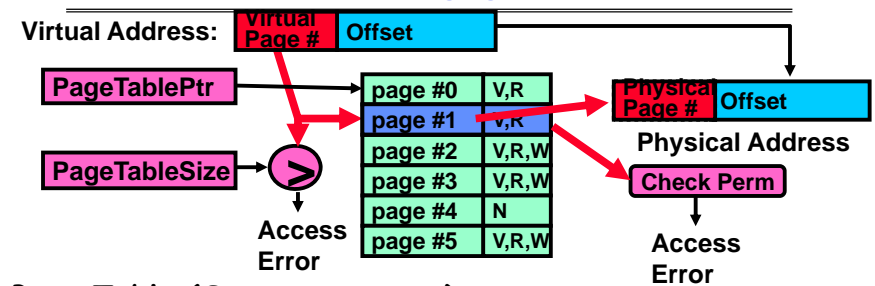


# CS162 Operating Systems and Systems Programming Lecture 13

## Address Translation (Finished), Caching

March 9<sup>th</sup>, 2015  
Prof. John Kubiawicz  
<http://cs162.eecs.Berkeley.edu>

### Recall: Paging



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

3/9/15

Kubiawicz CS162 @UCB Spring 2015

Lec 13.2

### Recall: Simple Page Table Discussion

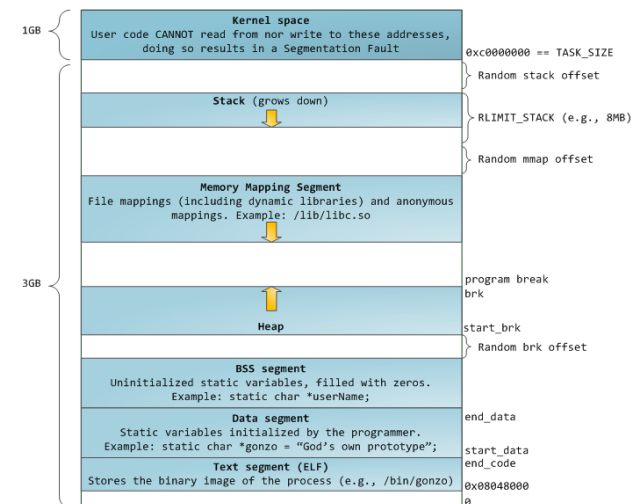
- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at  $(2^{31}-1)$ .
    - » With 1K pages, need 2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

3/9/15

Kubiawicz CS162 @UCB Spring 2015

Lec 13.3

### Memory Layout for Linux 32-bit



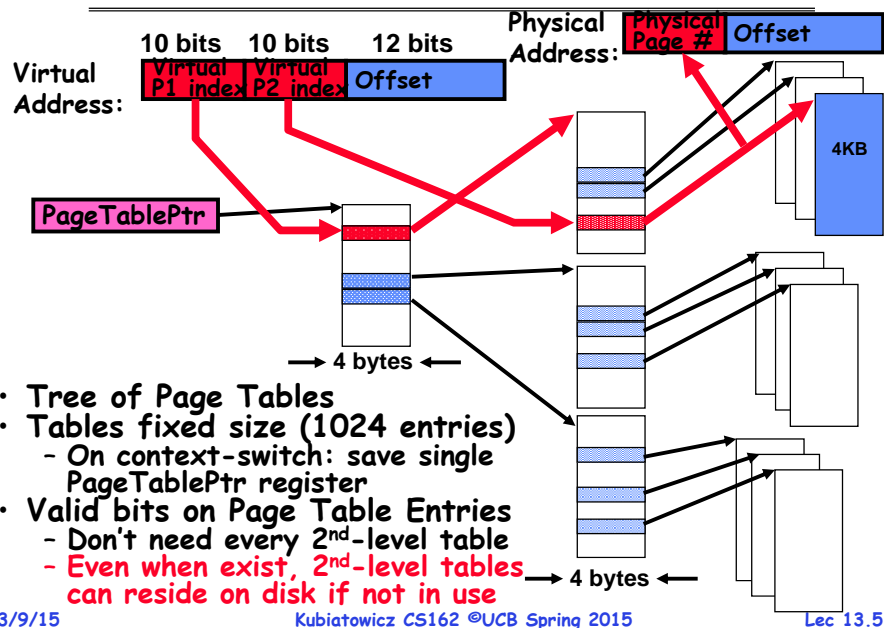
<http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png>

3/9/15

Kubiawicz CS162 @UCB Spring 2015

Lec 13.4

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



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2<sup>nd</sup>-level table
  - Even when exist, 2<sup>nd</sup>-level tables can reside on disk if not in use

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.5

## What is in a Page Table Entry?

- 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

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.6

## Examples of how to use a PTE

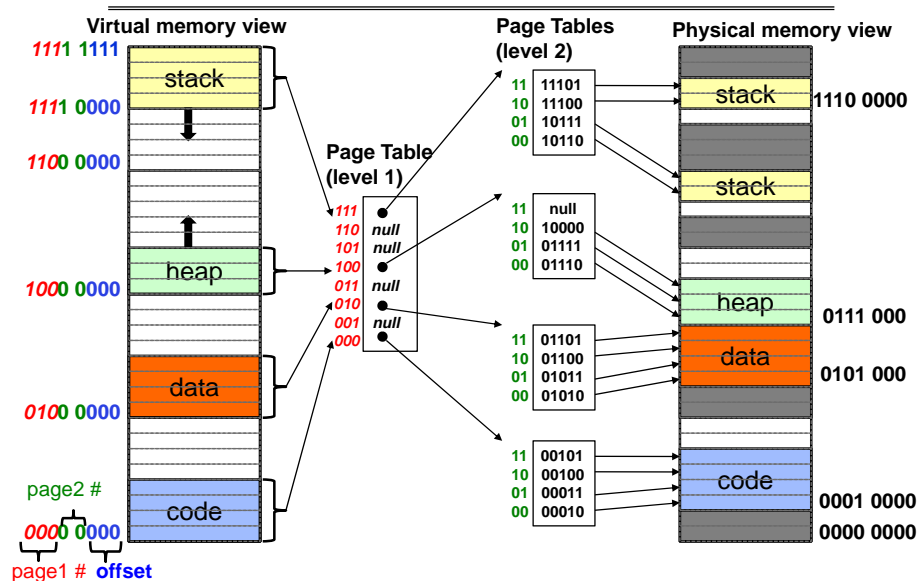
- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory
  - Validity checked first
    - » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - » Address spaces disconnected after child created
  - How to do this cheaply?
    - » Make copy of parent's page tables (point at same memory)
    - » Mark entries in both sets of page tables as read-only
    - » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.7

## Summary: Two-Level Paging

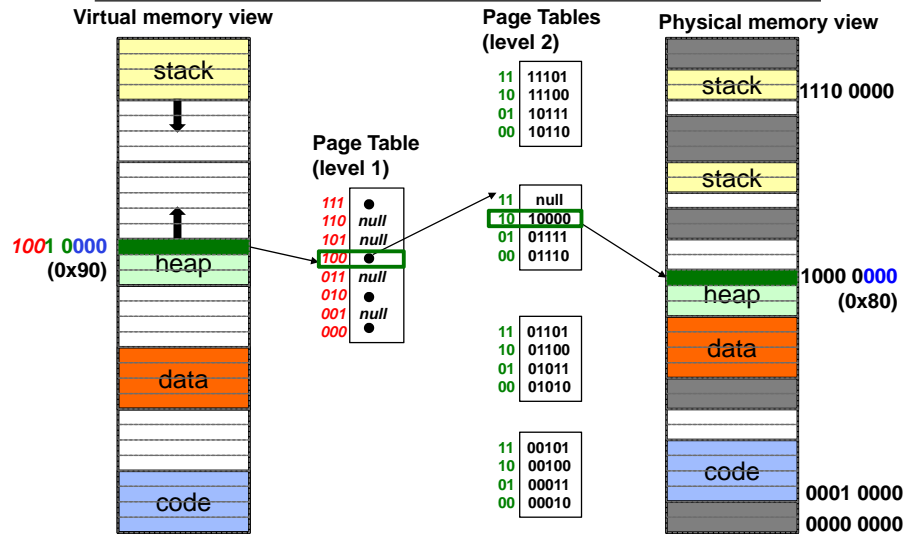


3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.8

## Summary: Two-Level Paging



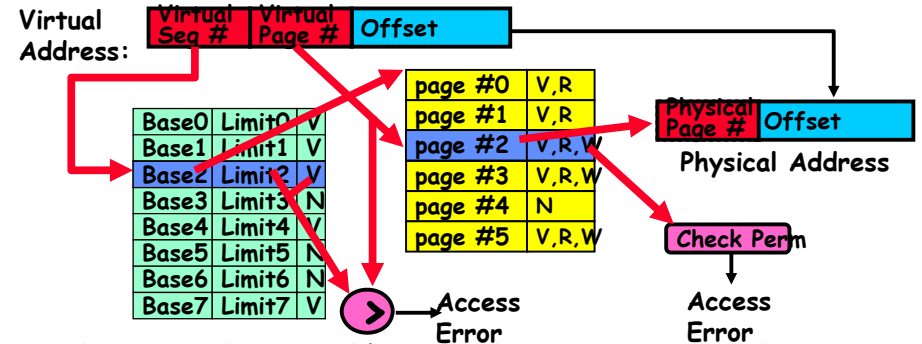
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.9

## Recall: Segments + Pages

- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



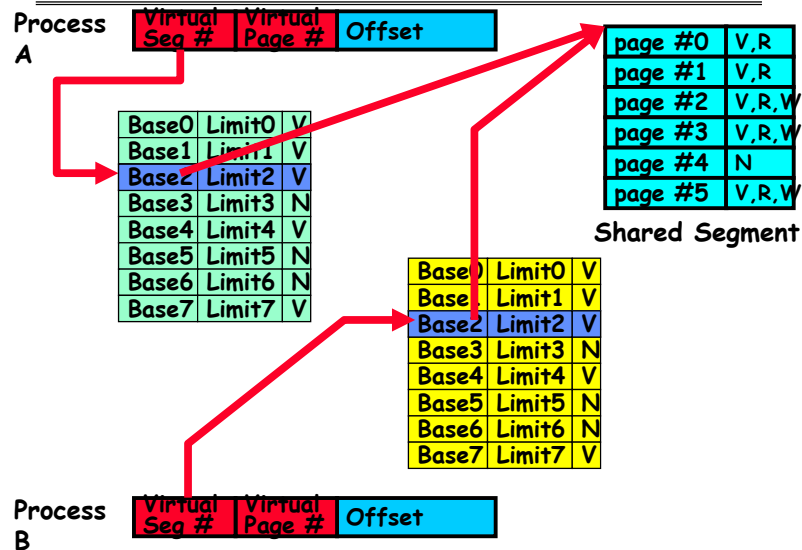
- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.10

## Recall Sharing (Complete Segment)



3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.11

## Multi-level Translation Analysis

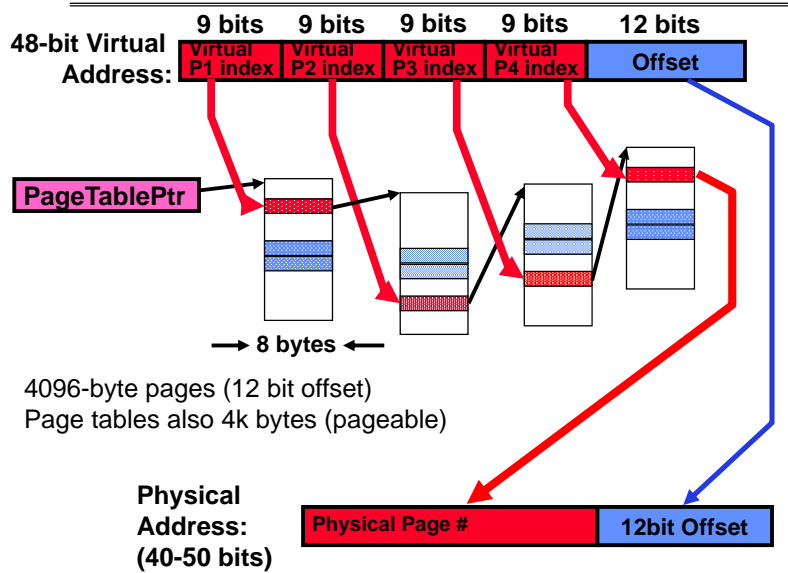
- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K - 16K pages today)
  - Page tables need to be contiguous
    - » However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.12

## X86\_64: Four-level page table!



3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.13

## IA64: 64bit addresses: Six-level page table?!?



No!

Too slow  
Too many almost-empty tables

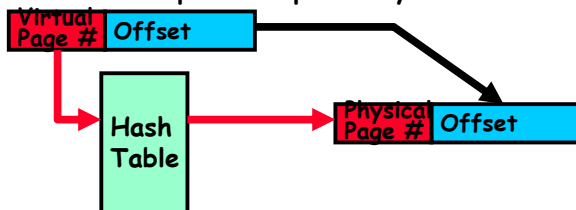
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.14

## Inverted Page Table

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



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

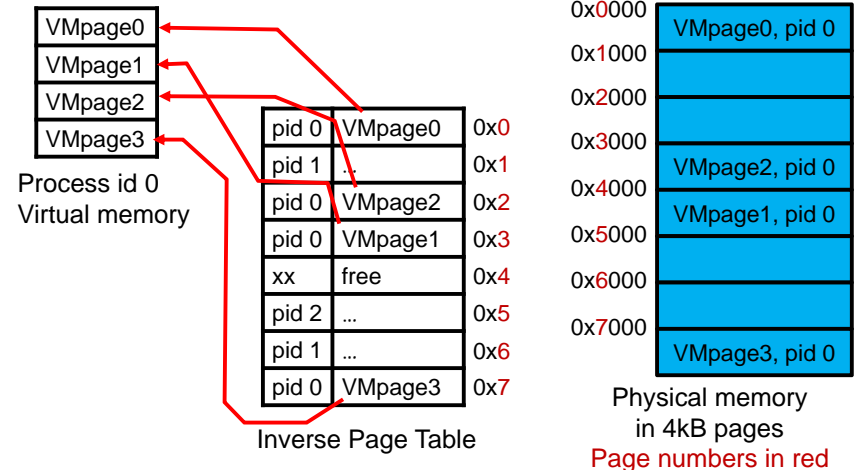
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.15

## IA64: Inverse Page Table (IPT)

Idea: index page table by physical pages instead of VM



3/9/15

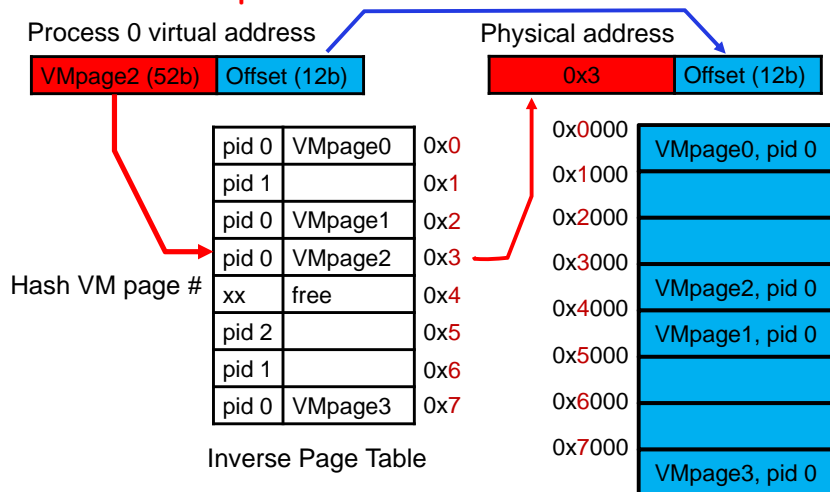
Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.16

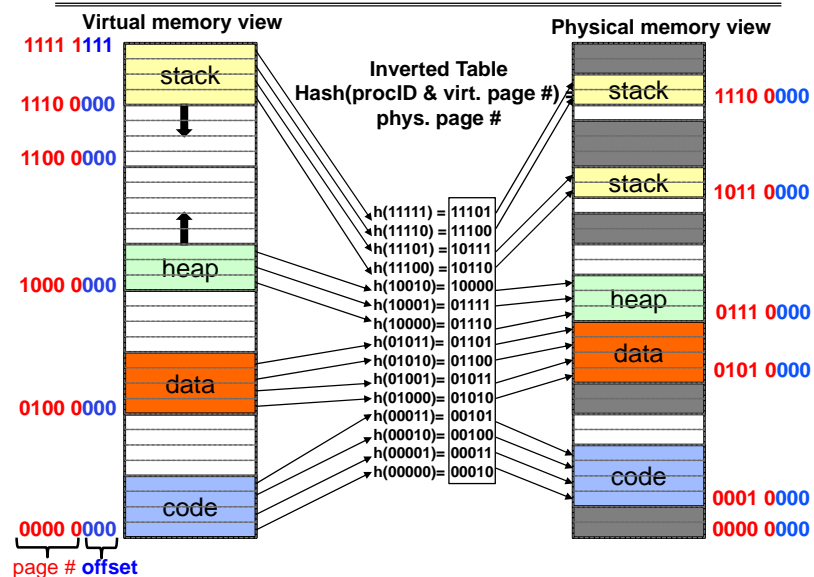
## IPT address translation

- Need an associative map from VM page to IPT address:

- Use a hash map



## Summary: Inverted Table



## Address Translation Comparison

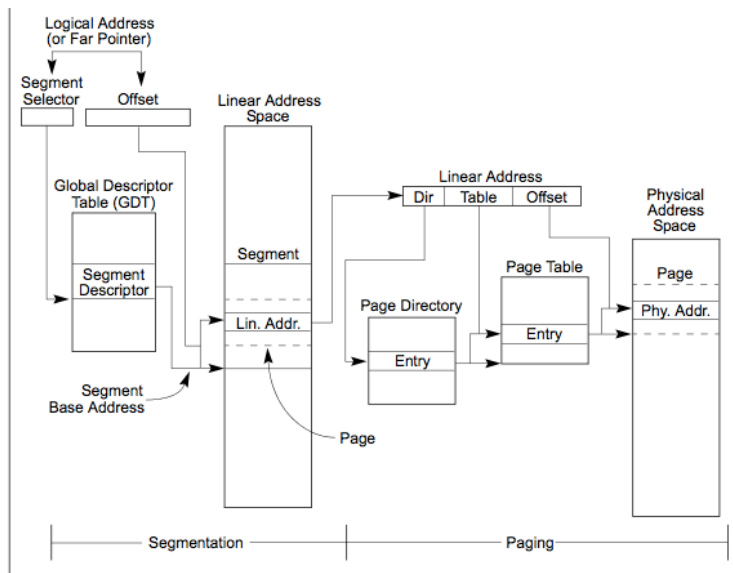
	Advantages	Disadvantages
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory Internal fragmentation
Paged segmentation	Table size ~ # of pages in <b>virtual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

## Administrivia

- Midterm I coming up on Wednesday!
  - March 11<sup>th</sup>. 7:00-10:00PM
  - Rooms: 1 PIMENTEL; 2060 VALLEY LSB
    - » Will be dividing up in advance - watch for Piazza post
  - All topics up to and including next Monday
  - Closed book
  - 1 page hand-written notes both sides



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



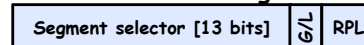
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

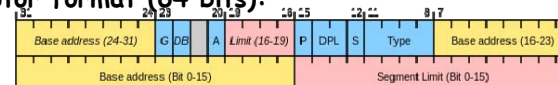
Lec 13.21

## X86 Segment Descriptors (32-bit Protected Mode)

- Segments are either implicit in the instruction (say for code segments) or actually part of the instruction
  - There are 6 registers: SS, CS, DS, ES, FS, GS
- What is in a segment register?
  - A *pointer* to the actual segment description:



- G/L selects between GDT and LDT tables (global vs local descriptor tables)
- Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
  - Includes length of table (for  $< 2^{13}$  entries)
- Descriptor format (64 bits):



- G: Granularity of segment (0: 16bit, 1: 4KiB unit)
- DB: Default operand size (0: 16bit, 1: 32bit)
- A: Freely available for use by software
- P: Segment present
- DPL: Descriptor Privilege Level
- S: System Segment (0: System, 1: code or data)
- Type: Code, Data, Segment

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.22

## Recall: How are segments used?

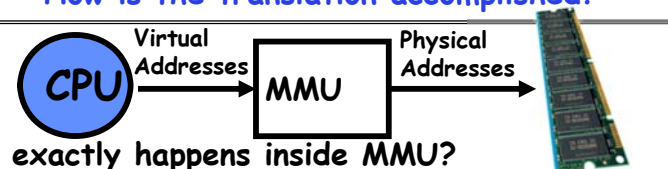
- One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process
- In legacy applications (16-bit mode):
  - Segments provide protection for different components of user programs
  - Separate segments for chunks of code, data, stacks
  - Limited to 64K segments
- Modern use in 32-bit Mode:
  - Segments "flattened", i.e. every segment is 4GB in size
  - One exception: Use of GS (or FS) as a pointer to "Thread Local Storage"
    - » A thread can make accesses to TLS like this:  
`mov eax, gs(0x0)`
- Modern use in 64-bit ("long") mode
  - Most segments (SS, CS, DS, ES) have zero base and no length limits
  - Only FS and GS retain their functionality (for use in TLS)

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.23

## How is the translation accomplished?



- What, exactly happens inside MMU?
- One possibility: Hardware Tree Traversal
  - For each virtual address, takes page table base pointer and traverses the page table in hardware
  - Generates a "Page Fault" if it encounters invalid PTE
    - » Fault handler will decide what to do
    - » More on this next lecture
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
- Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Every translation must invoke Fault!
- In fact, need way to *cache* translations for either case!

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.24

## Recall: Dual-Mode Operation

- Can a process modify its own translation tables?
  - **NO!**
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, **Hardware** provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
  - PL (Privilege Level) from 0 - 3
    - » PLO has full access, PL3 has least
  - Privilege Level set in code segment descriptor (CS)
  - Mirrored "IOPL" bits in condition register gives permission to programs to use the I/O instructions
  - Typical OS kernels on Intel processors only use PLO ("kernel") and PL3 ("user")

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.25

## How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program
- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)

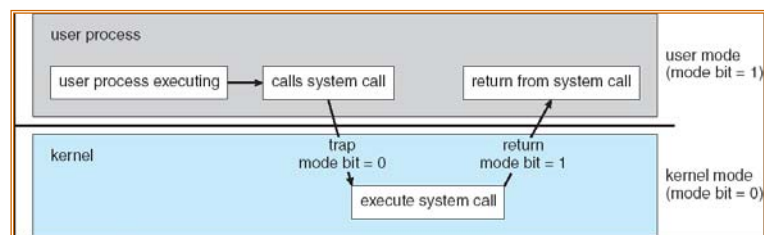
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.26

## Recall: User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- **System call**: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - » No! Only specific ones.
  - System call ID encoded into system call instruction
    - » **Index forces well-defined interface with kernel**

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.27

## User→Kernel (Exceptions: Traps and 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
  - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.28

## Closing thought: Protection without Hardware

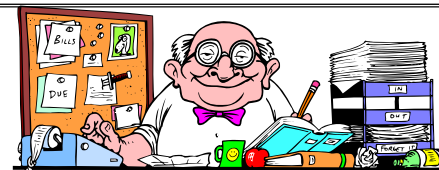
- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can't express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can't step out of bounds
    - Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
    - Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

3/9/15

Kubiatowicz CS162 @UCB Spring 2015

Lec 13.29

## Caching Concept



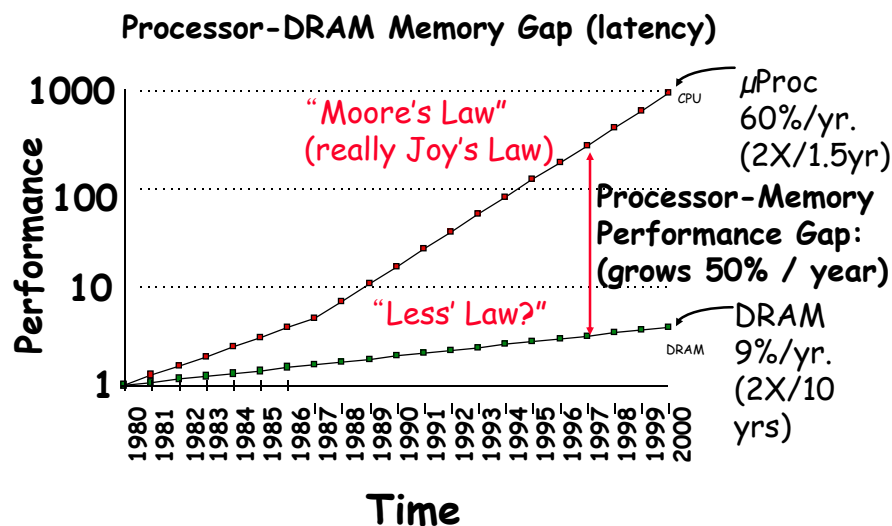
- Cache:** a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many of the techniques that are used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc...
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time =  $(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})$

3/9/15

Kubiatowicz CS162 @UCB Spring 2015

Lec 13.30

## Why Bother with Caching?

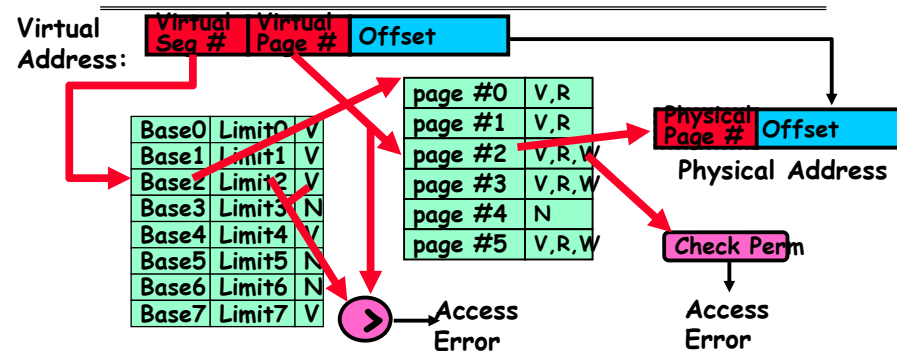


3/9/15

Kubiatowicz CS162 @UCB Spring 2015

Lec 13.31

## Another Major Reason to Deal with Caching



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

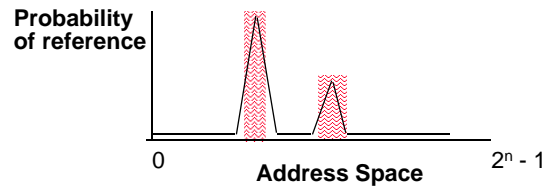
3/9/15

Kubiatowicz CS162 @UCB Spring 2015

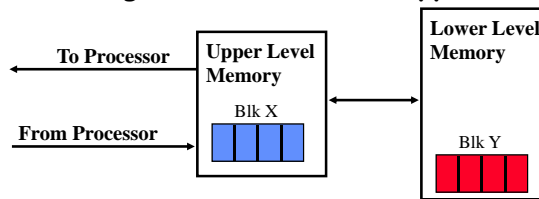
Lec 13.32



## Why Does Caching Help? Locality!



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



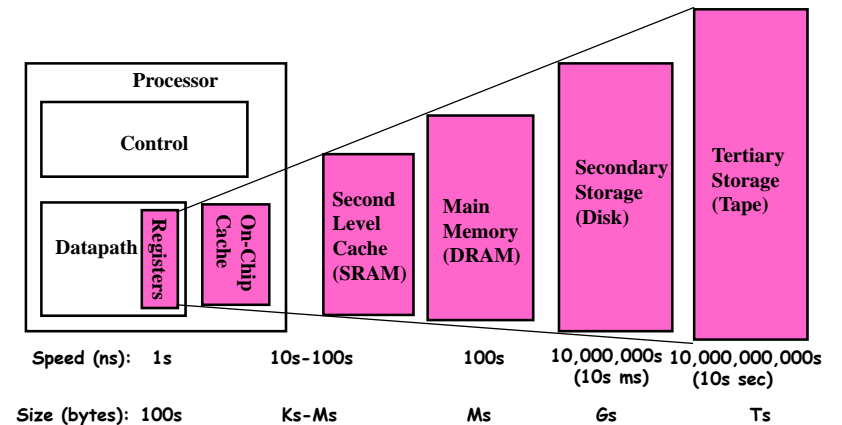
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.33

## Memory Hierarchy of a Modern Computer System

- 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



3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.34

## A Summary on Sources of Cache Misses

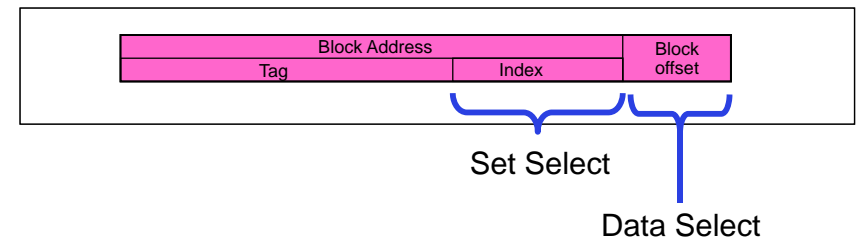
- **Compulsory** (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- **Capacity:**
  - Cache cannot contain all blocks 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

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.35

## 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

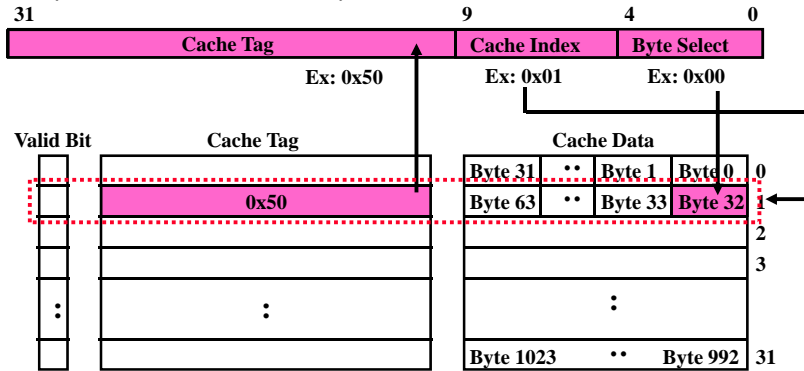
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.36

## 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



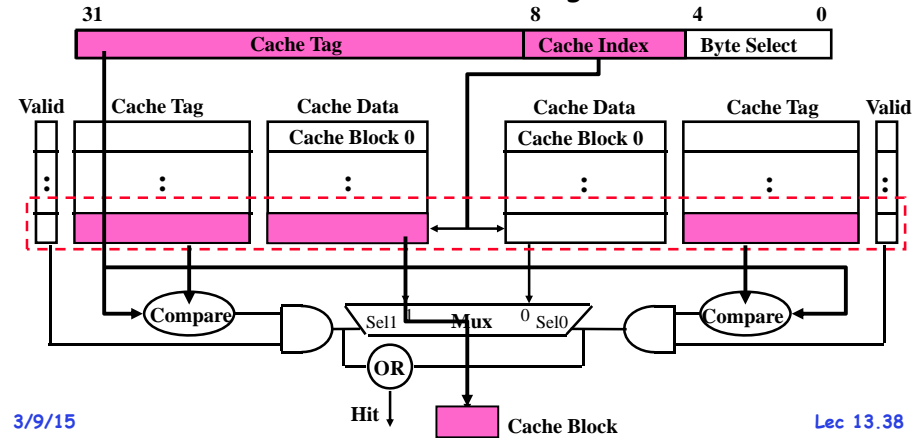
3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.37

## 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

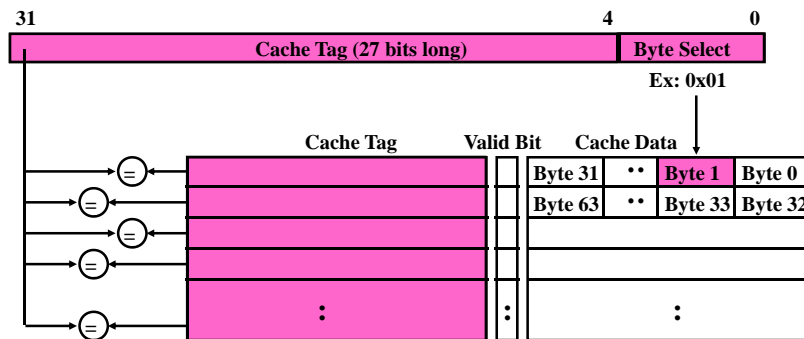


3/9/15

Lec 13.38

## 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



3/9/15

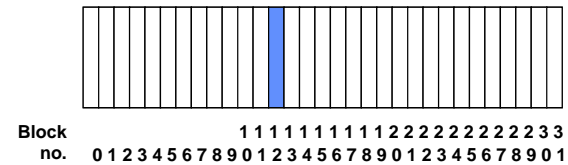
Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.39

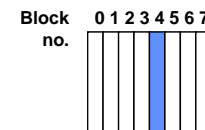
## Where does a Block Get Placed in a Cache?

- **Example: Block 12 placed in 8 block cache**

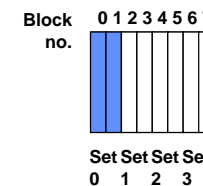
32-Block Address Space:



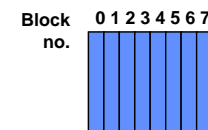
**Direct mapped:**  
block 12 can go  
only into block 4  
(12 mod 8)



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



**Fully associative:**  
block 12 can go  
anywhere



3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.40

## 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)

Size	2-way		4-way		8-way	
	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%

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.41

## Review: What happens on a write?

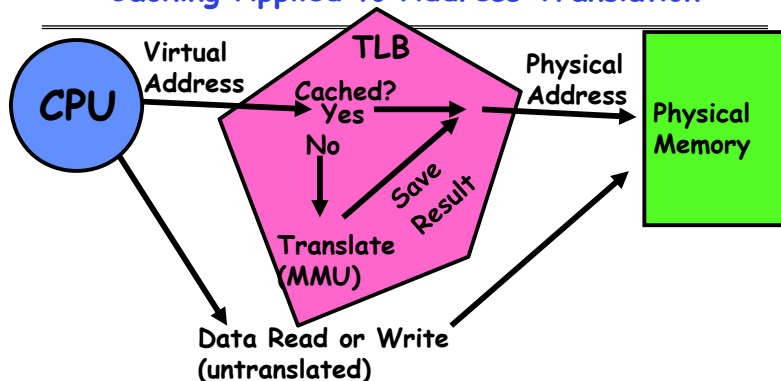
- **Write through:** The information is written to both the block in the cache and to the block in the lower-level memory
- **Write back:** The information is written only to the block in the cache.
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean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

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.42

## 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

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.43

## 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

3/9/15

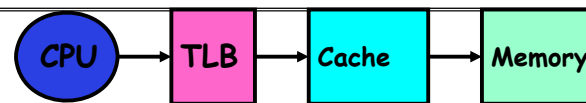
Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.44

## 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!

## 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

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

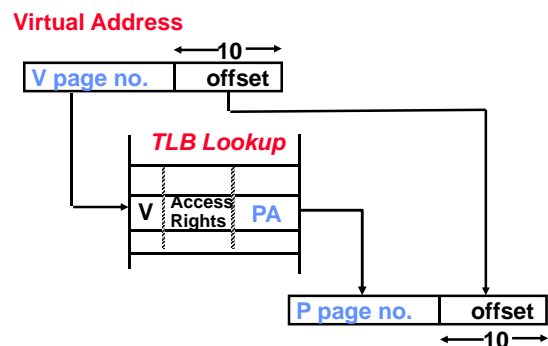
ASID	V. Page Number	Offset
6	20	12

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

## Reducing translation time further

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

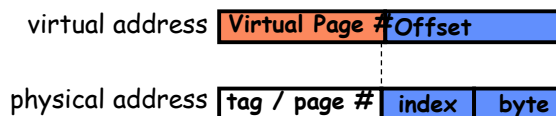


- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

## Overlapping TLB & Cache Access (1/2)

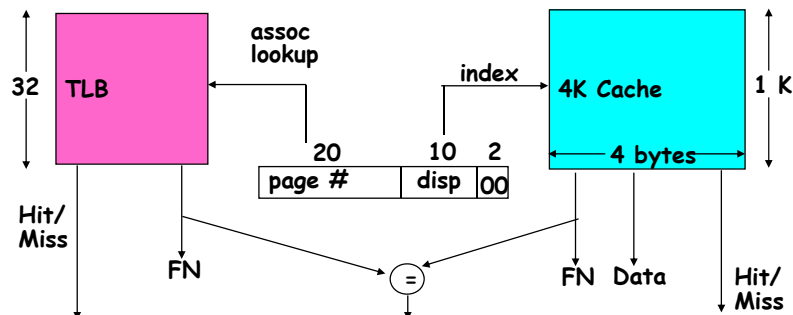
- Main idea:

- 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



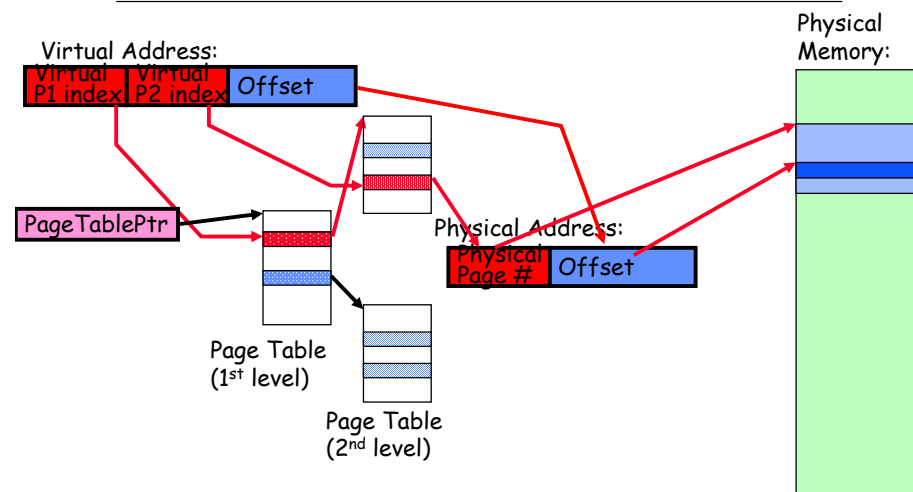
## Overlapping TLB & Cache Access

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



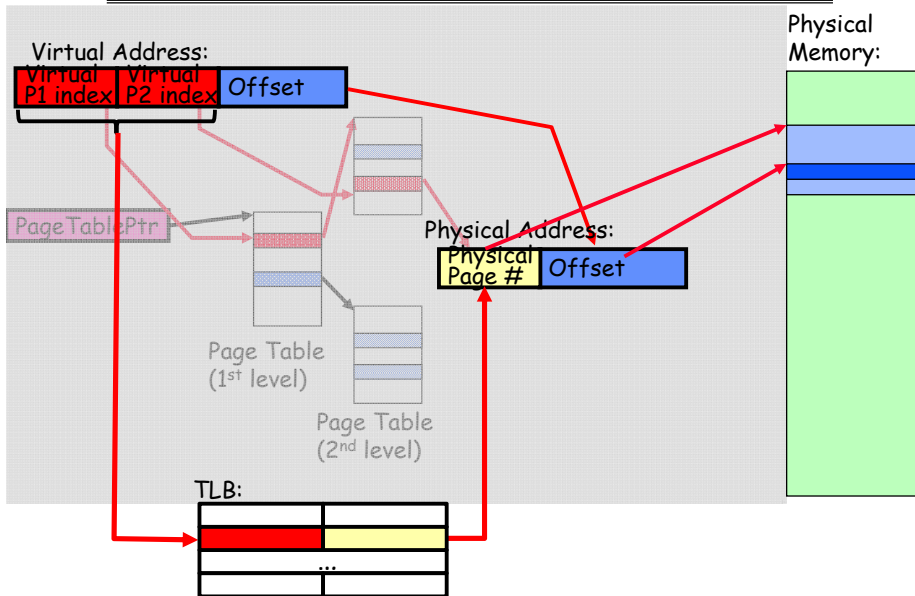
- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

## Putting Everything Together: Address Translation





## Putting Everything Together: TLB

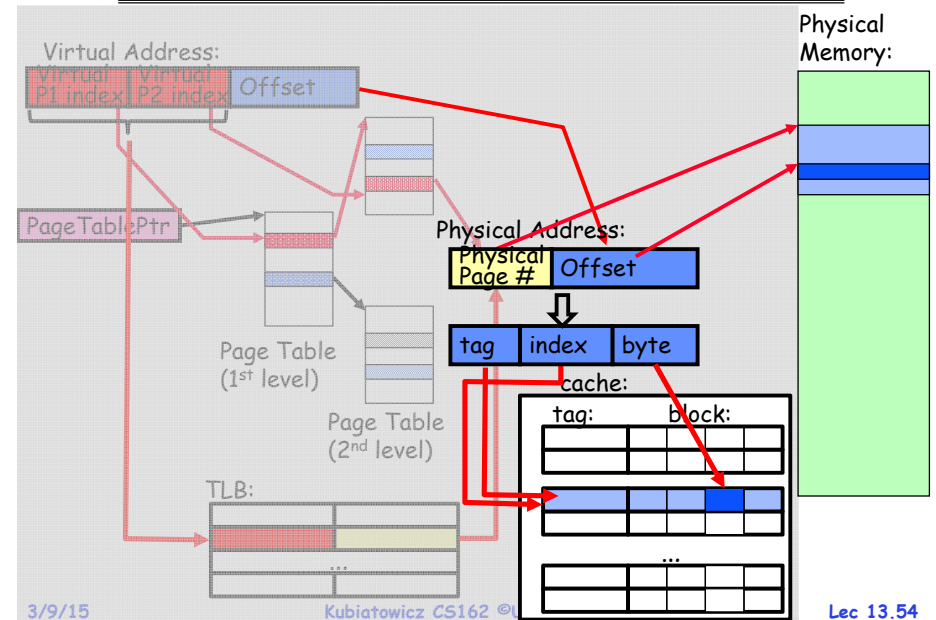


3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.53

## Putting Everything Together: Cache

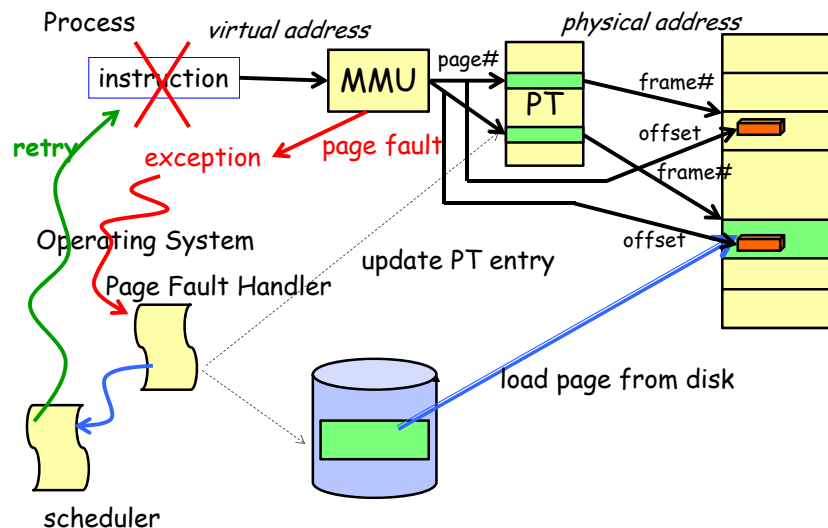


3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.54

## Next Up: What happens when ...



3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.55

## Summary (1/3)

- **Page Tables**
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- **Multi-Level Tables**
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- **Inverted page table**
  - Size of page table related to physical memory size
- **PTE: Page Table Entries**
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)

3/9/15

Kubiatowicz CS162 ©UCB Spring 2015

Lec 13.56

## Summary (2/3)

- **The Principle of Locality:**
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - » **Temporal Locality:** Locality in Time
    - » **Spatial Locality:** Locality in Space
- **Three (+1) Major Categories of Cache Misses:**
  - **Compulsory Misses:** sad facts of life. Example: cold start misses.
  - **Conflict Misses:** increase cache size and/or associativity
  - **Capacity Misses:** increase cache size
  - **Coherence Misses:** Caused by external processors or I/O devices
- **Cache Organizations:**
  - **Direct Mapped:** single block per set
  - **Set associative:** more than one block per set
  - **Fully associative:** all entries equivalent

## Summary (3/3): Translation Caching (TLB)

- **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