

CS162 Operating Systems and Systems Programming Lecture 14

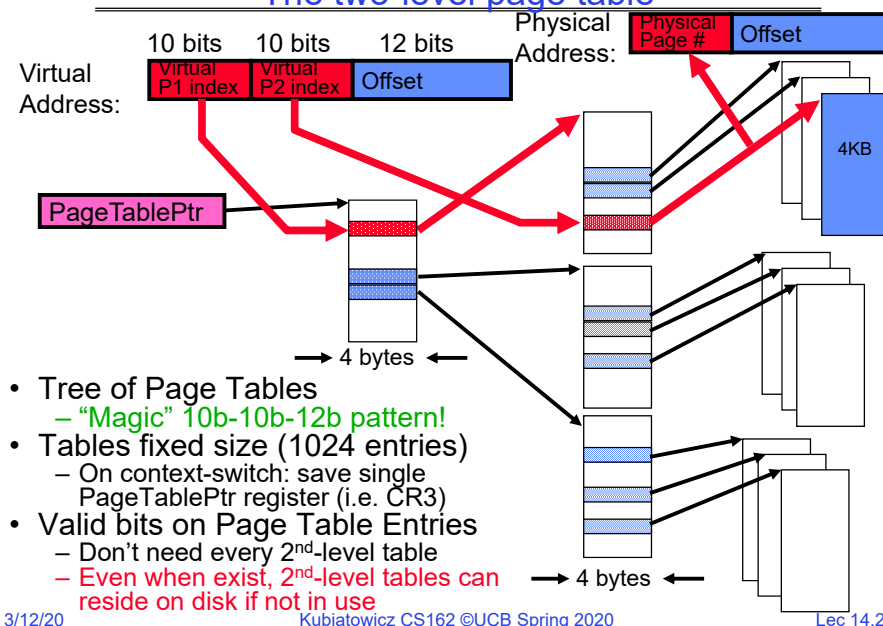
Caching and TLBs (Finished), Demand Paging

March 12th, 2020

Prof. John Kubiatowicz

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

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

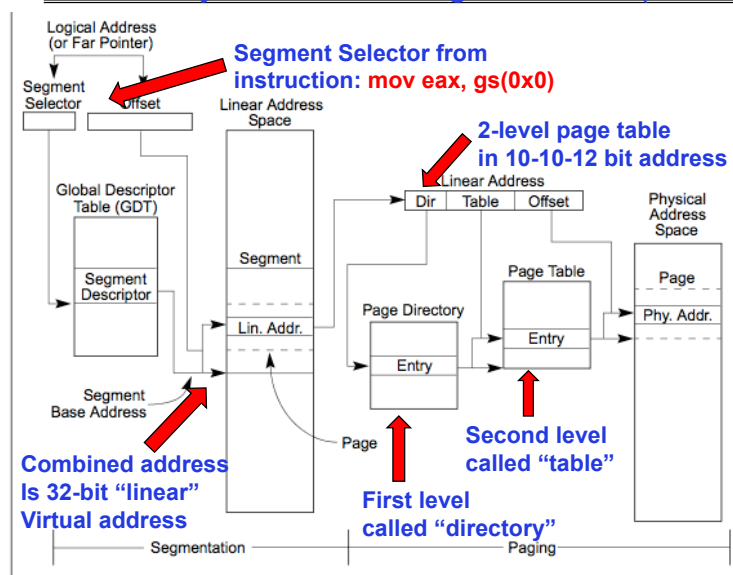


3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.2

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



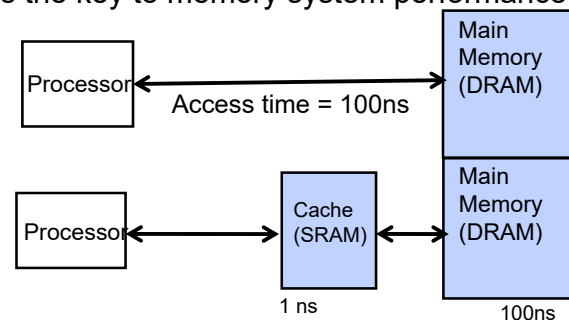
3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.3

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

- Caching is the key to memory system performance



Average Memory Access Time (AMAT)

$$= (\text{Hit Rate} \times \text{HitTime}) + (\text{Miss Rate} \times \text{MissTime})$$

Where HitRate + MissRate = 1

$$\text{HitRate} = 90\% \Rightarrow \text{AMAT} = (0.9 \times 1) + (0.1 \times 101) = 11.1 \text{ ns}$$

$$\text{HitRate} = 99\% \Rightarrow \text{AMAT} = (0.99 \times 1) + (0.01 \times 101) = 2.01 \text{ ns}$$

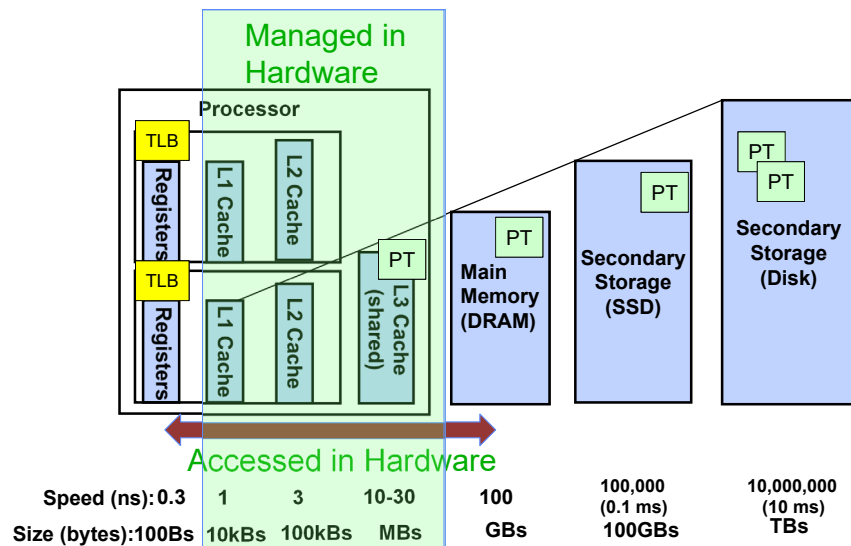
$$\text{MissTime}_{L1} \text{ includes } \text{HitTime}_{L1} + \text{MissPenalty}_{L1} \equiv \text{HitTime}_{L1} + \text{AMAT}_{L2}$$

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.4

Recall: The Memory Hierarchy



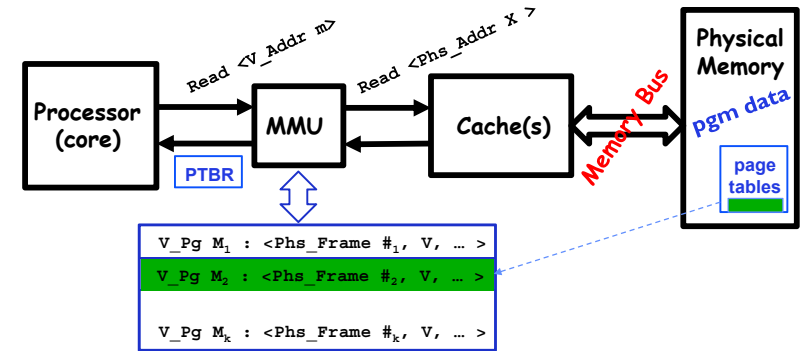
3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.5

Recall: How to make Address Translation Fast?

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



3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.6

Translation Look-Aside Buffer

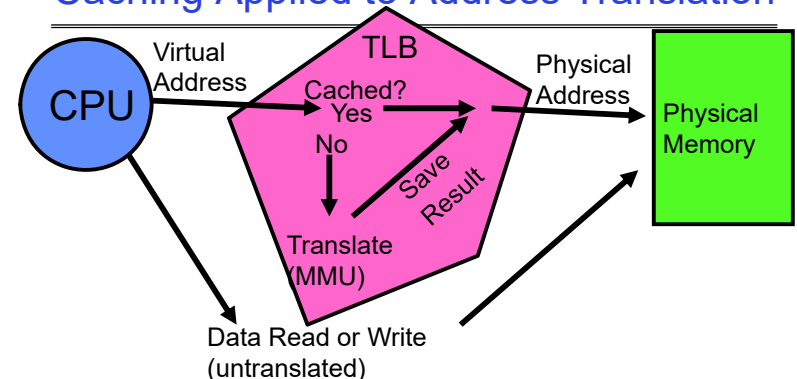
- TLB is a cache of translations:
 - Record recent Virtual Page # to Physical Frame # translation
- If present, get the physical address from TLB without reading any of the page tables !!!
 - Even if the translation involved multiple levels
 - Caches the end-to-end result
- Was invented by Sir Maurice 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
 - Ultimately invokes page table walk

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.7

Caching Applied to Address Translation



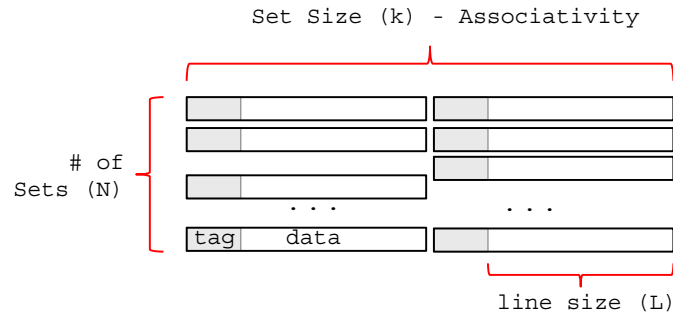
- Question is one of page locality: does it exist?
 - Instruction accesses spend a lot of time on the same page (since accesses sequential)
 - Stack accesses have definite locality of reference
 - Data accesses have less page locality, but still some...
- Can we have a TLB hierarchy?
 - Sure: multiple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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.8

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, ...)

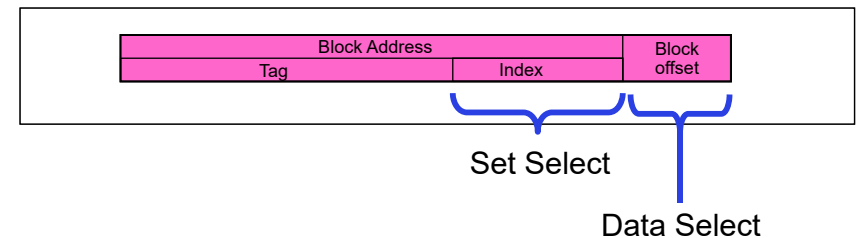
How might organization of TLB differ from that of a conventional instruction or data cache?

- Let's do some review ...

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
 - 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

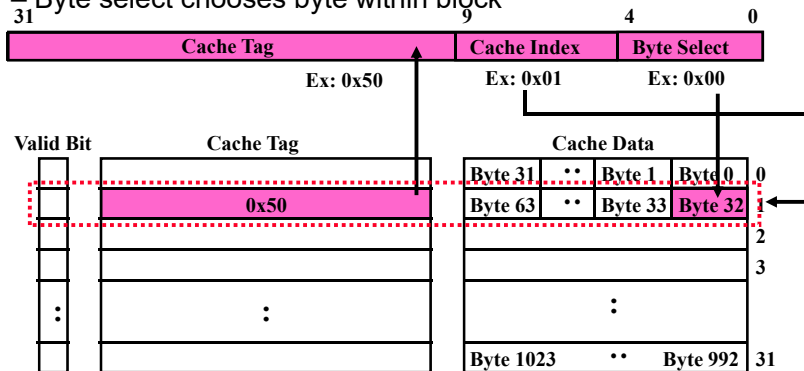
How is a Block found in a Cache?



- Block** is minimum quantum of caching
 - Data select field used to select data (byte) 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



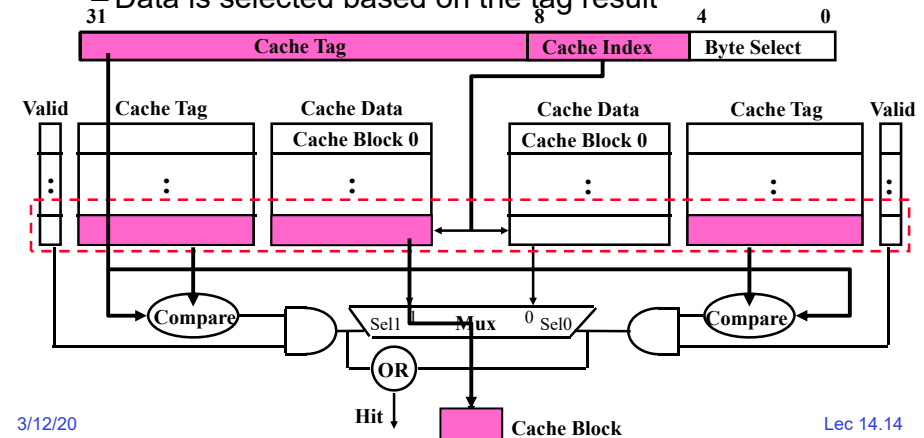
3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.13

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

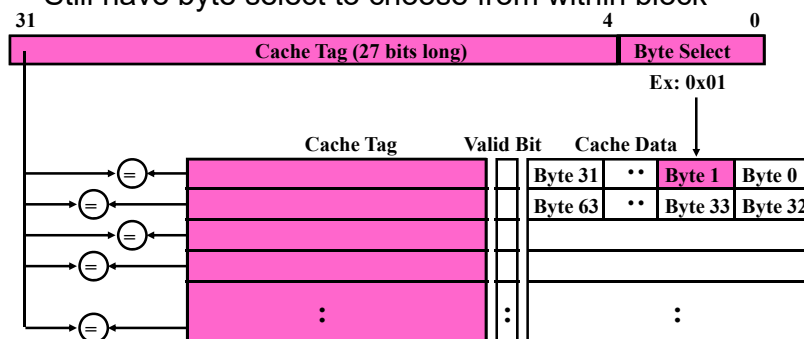


3/12/20

Lec 14.14

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



3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.15

Administrivia (1/2)

- **Saturday (3/14) is π Day!!!**
 - 40 digits are sufficient to calculate circumference of visible universe to atomic dimensions:
- See: <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 π 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 last year (3/14/19) that Emma Haruka Iwao calculated π to 31,415,926,535,897 digits (new record...)

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.16

Administrivia (2/2)

- Please remember to fill out your Project 1 Peer evaluations!
 - It is very important that you fill these out!
 - It is as important as getting to know your TA.
 - The project grades are a zero-sum game; if you do not contribute to the project, your points might be distributed to those who do!
- Project 2 Design Docs due Yesterday
- (Virtual) Design Reviews are still MANDATORY
 - They are an essential Oral Exam: you discuss design with your TA
 - Everyone must be present on the Zoom call (and be able to enable cameras): Remember that your TA is evaluating your participation as part of the design process!
- Midterm 2: Still Scheduled for 4/02
 - Will likely be virtual, we don't know yet
 - Ok, this is a few weeks and after Spring Break
 - Not sure what is going to happen yet; Study for it!
 - Material is up to and including some material from lecture 18

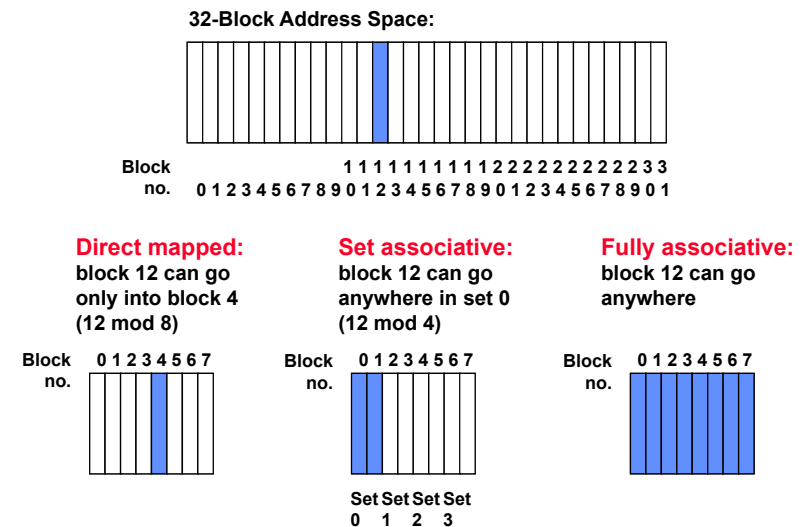
3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.17

Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache



3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.18

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%

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.19

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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.20

Impact of caches on Operating Systems

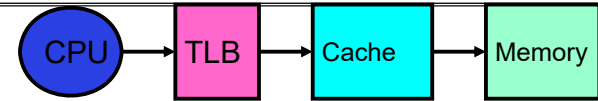
- Dealing with cache effects
 - Maintaining the correctness of various caches
 - E.g., TLB consistency:
 - » With PT across context switches ?
 - » Across updates to the PT ?
- Process scheduling
 - Which and how many processes are active ? Priorities ?
 - Large memory footprints versus small ones ?
 - 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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.21

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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.22

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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.23

Example: R3000 pipeline includes TLB “stages”

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

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

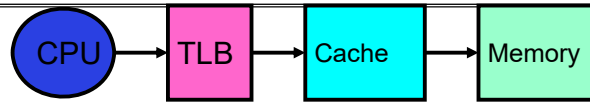
Allows context switching among
 64 user processes without TLB flush

3/12/20

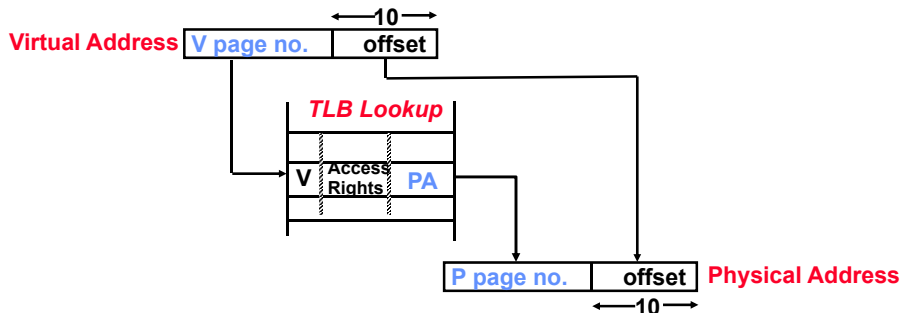
Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.24

Reducing translation time further



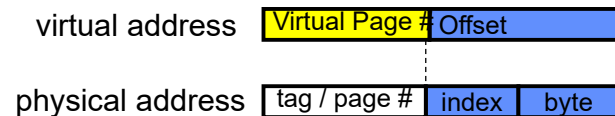
- As described, TLB lookup is in serial with cache lookup:



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

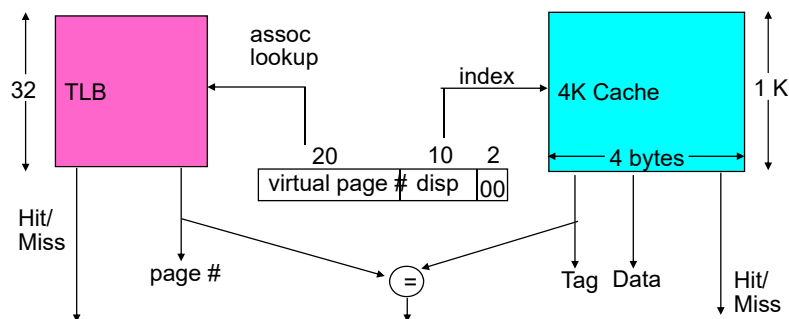
Overlapping TLB & Cache Access (1/2)

- 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, direct-mapped cache:



- This gets much more interesting if you want a larger cache
 - Increase page size (obvious, but not entirely desirable)
 - Increase associativity of cache (thereby decreasing index)
 - Start looking up in multiple chunks of cache, pick when TLB lookup is finished.

Example TLB Sizes: 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)

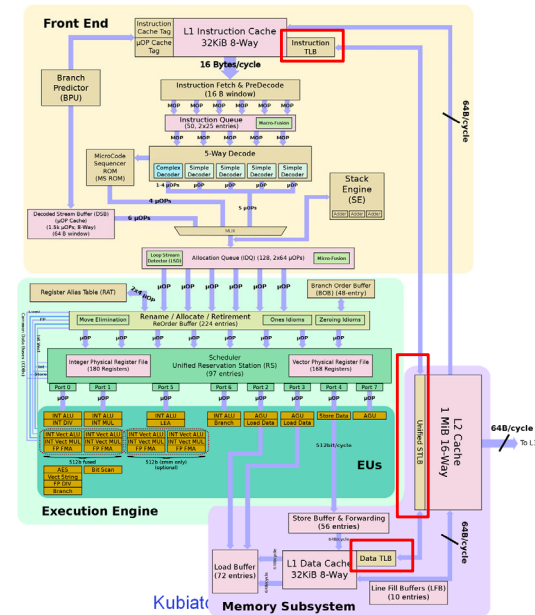
- L1 DTLB
 - 64 entries for 4 K pages and
 - 32 entries for 2/4 M pages,
- L1 ITLB
 - 128 entries for 4 K pages using 4-way associativity and
 - 14 fully associative entries for 2/4 MiB pages
- unified 512-entry L2 TLB for 4 KiB pages, 4-way associative.

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.29

Current Intel x86 (Skylake, Cascade Lake)



3/12/20

Kubiat

Memory Subsystem

Lec 14.30

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:

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.31

What happens to TLB on Context Switch?

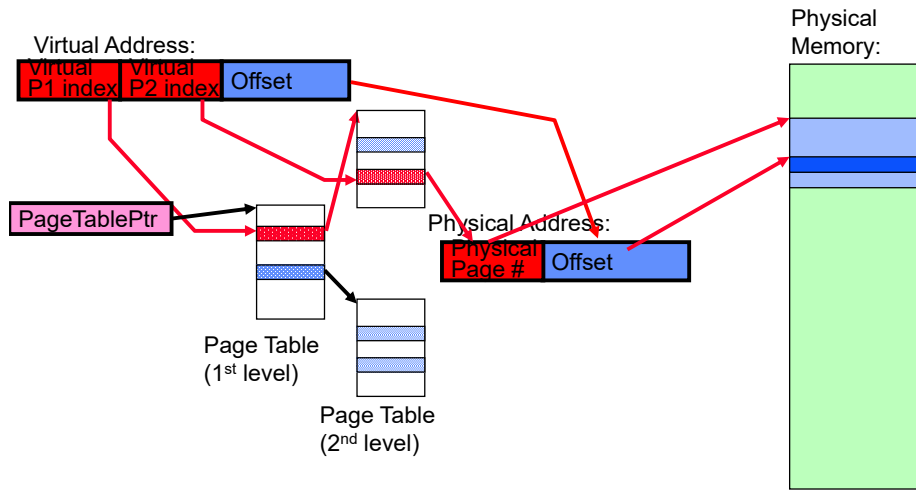
- Need to do something, since TLBs map virtual addresses to physical addresses
 - Address Space just changed, so TLB entries no longer valid!
- Options?
 - Invalidate TLB: simple but might be expensive
 - » What if switching frequently between processes?
 - Include ProcessID in TLB
 - » This is an architectural 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”

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.32

Putting Everything Together: Address Translation

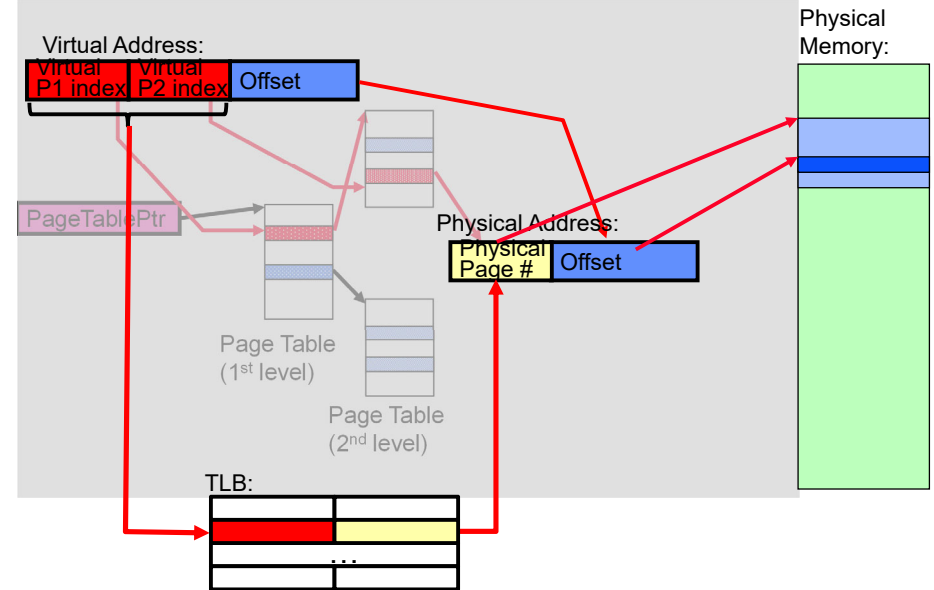


3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.33

Putting Everything Together: TLB

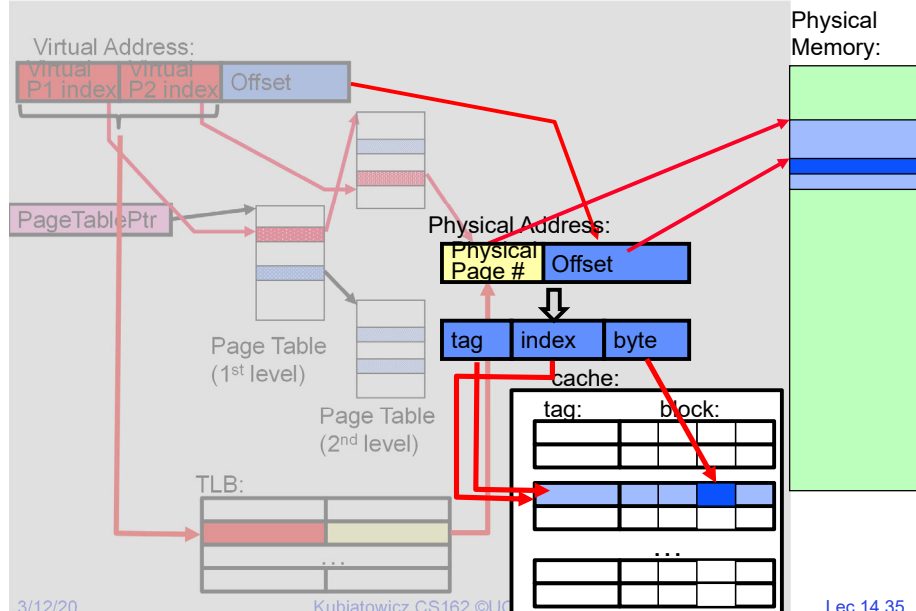


3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.34

Putting Everything Together: Cache

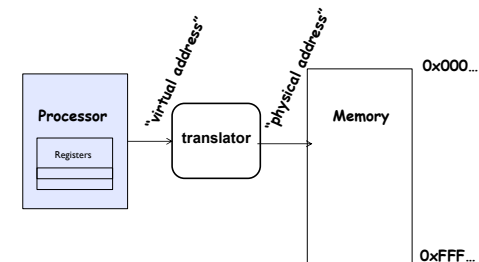


3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.35

Recall: Two Critical Issues in Address Translation



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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.36

Recall: User→Kernel (Exceptions: Traps & Interrupts)

- A system call instruction causes a synchronous exception (or “trap”)
 - In fact, often called a software “trap” instruction
- Other sources of **Synchronous Exceptions (“Trap”)**:
 - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
 - Segmentation Fault (address out of range)
 - Page Fault (for illusion of infinite-sized memory)
- Interrupts are **Asynchronous Exceptions**:
 - Examples: timer, disk ready, network, etc....
 - **Interrupts can be disabled, traps cannot!**
- On system call, exception, or interrupt:
 - Hardware enters kernel mode with interrupts disabled
 - Saves PC, then jumps to appropriate handler in kernel
 - Some processors (e.g. x86) also save registers, changes stack
- Handler does any required state preservation not done by CPU:
 - Might save registers, other CPU state, and switches to kernel stack

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.37

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, ...
- Imprecise \Rightarrow system software has to figure out what is where and put it all back together
- Performance goals may 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**
- Architectural support for OS is hard**
 - Original M68000 had paging, but didn't save fault address properly
 - Original Sun Unix workstation had two fault addressed, running one-cycle apart !#%\$!

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.38

Page Fault is a Synchronous Exception

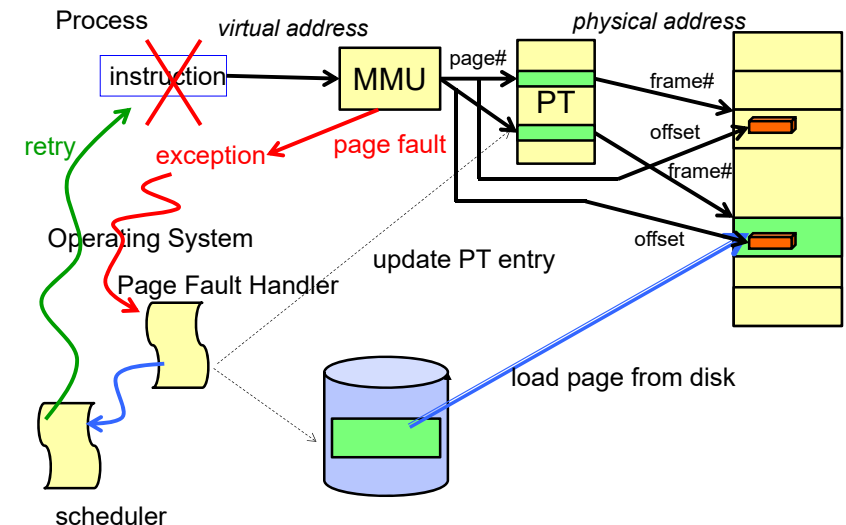
- The Virtual-to-Physical Translation fails
 - PTE marked invalid (at whatever level of page table), Privilege-Level Violation, Access violation
- Causes an Fault / Trap
 - Not an interrupt because synchronous to instruction execution!
 - May occur on instruction fetch or data access
 - Protection violations typically terminate the instruction in a way that is restartable (more later)
- 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 – demand paging
- Protection violations that cannot be resolved \Rightarrow terminate process (possibly “dumping core” image for debugging)
- Fundamental inversion of the hardware / software boundary

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.39

Next Up: What happens when ...



3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.40

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)

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.41

Demand Paging as Caching, ...

- What “block size”? - 1 page (e.g, 4 KB)
- What “organization” ie. direct-mapped, set-associ., fully-associative?
 - Any page in any frame of memory, i.e., fully associative: 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!

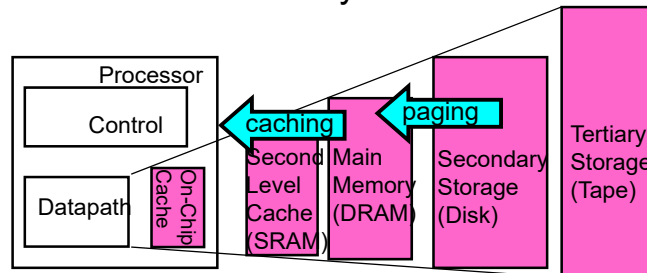
3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.42

Demand Paging

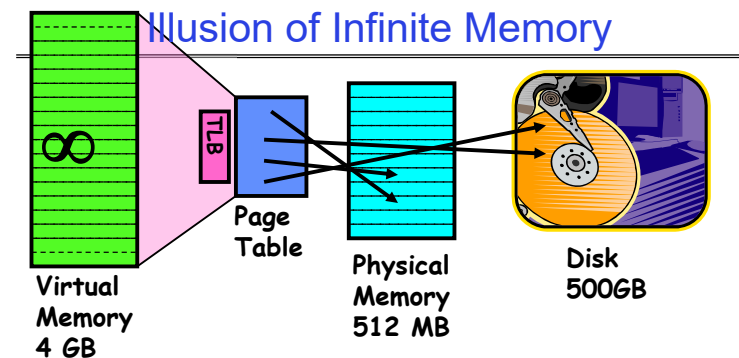
- Modern programs require a lot of physical memory
 - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
 - 90-10 rule: programs spend 90% of their time in 10% of their code
 - Wasteful to require all of user's code to be in memory
- Solution: use main memory as “cache” for disk



3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.43



- 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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.44

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 table (10, 10, 12-bit offset)
 - Intermediate page tables called "Directories"

Page Frame Number (Physical Page Number)	Free (OS)	0	PS	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

PS: Page Size: PS=1 \Rightarrow 4MB page (directory only).

Bottom 22 bits of virtual address serve as offset

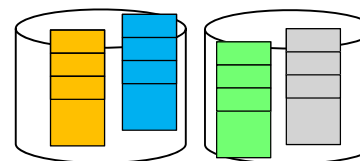
3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.45

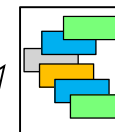
Origins of Paging

Keep most of the address space on disk



Disks provide most of the storage

Actively swap pages to/from



Relatively small memory, for many processes

Keep memory full of the frequently accesses pages

P



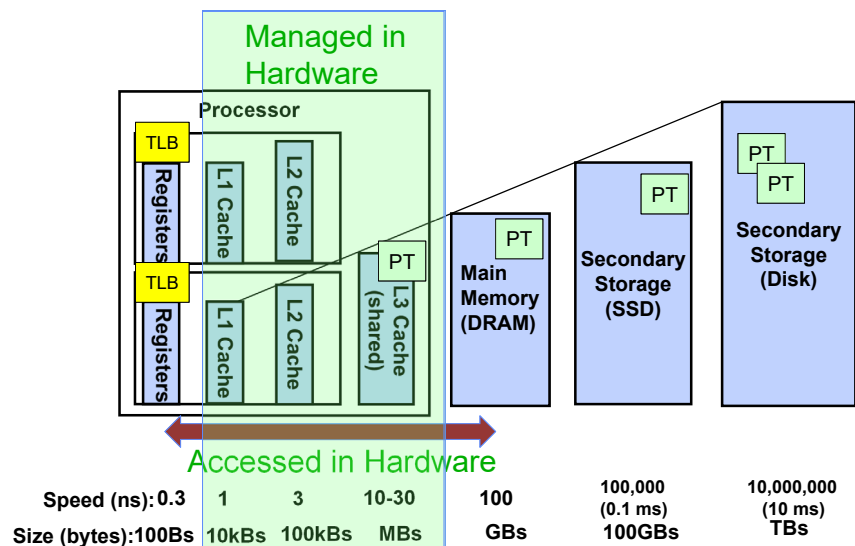
Many clients on dumb terminals running different programs

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.46

Recall: The Memory Hierarchy

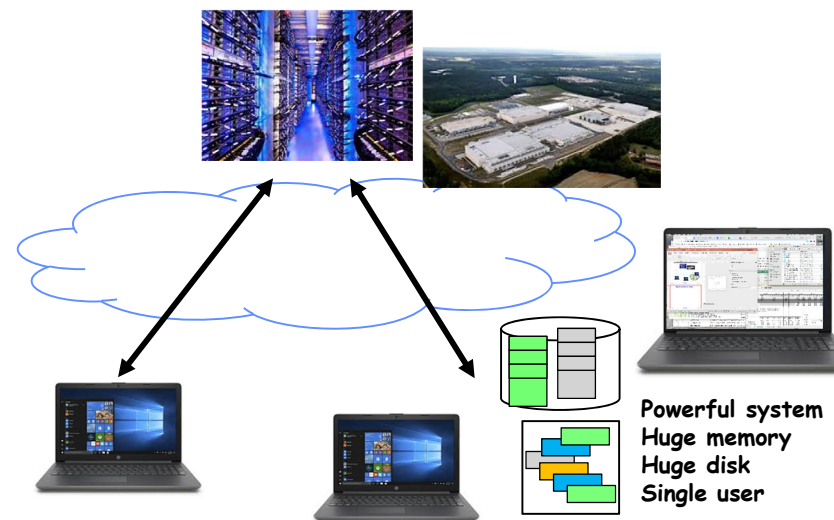


3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.47

Very Different Situation Today



3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.48

A Picture on one machine

```
Processes: 407 total, 2 running, 405 sleeping, 2135 threads
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 vszize, 1372M framework vszize, 68020510(0) swpains, 71200340(0) swapouts.
Networks: packets: 40629441/21G in, 21395374/7747M out.
Disks: 17026780/555G read, 15757470/638G written.
22:10:35
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORTS	MEM	PURG	CHPRS	PCRP	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	DiskMountout	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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.49

Many Uses of "Demand Paging" ...

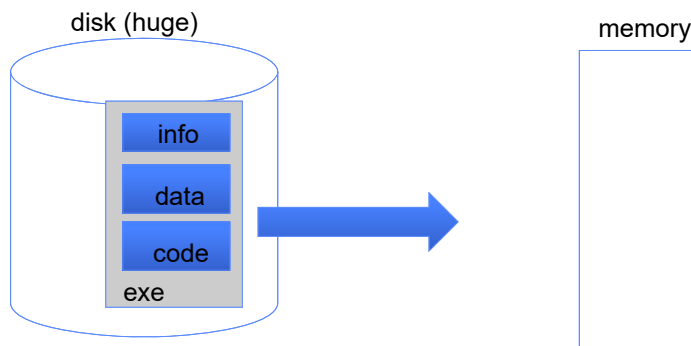
- Extend the stack
 - Allocate a page and zero it
- Extend the heap (sbrk of old, today mmap)
- Process Fork
 - Create a copy of the page table
 - Entries refer to parent pages – NO-WRITE
 - Shared read-only pages remain shared
 - Copy page on write
- Exec
 - Only bring in parts of the binary in active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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.50

Classic: Loading an executable into memory



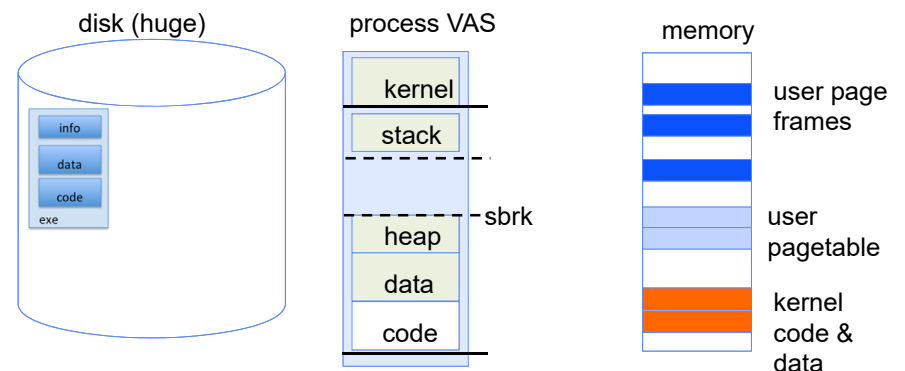
- .exe
 - lives on disk in the file system
 - contains contents of code & data segments, relocation entries and symbols
 - OS loads it into memory, initializes registers (and initial stack pointer)
 - program sets up stack and heap upon initialization: crt0 (C runtime init)

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.51

Create Virtual Address Space of the Process



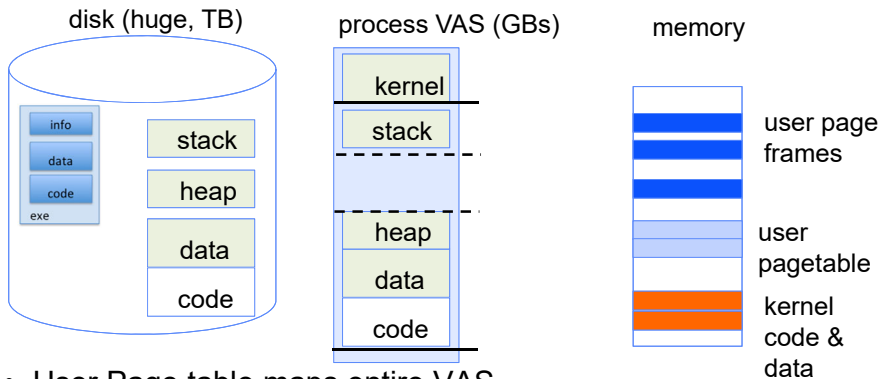
- Utilized pages in the VAS are backed by a page block on disk
 - Called the backing store or swap file
 - Typically in an optimized block store, but can think of it like a file

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.52

Create Virtual Address Space of the Process



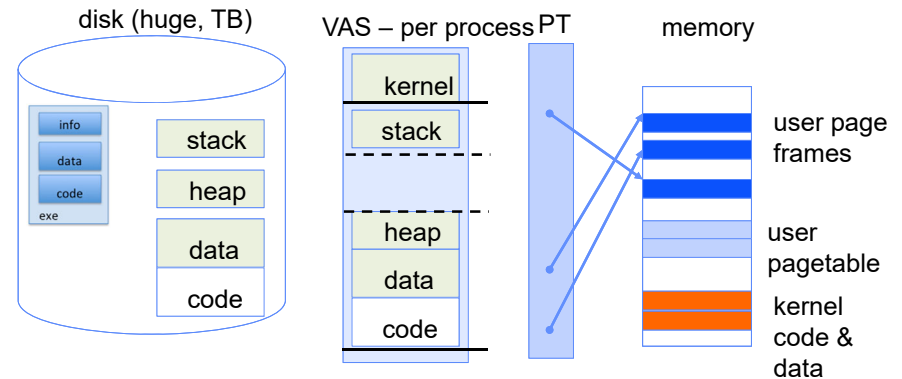
- User Page table maps entire VAS
- All the utilized regions are backed on disk
 - swapped into and out of memory as needed
- For *every* process

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.53

Create Virtual Address Space of the Process



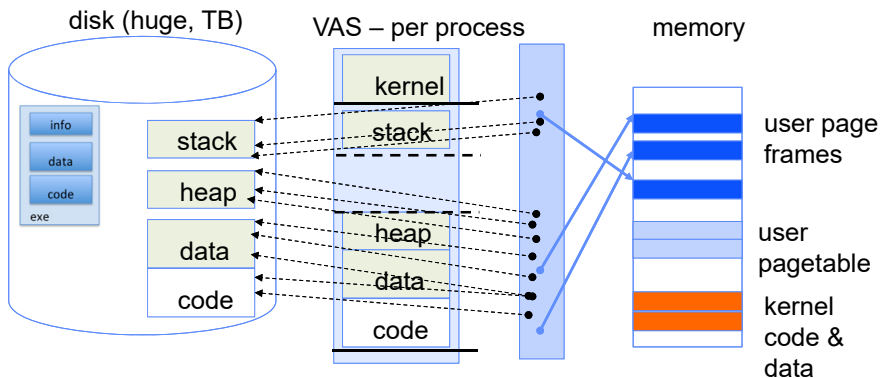
- User Page table maps entire VAS
 - Resident pages to the frame in memory they occupy
 - The portion of it that the HW needs to access must be resident in memory

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.54

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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.55

What Data Structure Maps Non-Resident Pages to Disk?

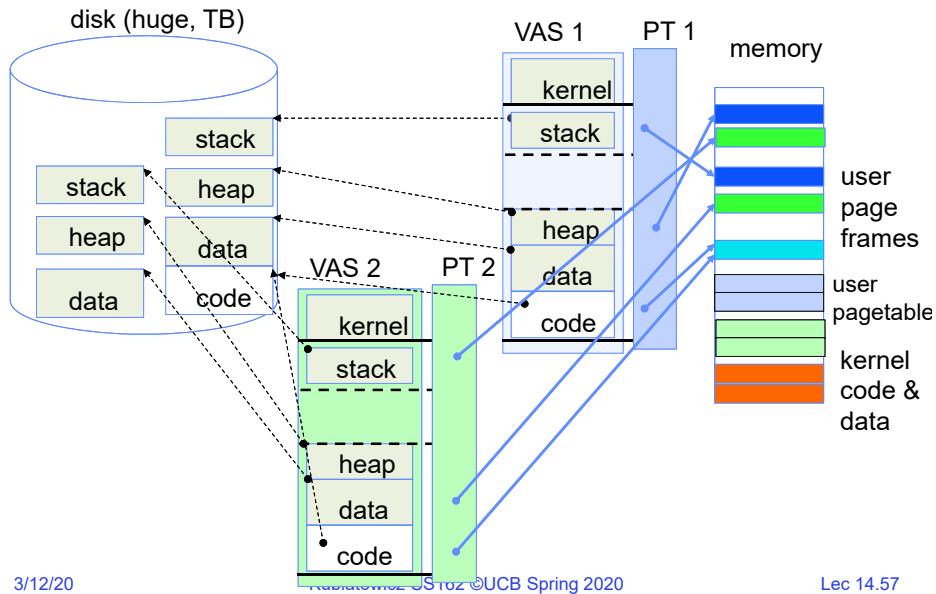
- FindBlock(PID, page#) → disk_block
 - Some OSs utilize spare space in PTE for paged blocks
 - Like the PT, but purely software
- Where to store it?
 - In memory – can be compact representation if swap storage is contiguous on disk
 - Could use hash table (like Inverted PT)
- Usually want backing store 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

3/12/20

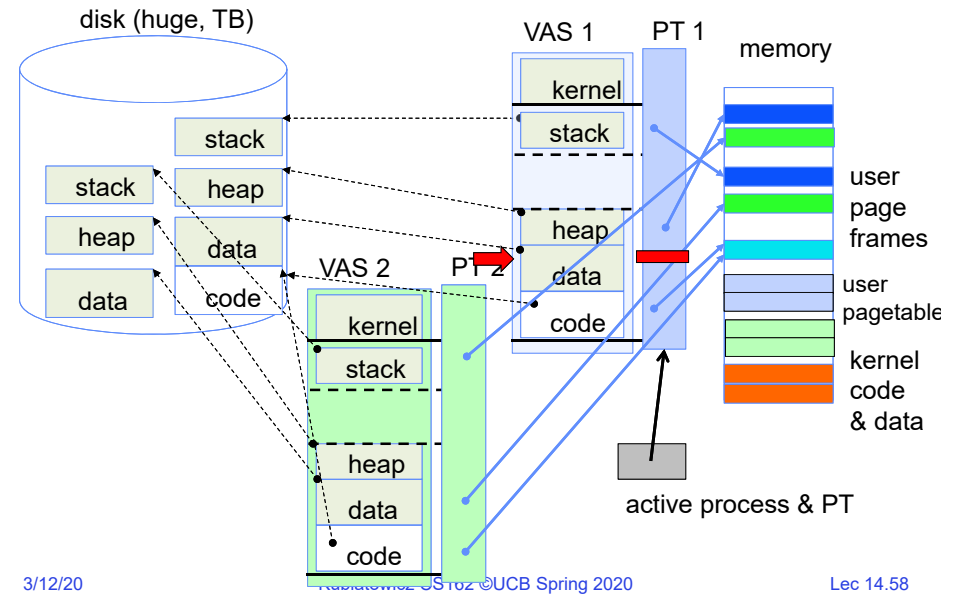
Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.56

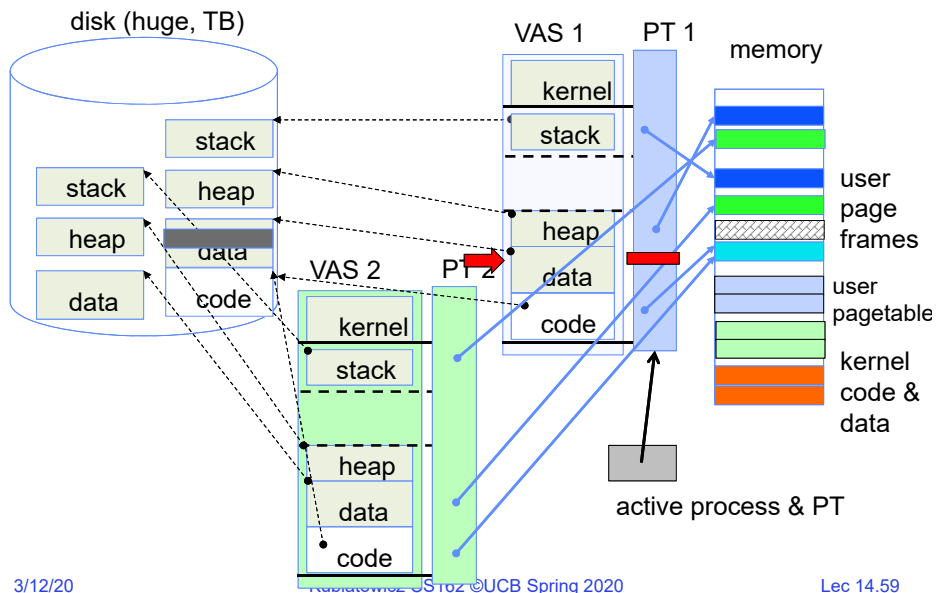
Provide Backing Store for VAS



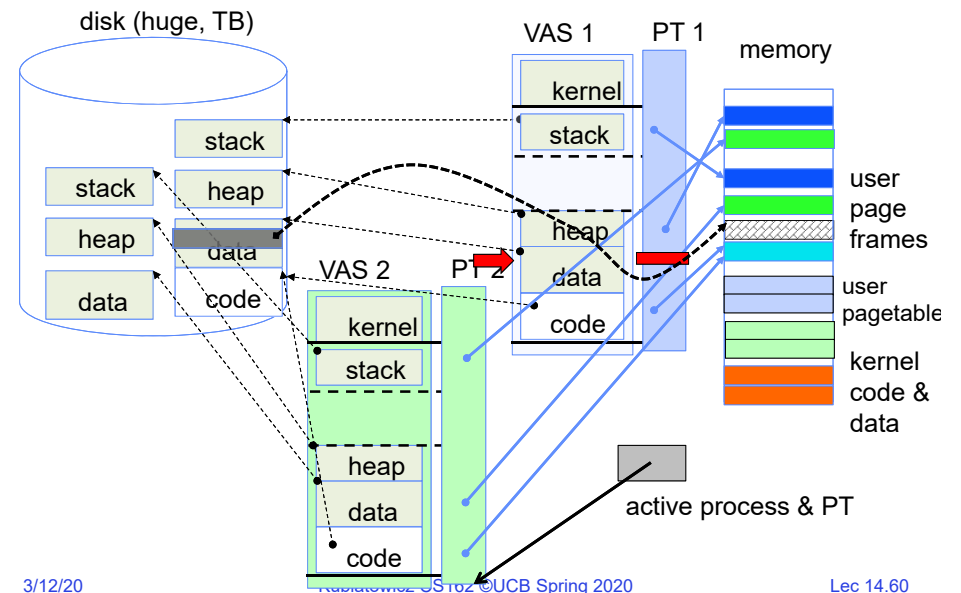
On page Fault ...



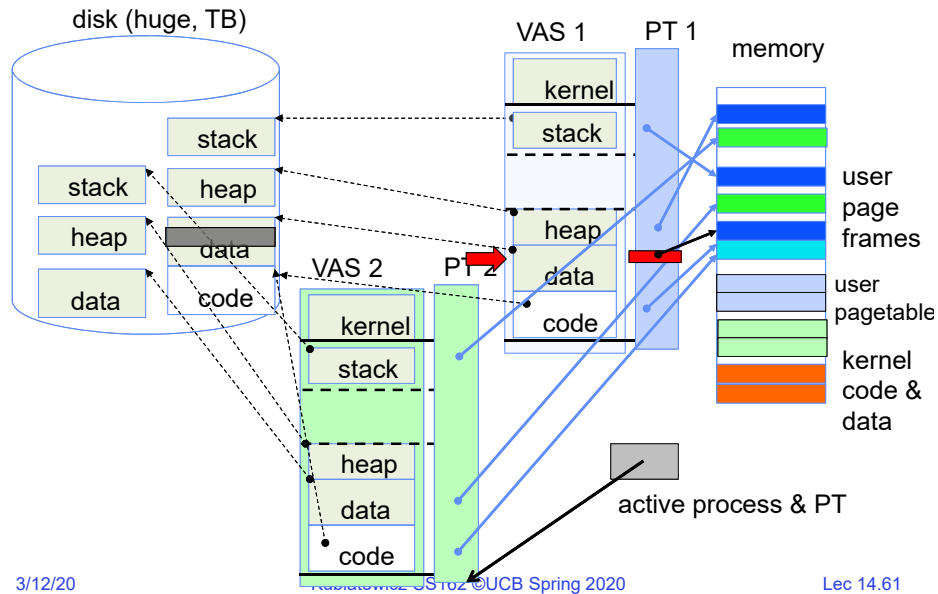
On page Fault ... find & start load



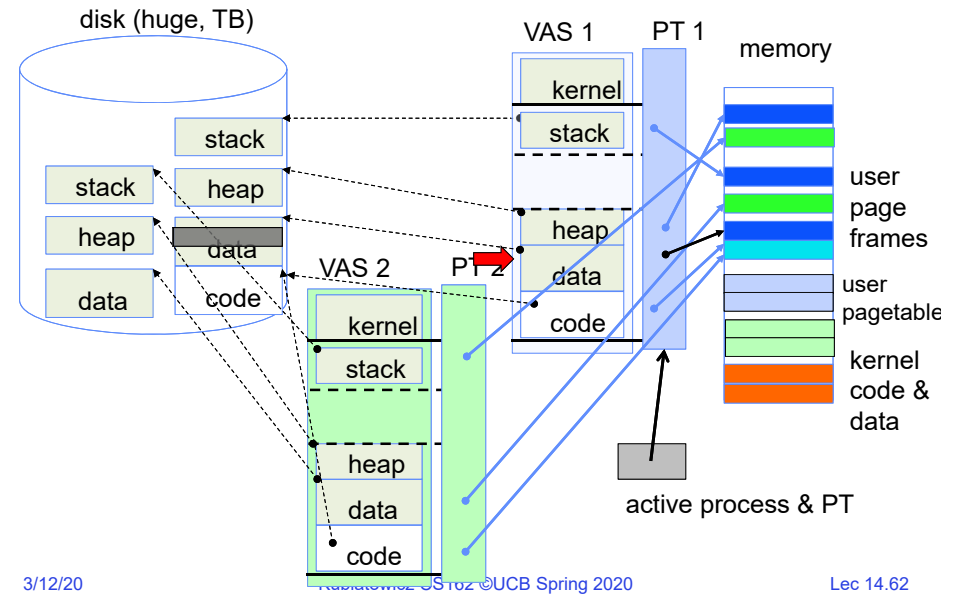
On page Fault ... schedule other P or T



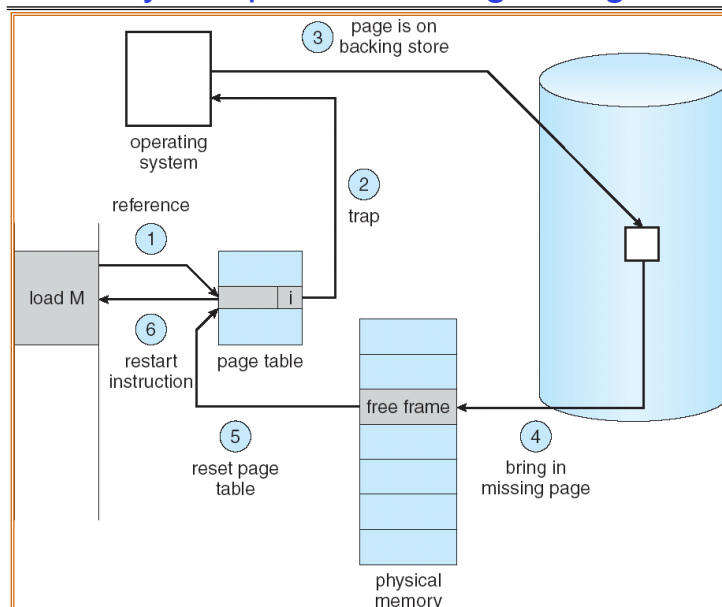
On page Fault ... update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault



Demand Paging 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

Some questions we need to answer!

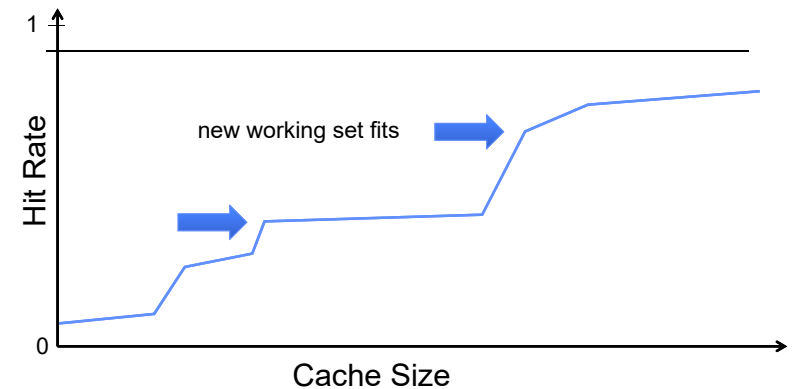
- During a page fault, where does the OS get a free frame?
 - Keeps a free list
 - Unix runs a “reaper” if memory gets too full
 - » Schedule dirty pages to be written back on disk
 - » Zero (clean) pages which haven’t been accessed in a while
 - As a last resort, evict a dirty page first
- How can we 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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.65

Cache Behavior under WS model



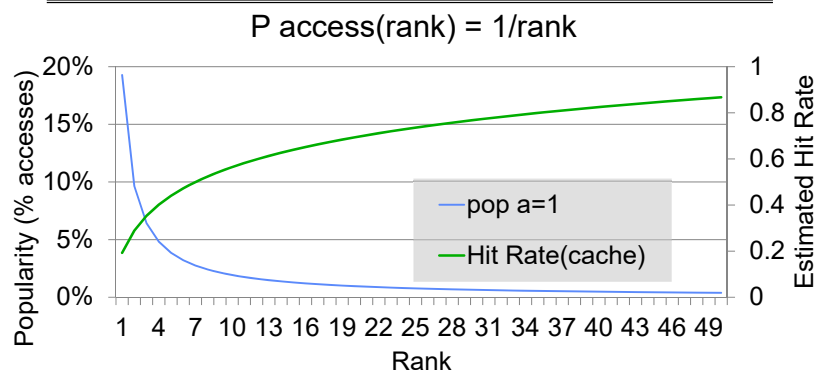
- Amortized by fraction of time the Working Set is active
- Transitions from one WS to the next
- Capacity, Conflict, Compulsory misses
- Applicable to memory caches and pages. Others ?

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.66

Another model of Locality: Zipf



- 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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.67

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:

$$EAT = 200\text{ns} + p \times 8\text{ms}$$

$$= 200\text{ns} + p \times 8,000,000\text{ns}$$
- If one access out of 1,000 causes a page fault, then $EAT = 8.2\text{ }\mu\text{s}$:
 - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
 - $200\text{ns} \times 1.1 < EAT \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400,000!

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.68

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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.69

Page Replacement Policies

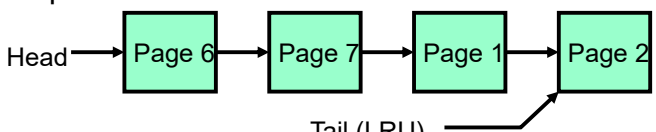
- Why do we care about Replacement Policy?
 - Replacement is an issue with any cache
 - Particularly important with pages
 - » The cost of being wrong is high: must go to disk
 - » Must keep important pages in memory, not toss them out
- **FIFO (First In, First Out)**
 - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
 - Bad – throws out heavily used pages instead of infrequently used
- **RANDOM:**
 - Pick random page for every replacement
 - Typical solution for TLB's. Simple hardware
 - Pretty unpredictable – makes it hard to make real-time guarantees
- **MIN (Minimum):**
 - Replace page that won't be used for the longest time
 - Great (provably optimal), but can't really know future...
 - **But past is a good predictor of the future ...**

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.70

Replacement Policies (Con't)

- **LRU (Least Recently Used):**
 - Replace page that hasn't been used for the longest time
 - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
 - Seems like LRU should be a good approximation to MIN.
 - How to implement LRU? Use a list!
- 
- On each use, remove page from list and place at head
 - LRU page is at tail
- Problems with this scheme for paging?
 - Need to know immediately when each page used so that can change position in list...
 - Many instructions for each hardware access
 - In practice, people **approximate** LRU (more later)

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.71

Example: FIFO (strawman)

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
 - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since need A again right away

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.72

Example: MIN / LRU

- Suppose we have the same reference stream:
– A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

- MIN: 5 faults
– Where will D be brought in? Look for page not referenced farthest in future
- What will LRU do?
– Same decisions as MIN here, but won't always be true!

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.73

Is LRU guaranteed to perform well?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

- Every reference is a page fault!
- Fairly contrived example of working set of N+1 on N frames

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.74

When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

- Every reference is a page fault!
- MIN Does much better:

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

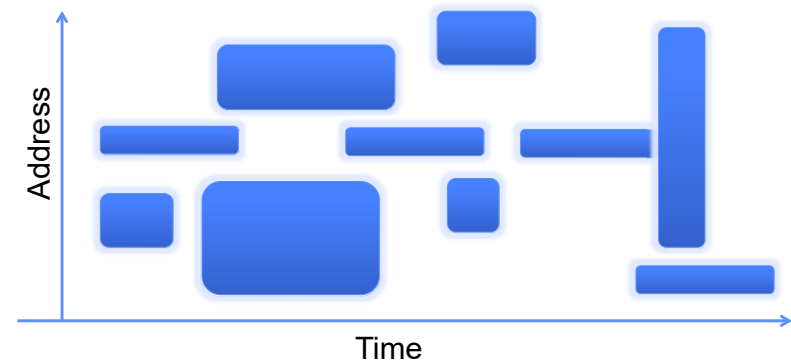
3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.75

Why it works in Practice: Working Set Model

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

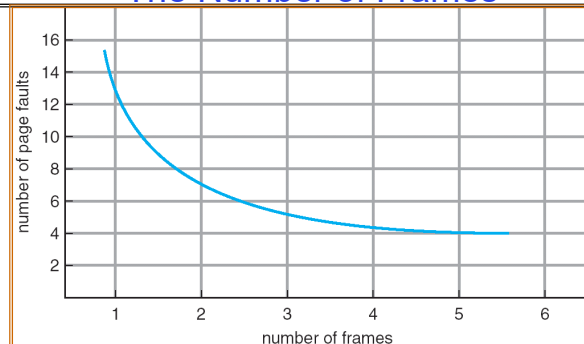


3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.76

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate drops (stack property)
 - Does this always happen?
 - Seems like it should, right?
- No: Bélády's anomaly
 - Certain replacement algorithms (FIFO) don't have this obvious property!

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.77

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
 - Yes for LRU and MIN
 - Not necessarily for FIFO! (Called Bélády's anomaly)

Ref. Page	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref. Page	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
 - With FIFO, contents can be completely different
 - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.78

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

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.79

Summary (2/2)

- “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)
- Precise Exception specifies a single instruction for which:
 - All previous instructions have completed (committed state)
 - No following instructions nor actual instruction have started
- Can manage caches in hardware or software or both
 - Goal is highest hit rate, even if it means more complex cache management

3/12/20

Kubiatowicz CS162 ©UCB Spring 2020

Lec 14.80