CS162
Operating Systems and Systems Programming
Lecture 14

Caching (Finished), Demand Paging

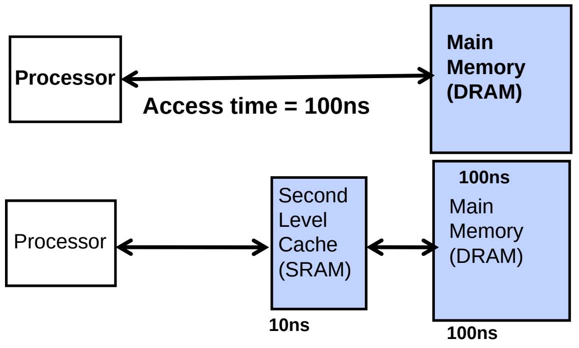
March 16th, 2015

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

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

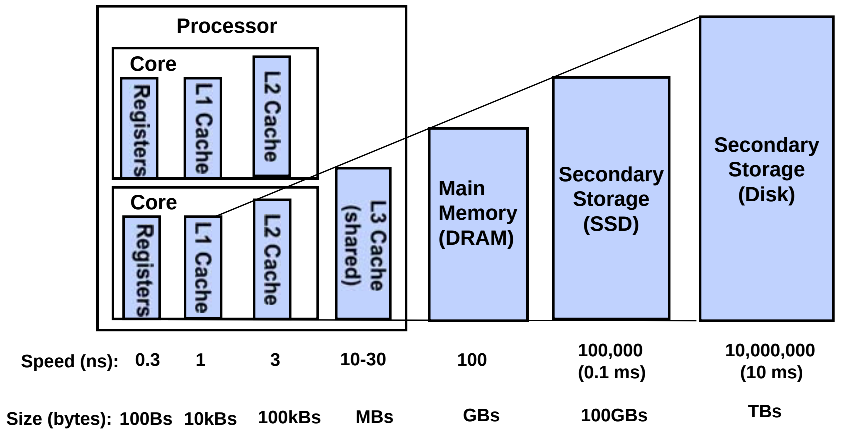
- Caching is the key to memory system performance



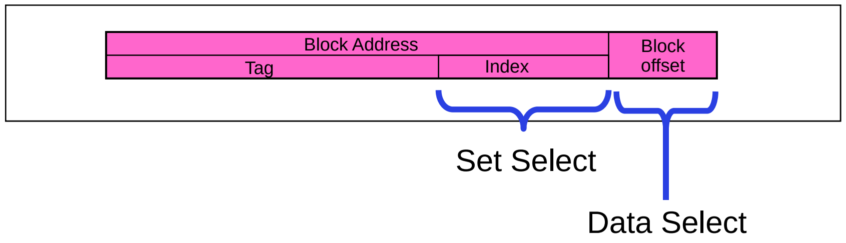
- Average Access time = (Hit Rate x HitTime) + (Miss Rate x MissTime)
- HitRate + MissRate = 1
- HitRate = 90% => Average Access Time = 19 ns
- HitRate = 99% => Average Access Time = 10.9ns

Review: 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



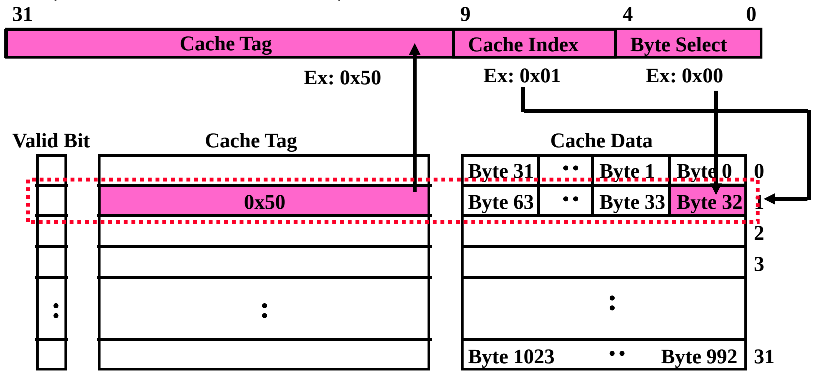
Recall: How is a Block found in a Cache?



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

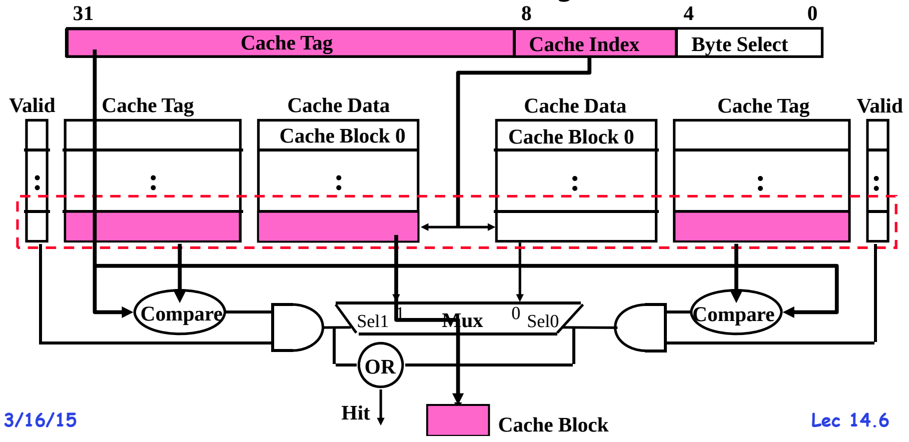
Review: Direct Mapped Cache

- 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



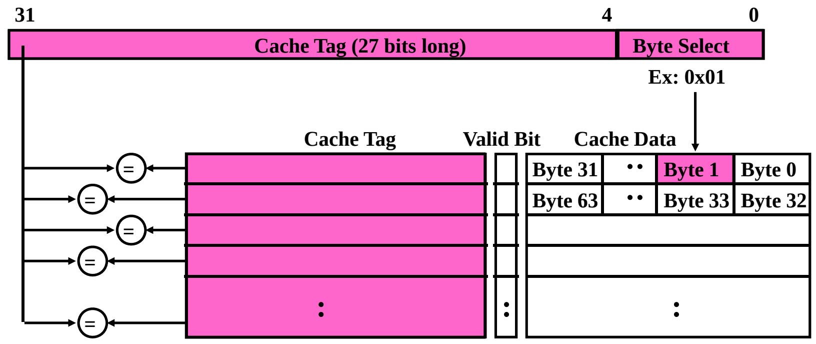
Review: Set Associative Cache

- 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



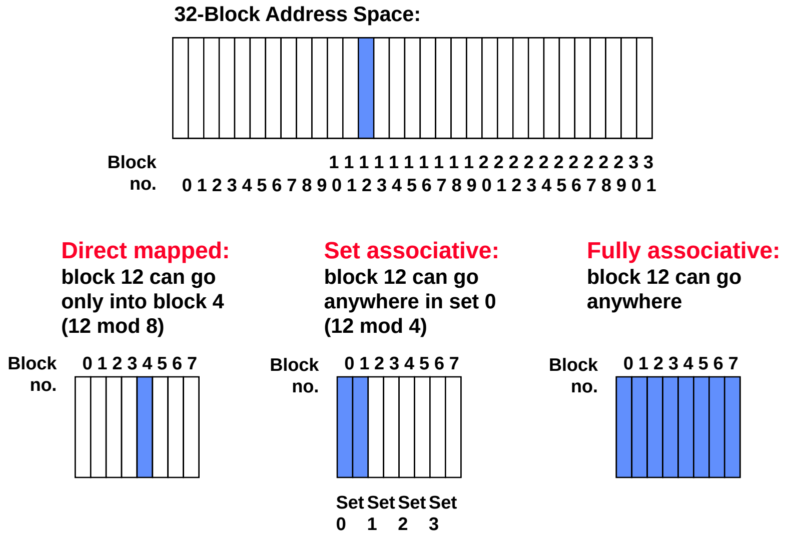
Review: Fully Associative Cache

- 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



Review: Which block should be replaced on a miss?

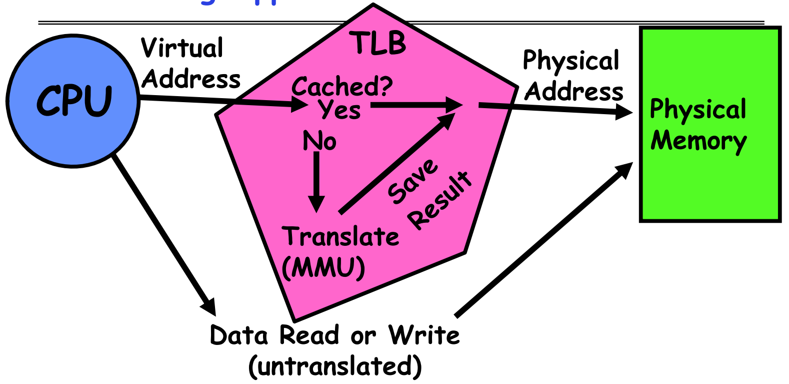
- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

	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

Caching Applied to Address Translation

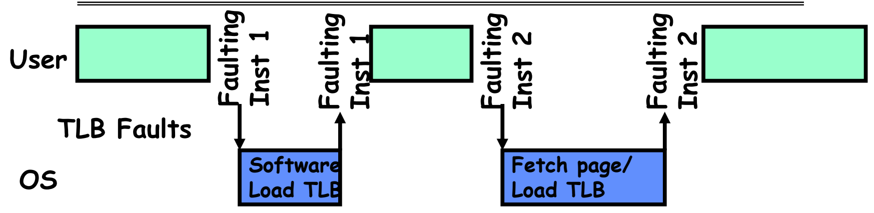


- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple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

What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables (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: TLB/Page fault



- How to transparently restart faulting instructions?
  - (Consider load or store that gets TLB or Page fault)
  - Could we just skip faulting instruction?
    - » No: need to perform load or store after reconnecting physical page
- Hardware must 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)`
    - » Precise exception state consists of 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 ⇒ 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, ...
  - MIPS takes this position
- Imprecise ⇒ 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

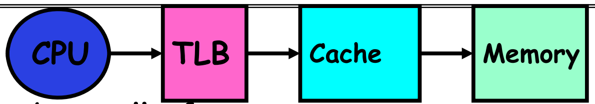
Administrivia

- Still working on the grading of exams
  - No deadline yet, will let you know
- Solutions are done!
  - Will post them on the website
- Checkpoint 1 for Project 2 delayed
  - Now due Monday after Spring Break

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"

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!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- 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

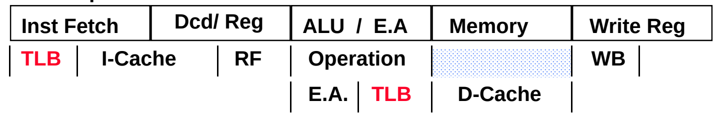
TLB organization: include protection

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

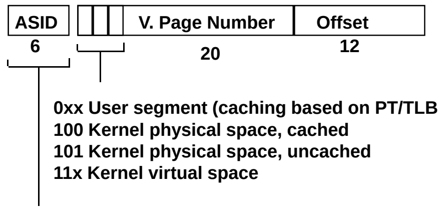
MIPS R3000 Pipeline



TLB

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

Virtual Address Space

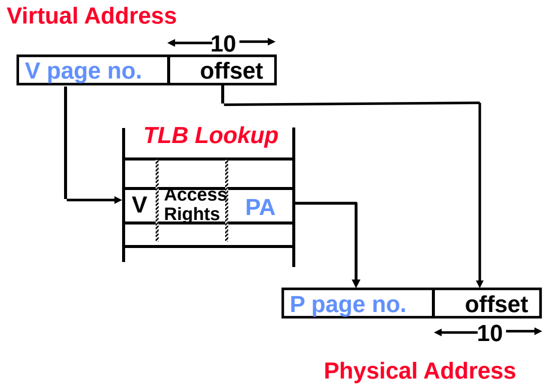


0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

Reducing translation time further

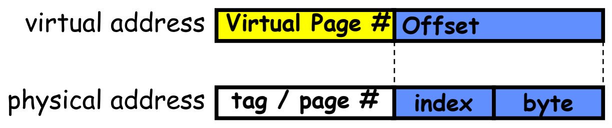
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

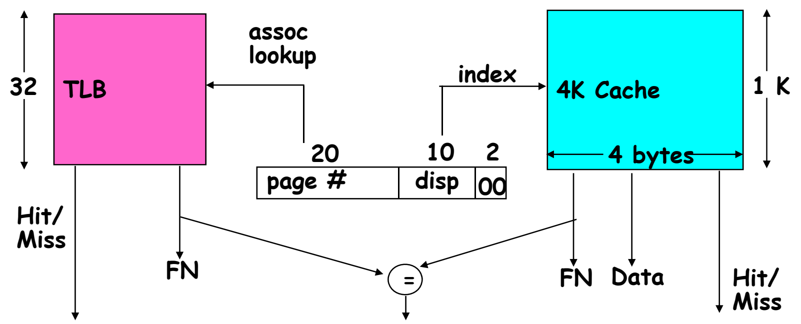
Overlapping TLB & Cache Access (1/2)

- Main idea:
  - 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



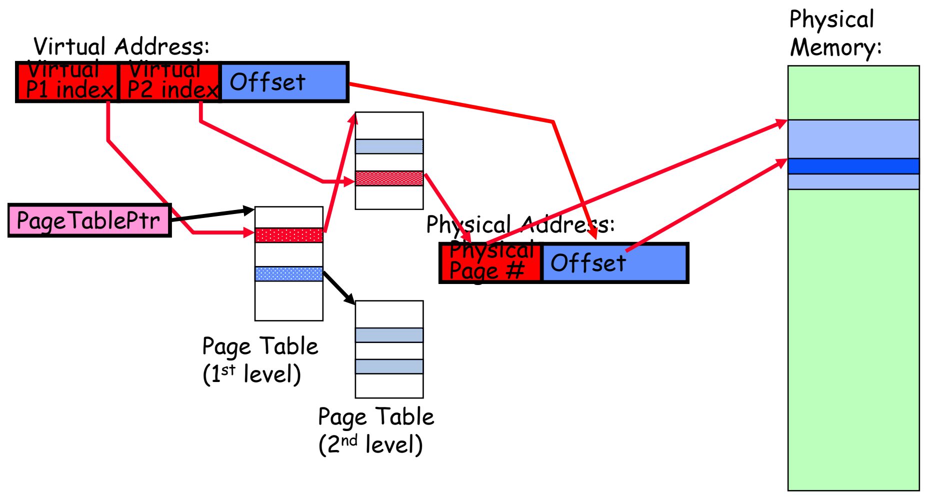
Overlapping TLB & Cache Access

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

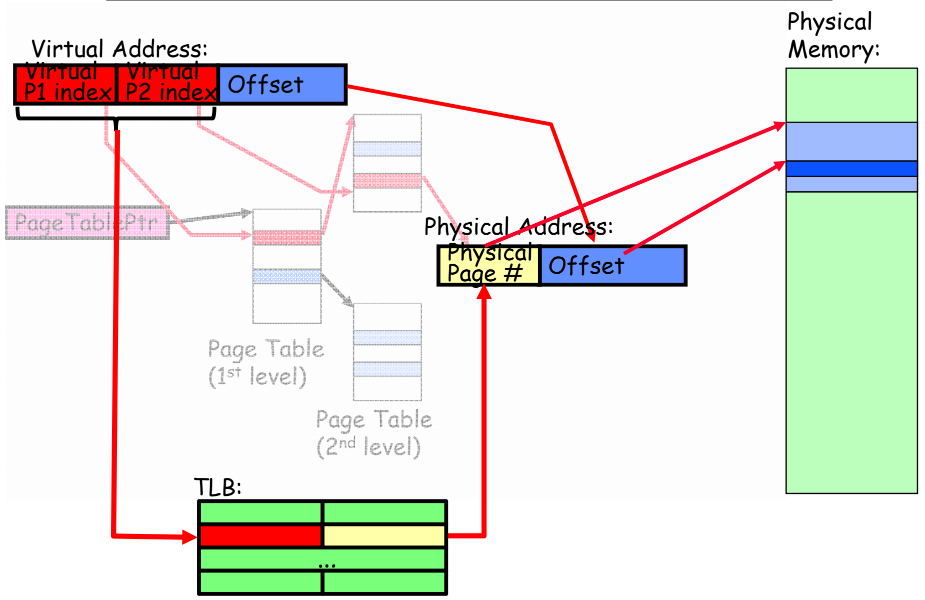


- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

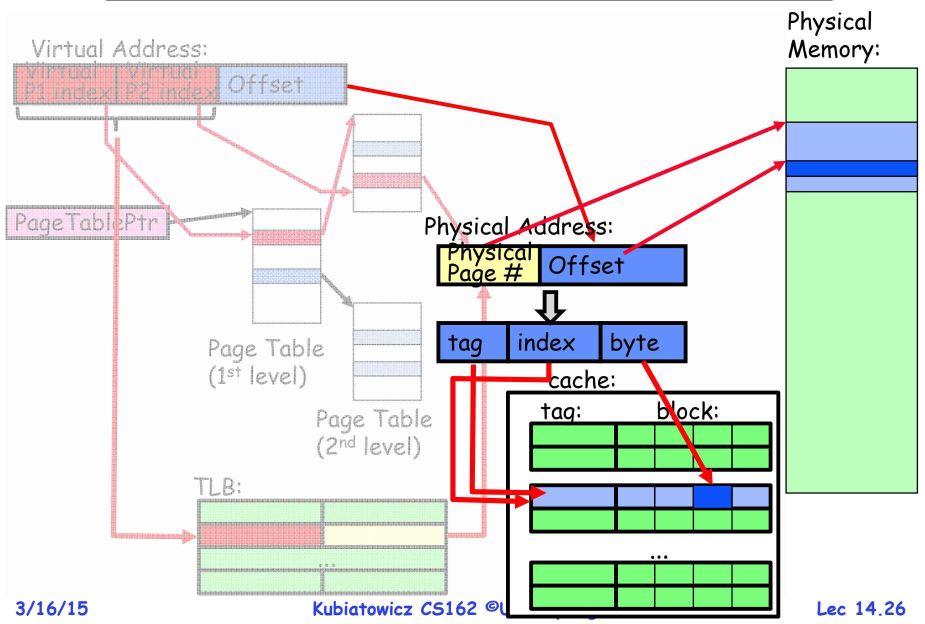
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: Cache



Where are all places that caching arises in Operating Systems?

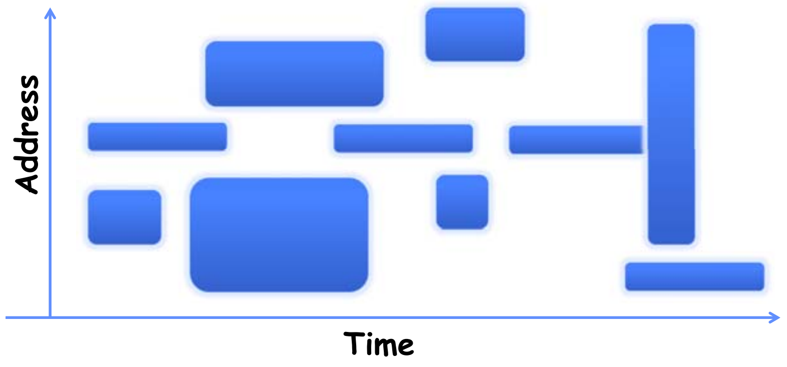
- Direct use of caching techniques
  - paged virtual memory (mem as cache for disk)
  - TLB (cache of PTEs)
  - file systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)
- Which pages to keep in memory?
  - All-important "Policy" aspect of virtual memory
  - Will spend a bit more time on this in a moment

Impact of caches on Operating Systems

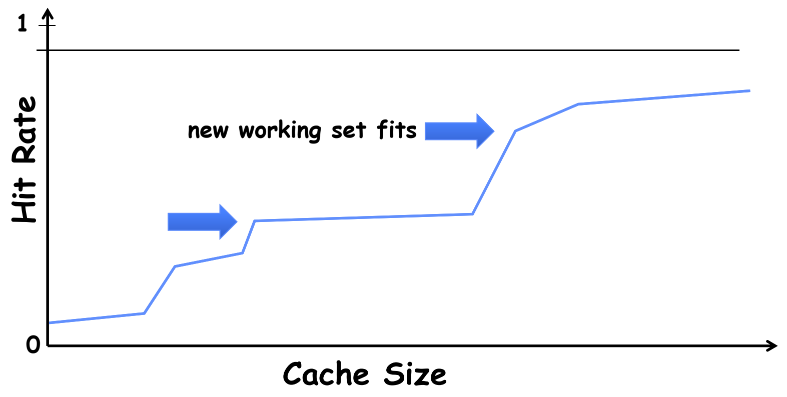
- Indirect - dealing with cache effects
- Process scheduling
  - which and how many processes are active ?
  - large memory footprints versus small ones ?
  - priorities ?
  - Shared pages mapped into VAS of multiple processes ?
- Impact of thread scheduling on cache performance
  - rapid interleaving of threads (small quantum) may degrade cache performance
    - » increase average memory access time (AMAT) !!!
- Designing operating system data structures for cache performance
- Maintaining the correctness of various caches
  - TLB consistency:
    - » With PT across context switches ?
    - » Across updates to the PT ?

Working Set Model

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

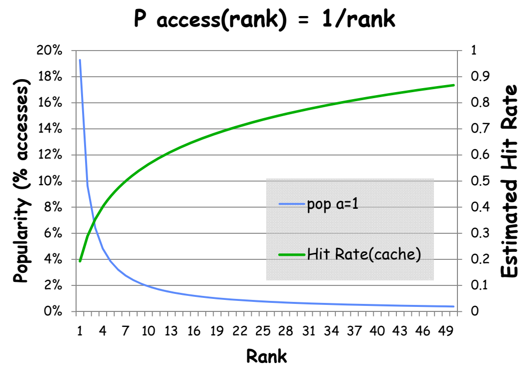


Cache Behavior under WS model



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

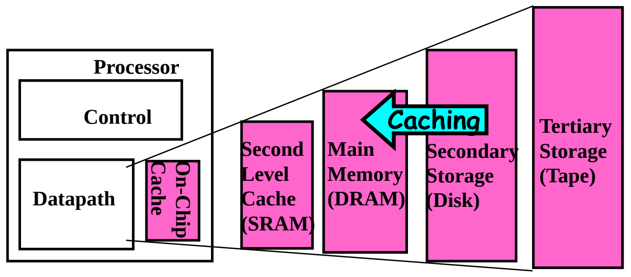
Another model of Locality: Zipf



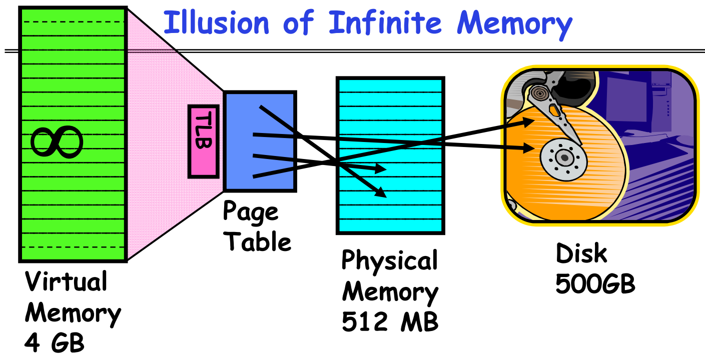
- Likelihood of accessing item of rank r is α1/r^a
- Although rare to access items below the top few, there are so many that it yields a "heavy tailed" distribution.
- Substantial value from even a tiny cache
- Substantial misses from even a very large one

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 is Caching

- Since Demand Paging is 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:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

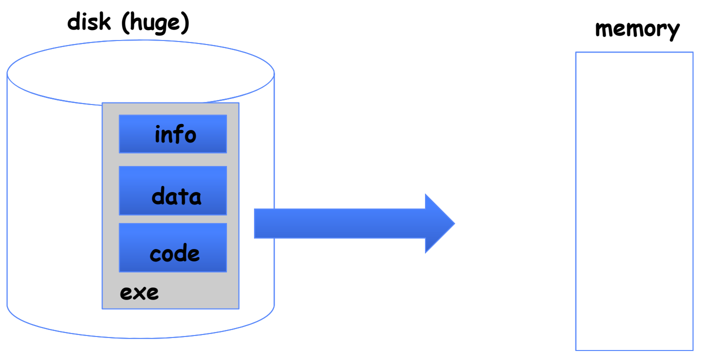
Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

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

Demand Paging Mechanisms

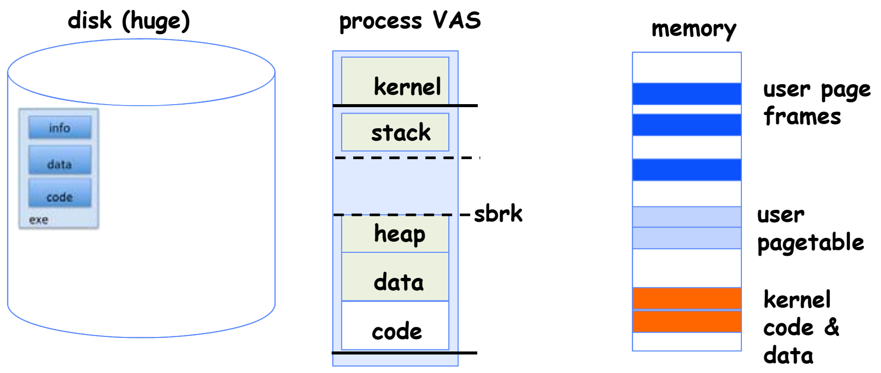
- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is 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

Loading an executable into memory



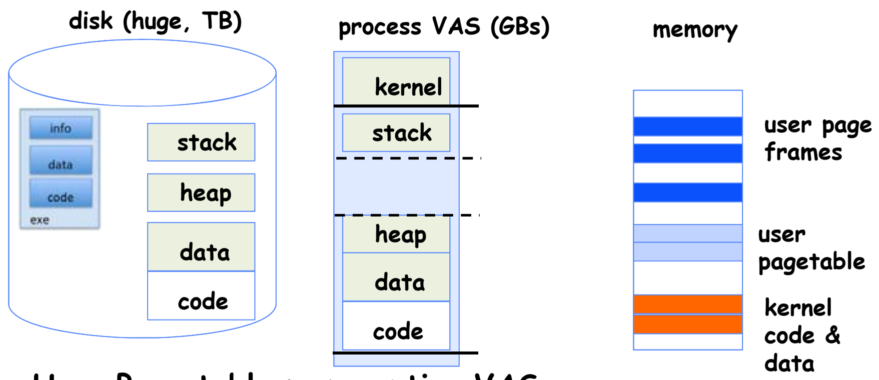
- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization: CRT0

Create Virtual Address Space of the Process



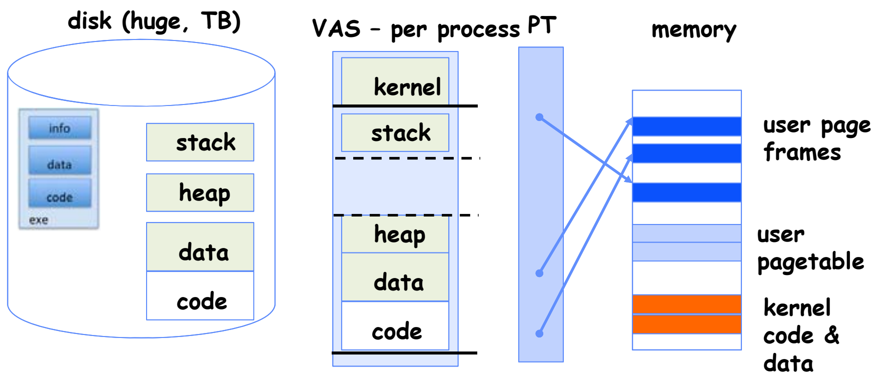
- Utilized pages in the VAS are backed by a page block on disk
  - called the backing store
  - typically in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



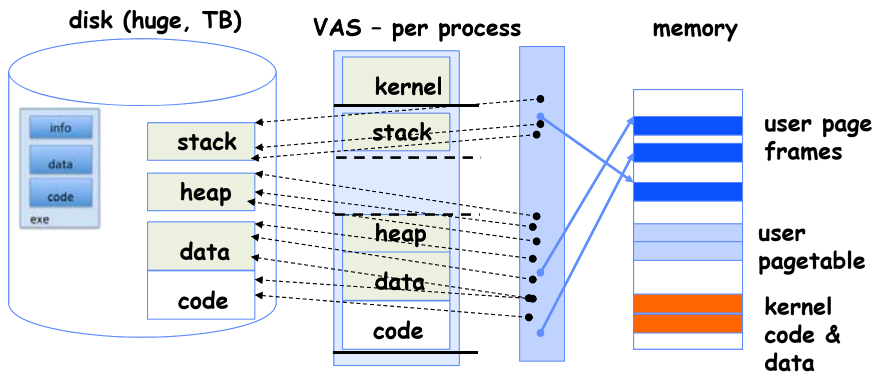
- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - swapped into and out of memory as needed
- For *every* process

Create Virtual Address Space of the Process



- User Page table maps entire VAS
  - resident pages to the frame in memory they occupy
  - the portion of it that the HW needs to access must be resident in memory

Provide Backing Store for VAS

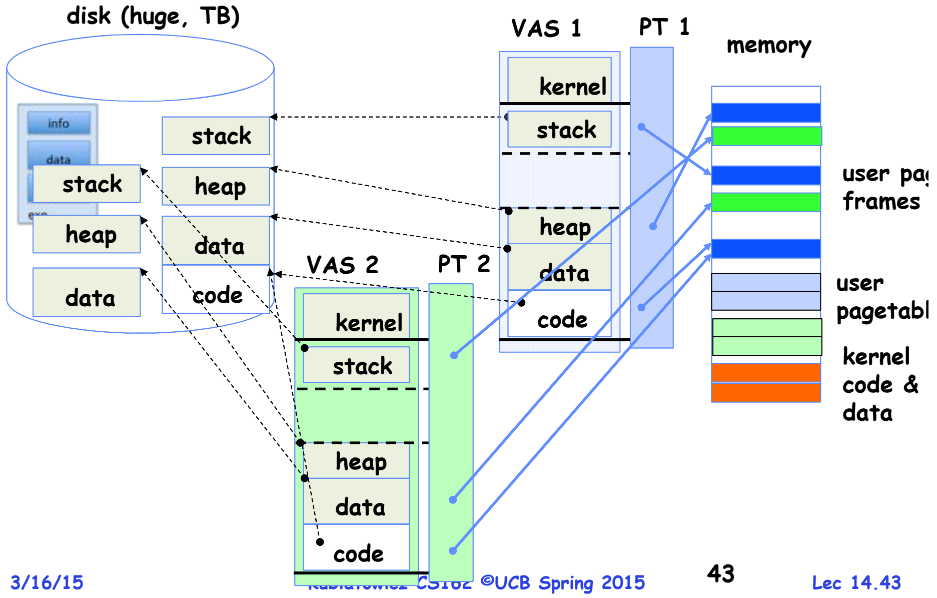


- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk

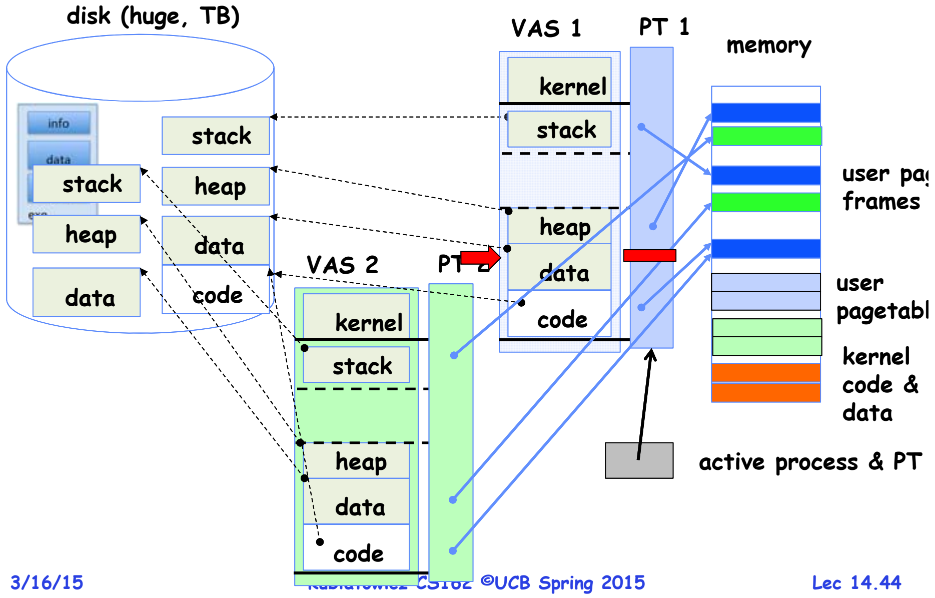
What data structure is required to map non-resident pages to disk?

- FindBlock(PID, page#) => disk_block
  - Some OSs utilize spare space in PTE for paged blocks
  - Like the PT, but purely software
- Where to store it?
  - In memory – can be compact representation if swap storage is contiguous on disk
  - Could use hash table (like Inverted PT)
- Usually want backing store for resident pages too.
- May map code segment directly to on-disk image
  - Saves a copy of code to swap file
- May share code segment with multiple instances of the program

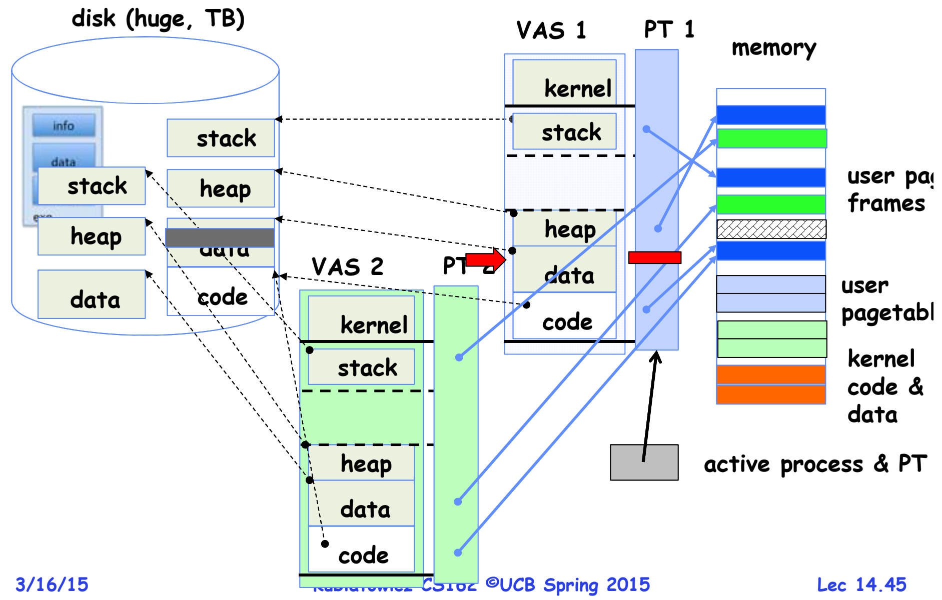
Provide Backing Store for VAS



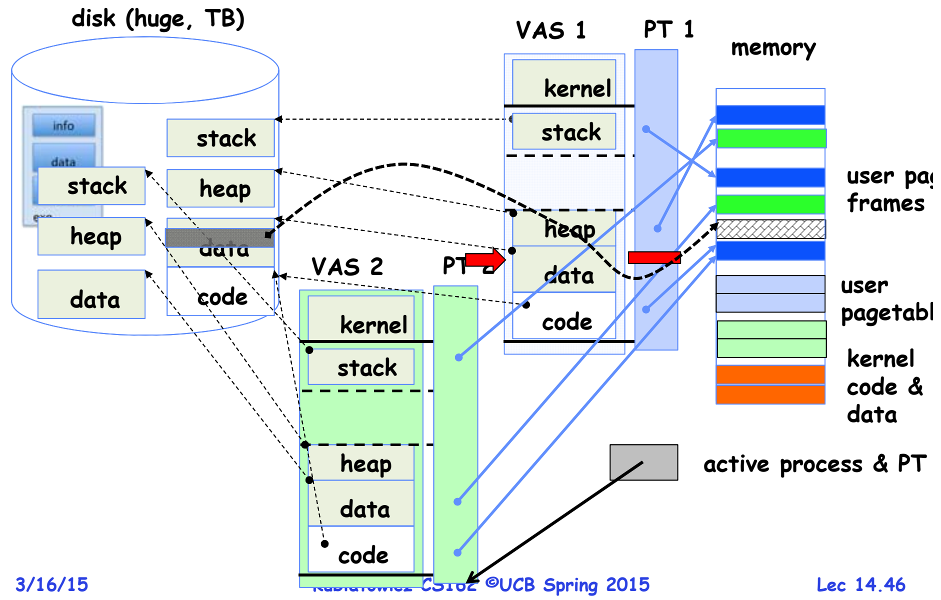
On page Fault …



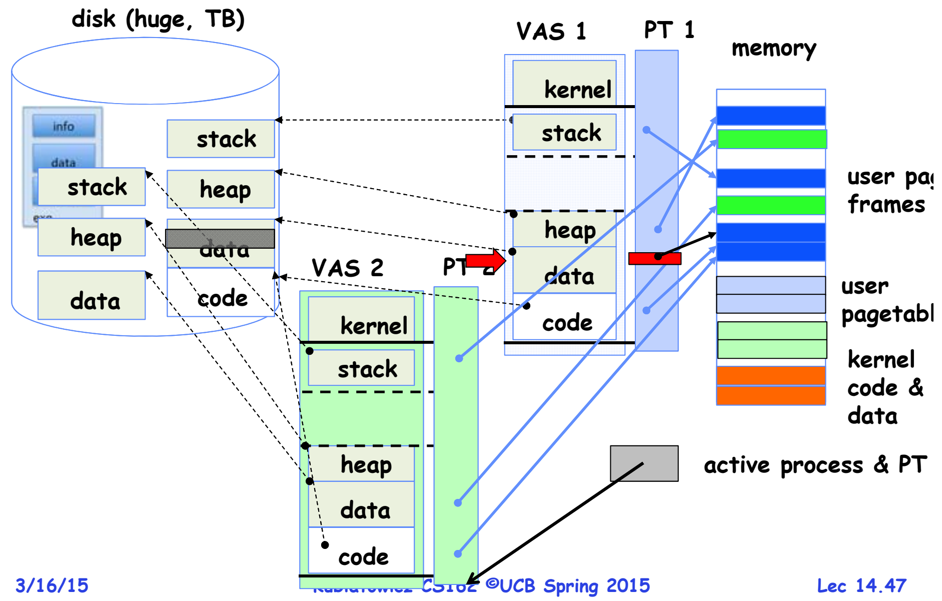
On page Fault … find & start load



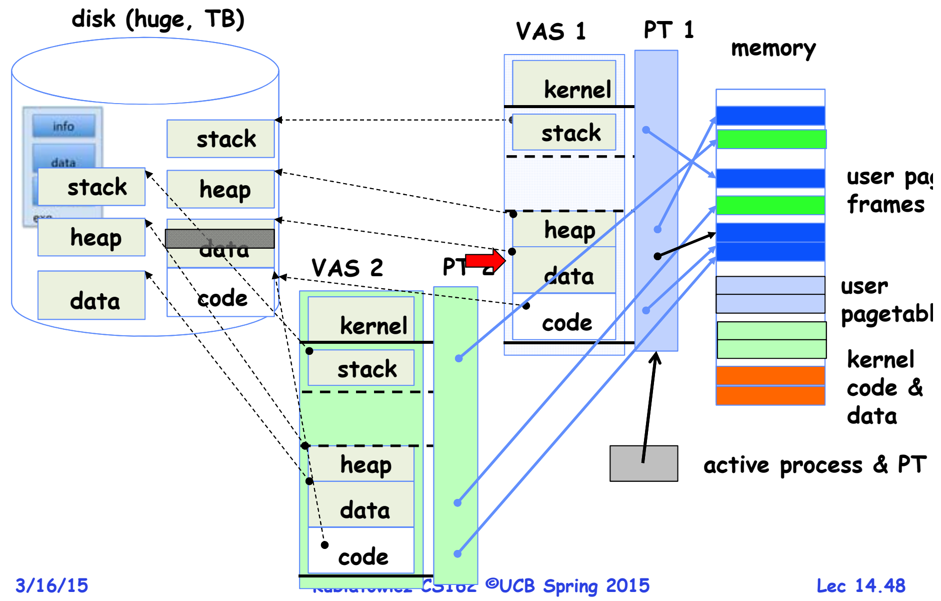
On page Fault … schedule other P or T



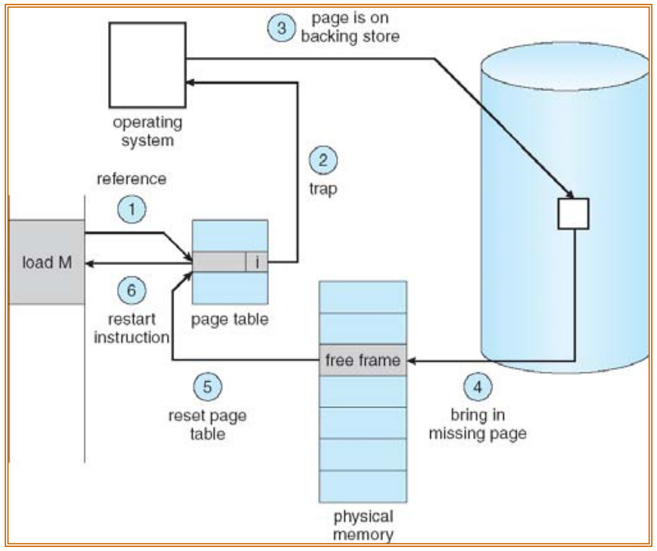
On page Fault … update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault



Demand Paging (more details)

- Does software-loaded TLB need use bit? Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    - » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    - » Can't push page out to disk without invalidating all PTEs

Summary

- A cache of translations called a "Translation Lookaside Buffer" (TLB)
  - Relatively small number of entries (< 512)
  - Fully Associative (Since conflict misses expensive)
  - TLB entries contain PTE and optional process ID
- On TLB miss, page table must be traversed
  - If located PTE is invalid, cause Page Fault
- On context switch/change in page table
  - TLB entries must be invalidated somehow
- TLB is logically in front of cache
  - Thus, needs to be overlapped with cache access to be really fast
- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started