Computer Science 146
Computer Architecture

Fall 2019

Harvard University

Instructor: Prof. David Brooks
dbrooks@eecs.harvard.edu

Lecture 18: Virtual Memory

Lecture Outline

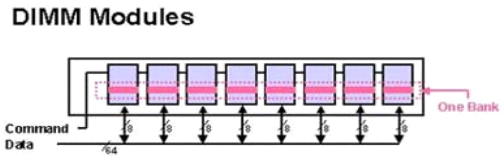
- Review of Main Memory
- Virtual Memory

Simple Interleaving

Cycle	Addr	Bank0	Bank1	Bank2	Bank3	steady
1	12	A	A	A	A	
2		A	A	A	A	
3		T/B	B	B	B	*
4		B	T/B	B	B	*
5				T		*
6					T	*

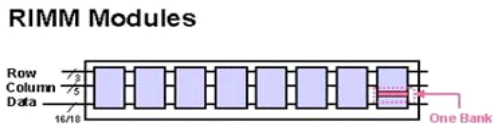
- 4-word access = 6-cycles
- 4-word cycle = 4-cycles
  - Can start a new access in cycle 5
  - Overlap access with transfer (and still use a 32-bit bus!)

Independent Memory Banks



- DIMM (Dual-Inline Memory Module) Configuration
- 1 Rank of devices responds to each access
  - All devices respond similarly
- Single-Sided DIMM
  - 4 banks per device => DIMM has 4 banks
- 512MB DIMM = 8x64Mx8, 4 Banks

Independent Memory Banks



- RAMBUS modules
- 1-32 devices per RIMM module
  - 1 device responds to each access
  - Single-device upgrade granularity
  - Module bandwidth same as device bandwidth
- Devices are independent
  - 8 device RIMM, 16 banks each => RIMM has 128 banks

Independent Memory Banks

- Standard PC Upgrade Path
  - Traditional DIMMS => 8 devices at a time with 8-bit chips
  - Rambus RIMMs => One at a time
- PlayStation 2
- Rambus: 400MHz, 16-bits per channel, 2-bits per clock
  - 1.6GB/sec per channel (only 1 chip needed)
  - 2 Rambus Channels in Parallel, 3.2GB/sec memory bandwidth
- Traditional:PC100 SDRAM: 100MHz, 1-bit per clock
  - Would need 32 chips to achieve 3.2GB/sec bandwidth

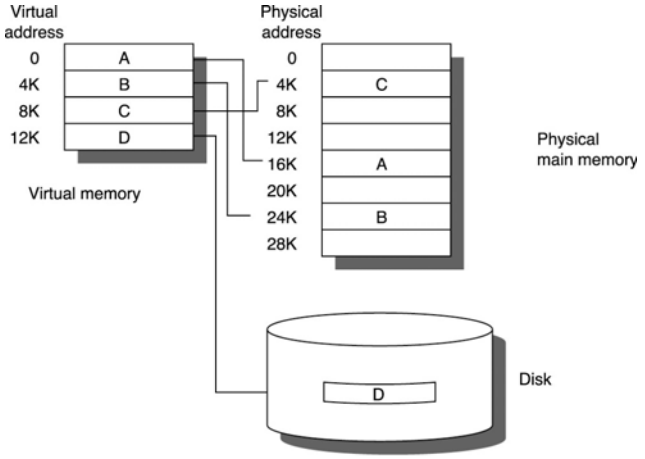
Virtual Memory

- Point we have been avoiding
  - Addresses generated by program are not the addresses that we use to access the memory (physical memory)
  - Why?

Virtual Memory: Motivation

- Original Motivation: Allow main memory to "act as a cache" for secondary storage (disks)
  - Physical memory expensive and not very dense (too small)
  - Programmers wrote "overlays" to load memory from disk
  - Programming nightmare, incompatible code across products
- Current Motivation: Use indirection of VM as a feature
  - Physical memories are quite large
  - Multiprogramming, sharing, relocation, protection
  - Fast program startup
  - Memory mapped files, networks

Virtual vs. Physical Memory



Virtual Memory: Cache Analogy

- Cache blocks/lines are called *pages or segments*
- Cache misses are *page faults or address faults*
- Processes use *virtual addresses (VA)*
- Physical memory uses *physical addresses (PA)*
- Addresses divided into page offset, page number
  - Virtual: Virtual Page Number (VPN)
  - Physical: Page Page Number (PPN)
- *Addresses translation*: system maps VA to PA
  - E.g. 4KB pages, 32-bit machine, 64MB physical memory
  - 32-bit VA, 26-bit PA ($\log_2$64MB), 12-bit page offset ($\log_2$4KB)

Virtual Memory: Cache Analogy

Parameter	First-Level Cache	Virtual Memory
Block (page) Size	16-128 Bytes	4KB – 64KB
Hit Time	1-3 clock cycles	50-150 clock cycles
Miss Penalty	8-150 clock cycles	1M-10M clock cycles
(access time)	(6-130 clock cycles)	(.8M – 8M clock cycles)
(transfer time)	(2-20 clock cycles)	(.2M – 2M clock cycles)
Miss Rate	0.1-10%	0.00001 –0.001%
Address Mapping	25-45bit PA to 14-20bit CacheAd	32-64 bit VA to 25-45 bit PA
Replacement Policy	Hardware Replacement	Software Replacement
Total Size	Independent of Address Space	Processor Address Space
Backing Store	Level 2 Cache	Physical Disk

System maps VA to PA

- Virtual Page Number (VPN) => Physical: Page Page Number (PPN)
- OS/Hardware perform the mapping, *not* processes
- Same VPNs in different processes have different PPNs
  - Protection: processes cannot use each other's PA
  - Programming Simplicity: Each process thinks its alone
  - Relocation: Program can be run anywhere in memory
    - Doesn't have to be physically contiguous
    - Can be paged out, paged back to a different physical location

Virtual Memory: 4 Cache Questions

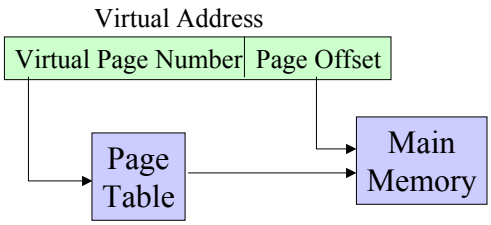
- Same four questions as caches
  - Page Placement: fully associative
    - Why?
  - Page Identification: address translation
    - Indirection through one or two page tables
  - Page Replacement: Sophisticated LRU + Working set
    - Why?
  - Write Strategy: Always write-back + write allocate
    - Why?

Why?

- Backing store to main memory is disk
  - Memory is 50-100x slower than processor
  - Disk is 20-100 *thousand* times slower than memory
    - Disk is 1 to 10 *Million* times slower than processor
- VA miss (page fault) is expensive
  - Minimize at all costs
  - Fully associative + Software Replacement reduce miss rate
  - Write-back reduces disk traffic
  - Large page sizes (4KB – 16KB) amortize reads

Page ID: Address Translation

- OS performs address translation using page table
  - Each process has its own page table
    - OS knows address of each process's page table
  - Page table is an array of Page Table Entries
    - One entry for each VPN of each process, indexed by VPN
  - Each PTE contains
    - Phys. Page Number
    - Permissions
    - Dirty bit
    - LRU
    - ~4bytes total

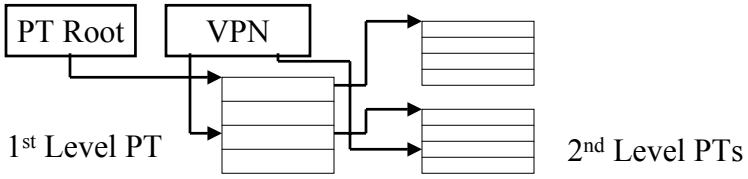


Page Table Size

- Page Table Size
  - Example #1: 32-bit VA, 4KB pages, 4-byte PTE
    - 1M Pages, 4MB Page Table
  - Example #2: 64-bit VA, 4KB pages, 4-byte PTE
    - 4P Pages, 16PB page table
- Page table reduction techniques
  - Multi-level page tables
  - Inverted page tables

Multi-Level Page Tables

- Most processes use only a tiny portion of total VA space
- Tree of page tables
  - L1 table points to L2 tables (and more if needed)
    - Different VPN bits are offsets at different levels
  - Save space: not all tables at all levels need to exist
  - Slow: Multi-hop chains of translations (space savings outweigh)



Multi Level Page Tables

- 32-bit address space, 4KB pages, 4 byte PTEs
- 2 level virtual page table
- 2nd-level tables are each the size of 1 data page
- Program uses only upper and lower 1MB of address space
- How much memory does page table take?
  - 4GB VM / 4KB pages => 1M pages
  - 4KB pages / 4B PTEs => 1K pages per 2nd level table
  - 1M pages / 1K pages per 2nd level table => 1K 2nd-level tables
  - 1K 2nd level tables + virtual page table => 4KB first level table
  - 1MB VA space + 4KB pages => 256 PTEs => 1 2nd level table
  - Memory = 1st level table (4KB) + 2 * 2nd level table (4KB) = 12KB

Inverted Page Table

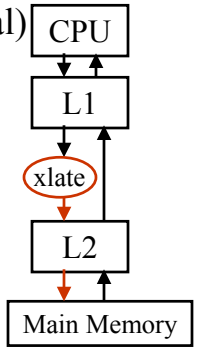
- Observation: don't need more PTEs than physical memory pages
- Apply hashing function to VA
  - Hash virtual addresses into array of PTEs
    - (hash collisions are chained)
  - Table is proportional to memory size (not VA size)
    - Page table size = (memory size / page size) * (PTE size + pointer)
  - Slow searches => PTE pointer chasing

Address Translation

- How does address translation really work?
- Two-level mapped page tables
  - Several levels of indirection: 3 memory accesses for 1 virtual memory access (slow!)
  - Processes do not read page table + translate: system does
- Hardware involvement: Translation Lookaside Buffer
  - Cache dedicated to these translations

Fast Translation: Virtual Caches

- First-level caches are "virtually addressed"
- L2 and main memory are "physically addressed"
- Address translation only on a miss (not critical)
- Why not?
  - Protection: xlate checks page level protection
  - Context switch: Cache flush required (PID tags?)
  - I/O: typically uses PAs (would need conversion to access L1 cache)
  - Synonyms: 2 VAs => 1 PA (2 copies in cache)

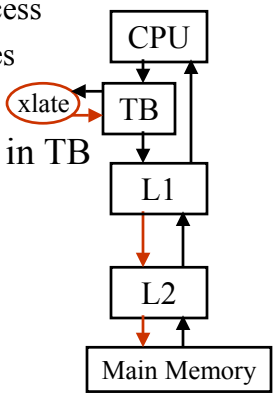


Synonyms: Another problem with Virtual Caches

- VA => PA is not always unique (sharing among processes)
- Memory location could be fetched into the cache by two different virtual addresses: consistency problem
- Solutions
  - Eliminate/Restrict sharing
  - Restrict sharing within a process, flush on context switch
  - Search all possible synonymous sets in parallel
  - Restrict page placement in OS such that index(VA) = index(PA)

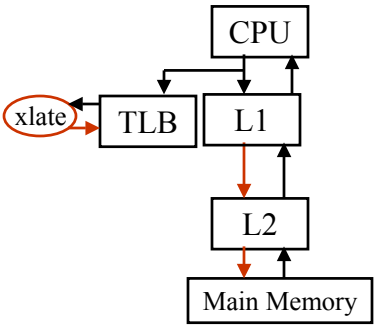
Fast Translation: Physical Caches with Translation Buffers

- Solution #2: First level caches are physical
  - Address translation before every cache access
  - Works fine with I/O, address space changes
  - SLOW
- Solution #2a: Cache recent translations in TB
  - Only go to page table on TB miss
    - Hit time problem: still 2 serial accesses



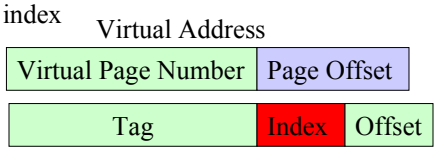
Fast Translation: Physical Caches with Translation Lookaside Buffers

- Solution #3: Address translation & L1 cache access in parallel
  - Translation lookaside buffer (TLB)
  - Fast (one step access)
  - No problems changing VA spaces
  - Keeps I/O coherent



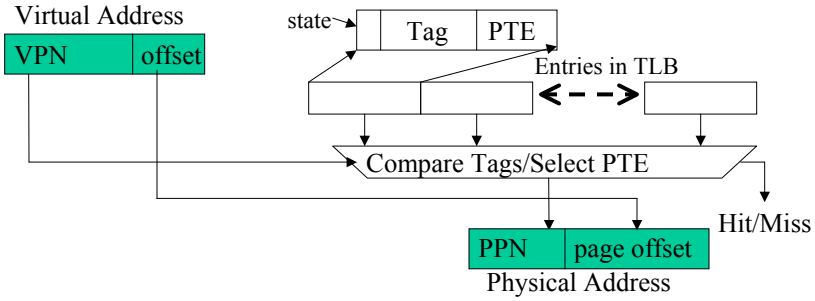
Cache: Virtual Index, Physical Tag

- Physical cache with virtual address
  - Only cache index matters for access
  - Only part of virtual address changes during translation
  - Make sure index is in untranslated part
    - Index is within page offset
    - Virtual index == physical index



  - Fast
  - Restricts cache size (Block size * #sets <= page size)
  - Use associativity to increase size

Basic TLB Organization



- Fully Associative Structure
- Example: VA = 44bits, Page Size = 4MB, PA Space = 1GB
- VPN bits = bits (VA) - $\log_2$(page size) = 44 – 22 = 22 bits
- Physical Addr. = $\log_2$(PA Size) = 30 bits (8 PPN + 22 Page Offset)

Selecting Page Size

- Larger Page Size
  - Page table is smaller (inversely proportional to page size)
  - Larger page size may allow larger caches with virtually indexed, physically tagged caches (larger page offset)
  - Page transfers can be more efficient
  - More efficient TLB => reduces number of TLB misses
- Smaller Page Size
  - *Internal fragmentation*: contiguous region of virtual memory not a multiple of the page size
  - Process startup time (load in large pages for small processes)
- Multiple Page Sizes
  - Some processors support multiple choices => larger pages are powers of 2 times the smaller page sizes

Protection Basics

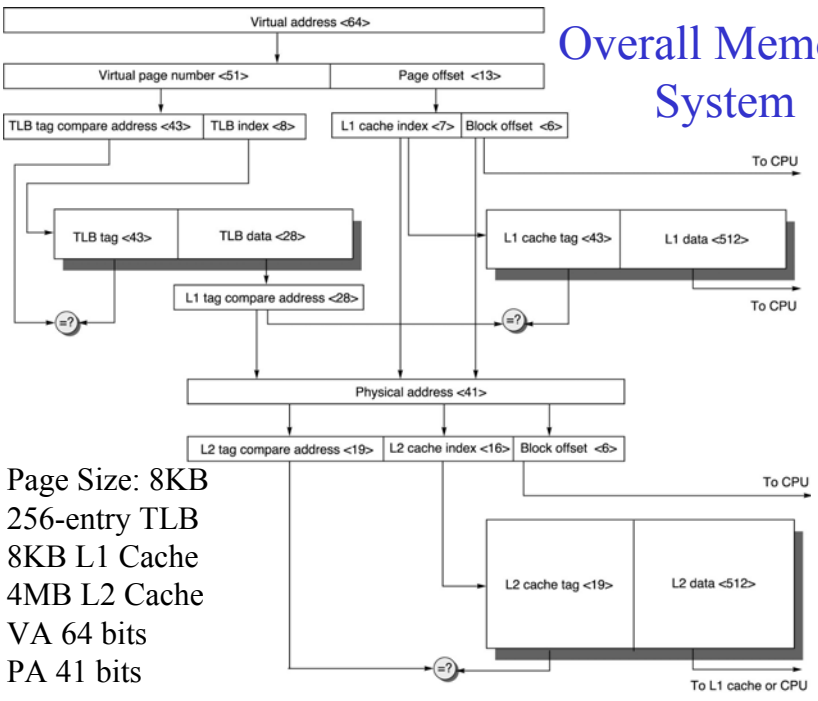
- Processes should not interfere with each other in multiprogramming environments
- Simplest Scheme: Two registers: Base + Bound

  (Base + Address) < Bound
- How to modify these registers?
- Programs can't modify or they can cheat!

Protection: Requirements

- OS kernel mode: special privileges available
  - Can access memory via physical addresses
  - Deals with base offset registers
  - System calls jump from user mode to kernel mode
  - User process says what it wants done, kernel does it
- More robust scheme:
  - Maintain separate VA spaces (page tables) per process
  - Must access memory through address translation
  - Do not know about address translation mechanism (page table)

Overall Memory System



Page Size: 8KB
256-entry TLB
8KB L1 Cache
4MB L2 Cache
VA 64 bits
PA 41 bits

Memory Summary

- Main Memory
  - DRAM is slow but dense
  - Interleaving/banking for high bandwidth
- Virtual Memory, Address Translation, Protection
  - Larger memory, protection, relocation, multiprogramming
  - Page tables
  - TLB: cache translations for speed
    - Access in parallel with cache tags