CPU  TIME    #TH   #WQ   #PORTS MEM   PURG   CMPRS  PGRP  PPID  STATE
90498 bash          0.0   00:00.41 1     0     21    1080K 0B     564K   90498 90497 sleeping
90497 login         0.0   00:00.10 2     1     31    1236K 0B     1220K   90497 90496 sleeping
90496 Terminal     0.5   01:43.28 6     1     378-  103M- 16M    13M    90496 1     sleeping
89197 siriknowledg  0.0   00:00.83 2     2     45    2664K 0B     1528K   89197 1     sleeping
89193 com.apple.DF  0.0   00:17.34 2     1     68    2688K 0B     1700K   89193 1     sleeping
82655 LookupViewSe  0.0   00:10.75 3     1    169    13M    0B     8064K   82655 1     sleeping
82453 PAH_Extensio  0.0   00:25.89 3     1    235    15M    0B     7996K   82453 1     sleeping
75819 tzlinkd       0.0   00:00.01 2     2     14    452K   0B     444K    75819 1     sleeping
75787 MTLCompilerS  0.0   00:00.10 2     2     24    9032K 0B     9020K   75787 1     sleeping
75776 secd        0.0   00:00.78 2     2     36    3208K 0B     2328K   75776 1     sleeping
75098 DiskUnmountW 0.0   00:00.48 2     2     34    1420K 0B     728K    75098 1     sleeping
75093 MTLCompilerS  0.0   00:00.06 2     2     21    5924K 0B     5912K   75093 1     sleeping
74938 ssh-agent     0.0   00:00.00 1     0     21    908K   0B     892K    74938 1     sleeping
74063 Google Chrom  0.0   10:48.49 15    1     678   192M   0B     51M     54320 54320 sleeping
```

- Memory stays about 75% used, 25% for dynamics
- A lot of it is shared 1.9 GB

Many Uses of Virtual Memory and “Demand Paging” ...

- Extend the stack
 - Allocate a page and zero it
- Extend the heap (sbrk of old, today mmap)
- Process Fork
 - Create a copy of the page table
 - Entries refer to parent pages – NO-WRITE
 - Shared read-only pages remain shared
 - Copy page on write
- Exec
 - Only bring in parts of the binary in active use
 - Do this on demand
- MMAP to explicitly share region (or to access a file as RAM)

Class Attendance Perk: 3/19/2026

- Take a selfie with the screen and QR code in the background
 - Want your face and the screen in same shot!
- Need to do this before end of class for some extra credit points.
- Happy Spring Break!

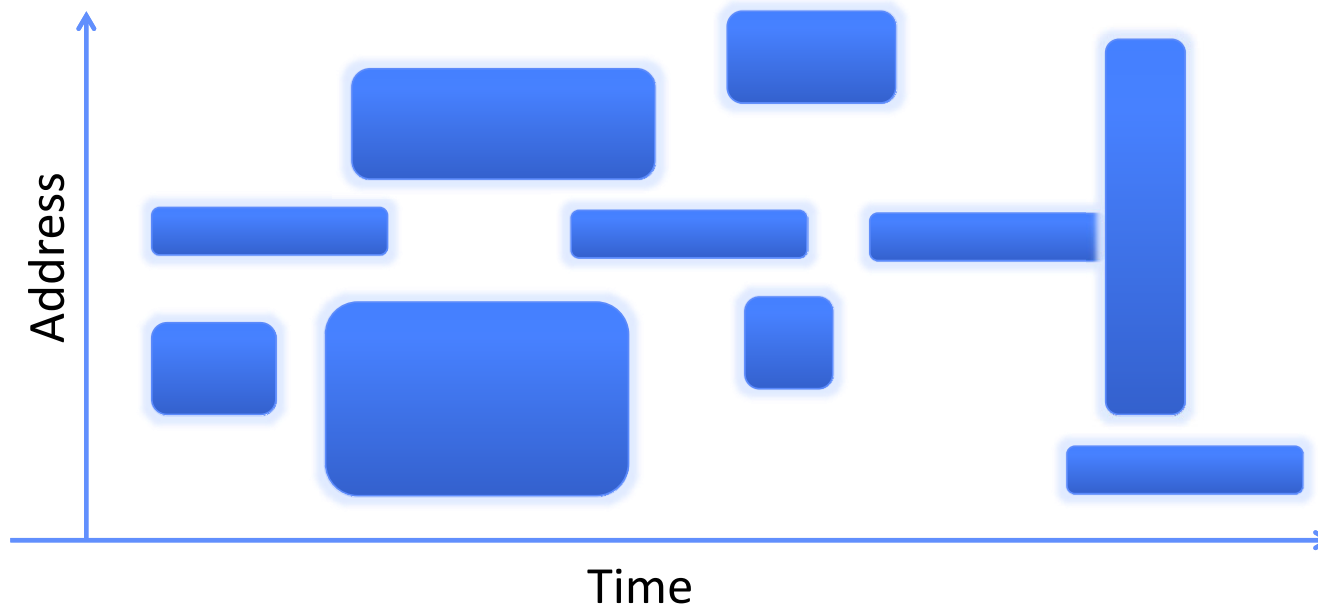


Some questions we need to answer!

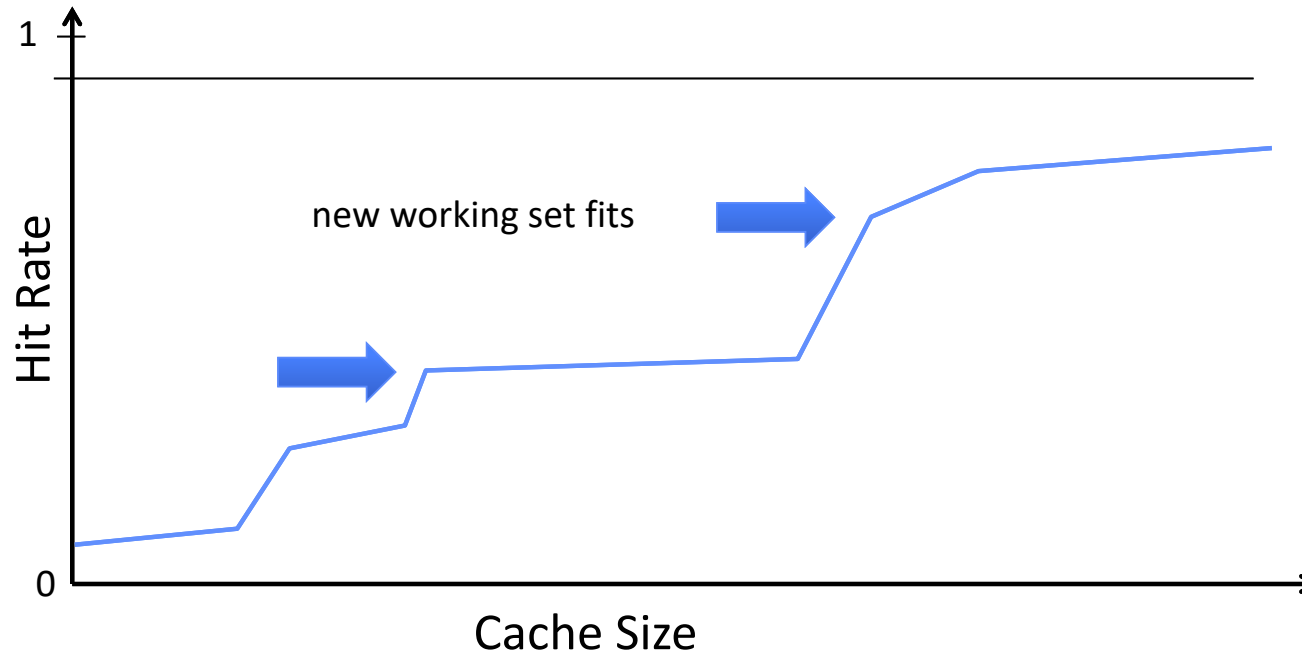
- During a page fault, where does the OS get a free frame?
 - Keeps a free list
 - Unix runs a “reaper” if memory gets too full
 - » Schedule dirty pages to be written back on disk
 - » Zero (clean) pages which haven’t been accessed in a while
 - As a last resort, evict a dirty page first
- How can we organize these mechanisms?
 - Work on the replacement policy
- How many page frames/process?
 - Like thread scheduling, need to “schedule” memory resources:
 - » Utilization? fairness? priority?
 - Allocation of disk paging bandwidth

Working Set Model

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



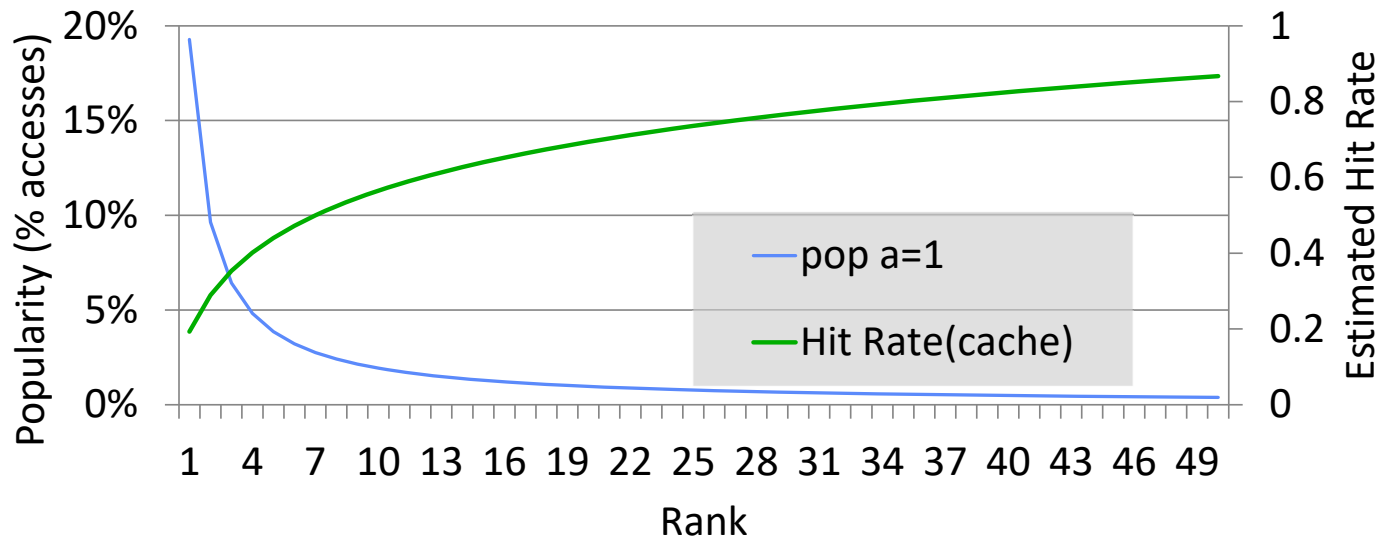
Cache Behavior under WS model



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

Another model of Locality: Zipf

$$P \text{ access}(\text{rank}) = 1/\text{rank}$$



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

Demand Paging Cost Model

- Since Demand Paging like caching, can compute average access time! (“Effective Access Time”)
 - $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$
 - $EAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$
- Example:
 - Memory access time = 200 nanoseconds
 - Average page-fault service time = 8 milliseconds
 - Suppose p = Probability of miss, $1-p$ = Probability of hit
 - Then, we can compute EAT as follows:
$$\begin{aligned} EAT &= 200\text{ns} + p \times 8 \text{ ms} \\ &= 200\text{ns} + p \times 8,000,000\text{ns} \end{aligned}$$
- If one access out of 1,000 causes a page fault, then $EAT = 8.2 \mu\text{s}$:
 - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
 - $EAT < 200\text{ns} \times 1.1 \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400,000!

What Factors Lead to Misses in Page Cache?

- **Compulsory Misses:**
 - Pages that have never been paged into memory before
 - How might we remove these misses?
 - » Prefetching: loading them into memory before needed
 - » Need to predict future somehow! More later
- **Capacity Misses:**
 - Not enough memory. Must somehow increase available memory size.
 - Can we do this?
 - » One option: Increase amount of DRAM (not quick fix!)
 - » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Conflict Misses:**
 - Technically, conflict misses don't exist in virtual memory, since it is a “fully-associative” cache
- **Policy Misses:**
 - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
 - How to fix? Better replacement policy

Conclusion

- “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
- Replacement policies
 - FIFO: Place pages on queue, replace page at end
 - MIN: Replace page that will be used farthest in future
 - LRU: Replace page used farthest in past
- Working Set:
 - Set of pages touched by a process recently
 - Point of Replacement algorithms is to try to keep working set in memory