National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 12: Virtual Memory

CS10014 Computer Organization

Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS252 at ETHZ
    - https://safari.ethz.ch/digitaltechnik/spring2023
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/

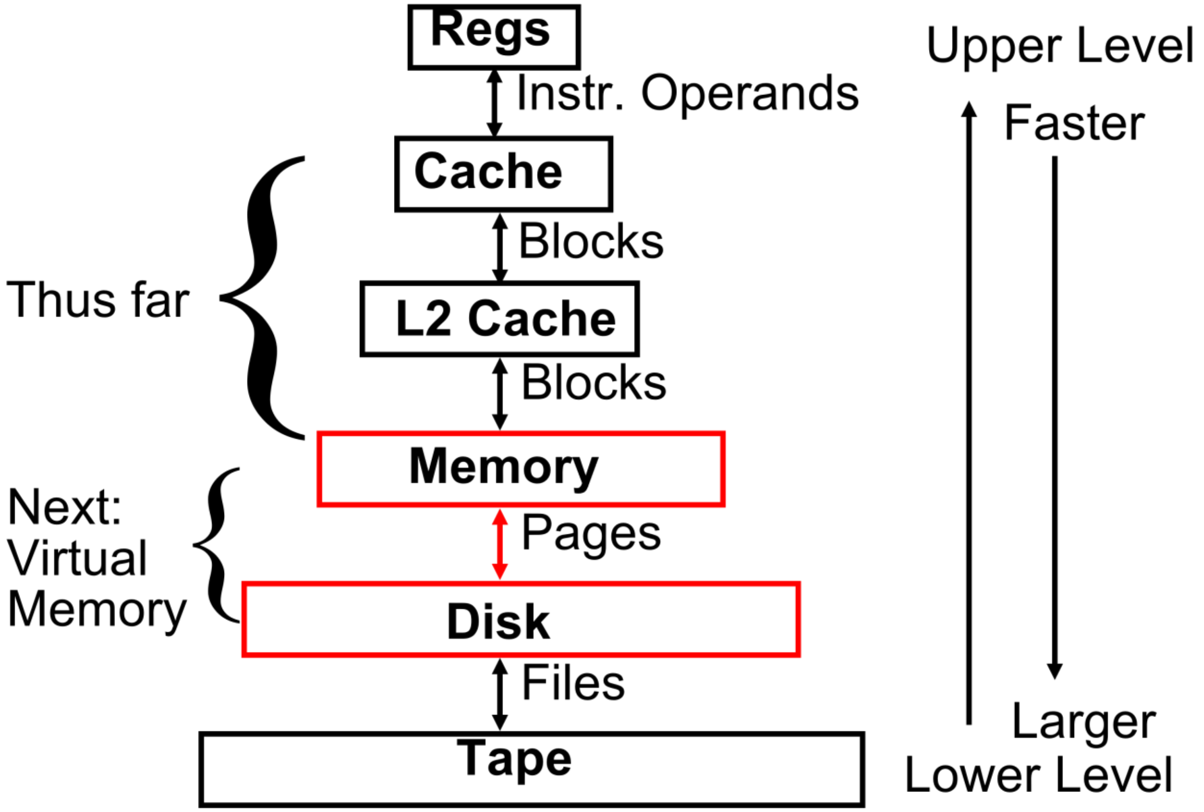
Outline

- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

Review

- **Cache design choices**
  - Size of cache: speed vs. capacity
  - Block size (i.e., cache aspect ratio)
  - Write policy (write through vs. write back)
  - Associativity choice of N (direct-mapped vs. set vs. fully associative)
  - Block replacement policy
  - Multi-level caches

Memory Hierarchy (1/3)



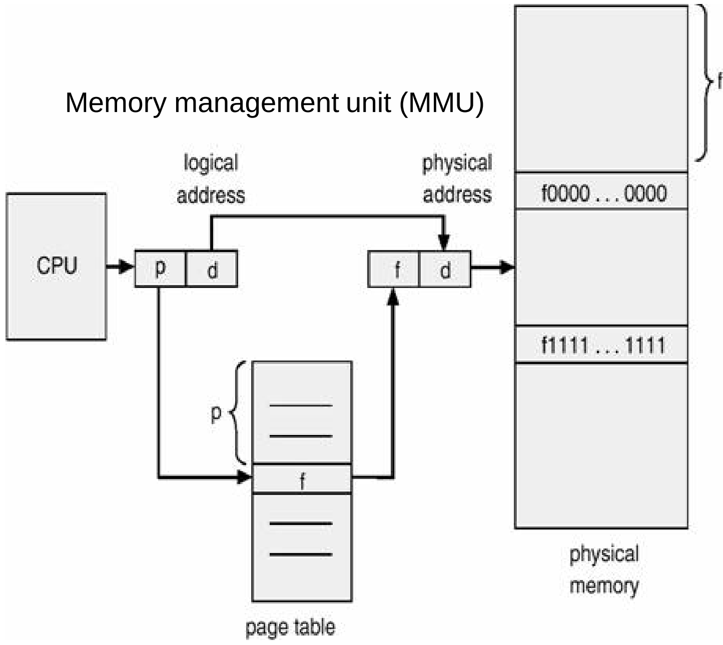
Memory Hierarchy (2/3)

- How does multiple apps (and the OS) share main memory?
  - Goal: <u>each applications thinks it has all of the memory</u>
- One app may want more memory than is in the system
  - App’s insn/data footprint may be larger than main memory
  - Requires main memory to act like a cache
    - With disk as next level in memory hierarchy (slow)
    - Write-back, write-allocate, large blocks or “pages”

Memory Hierarchy (3/3)

- Solutions
  - Part #1: treat memory as a **“cache”**
    - Store the overflowed blocks in “swap” space on disk
  - Part #2: add a level of indirection **(address translation)**
    - MMU is in the CPU

Memory management unit (MMU)

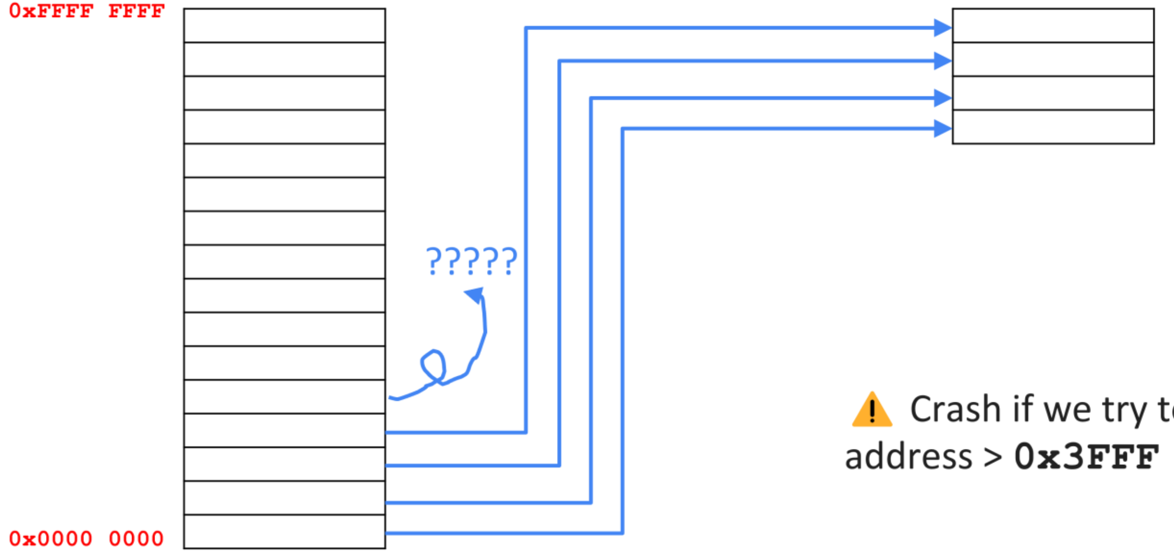


Virtual Memory Motivation (1/2)

- What if main memory is smaller than the program address space?

RV32I provides a 32-bit address space.
→ 2^{32} B = 4 GiB addressable memory

Suppose RAM is 1GiB.
→ 2^{30} B addressable memory.



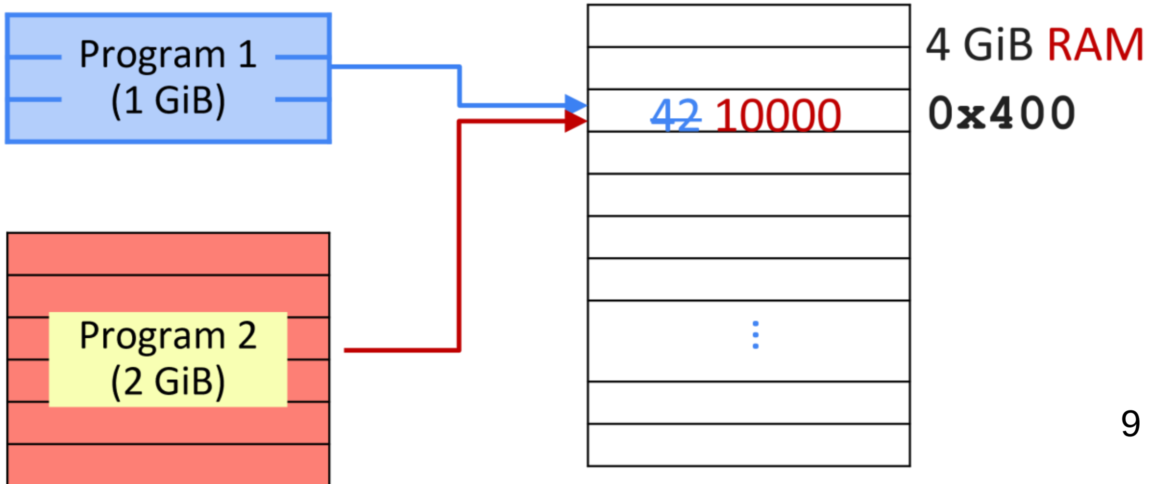
⚠ Crash if we try to access an address > **0x3FFF FFFF**!

Virtual Memory Motivation (2/2)

- What if two programs access the same memory address?
  - If all processes can access any 32-bit memory address, they can corrupt/crash others
  - Need protection (isolation) between processes

Program 1 stores your bank account balance at address **0x400**

Program 2 stores your video game score at address **0x400**

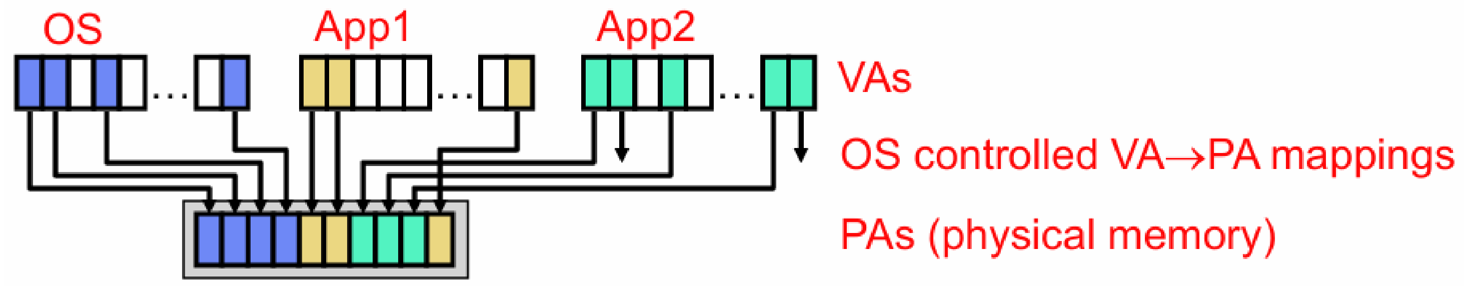


Outline

- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

Virtual Memory (1/5)

- **Virtual memory**
  - Level of indirection
  - Application generated addresses are **virtual addresses (VAs)**
    - Each process thinks it has its own 2^N bytes of address space
  - Memory accessed using **physical addresses (PAs)**
  - VAs translated to PAs at some coarse granularity (page)
  - OS controls VA to PA mapping for itself and all other processes



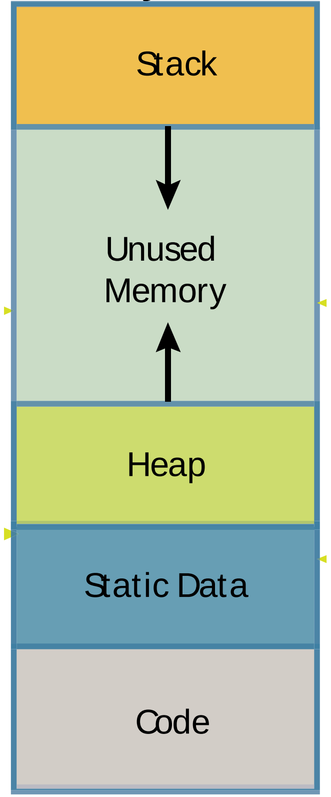
Virtual Memory (2/5)

- Programs use **virtual addresses (VA)**
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit)
- Memory uses **physical addresses (PA)**
  - PA size (M) typically M<N, especially if N = 64
- VA -> PA at page granularity (VP -> PP)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap) or nowhere (if not yet touched)

Virtual Memory (3/5)

- A program's address contains 4 regions
  - **Stack:**
    - local variables, grows downward
  - **Heap:**
    - space requested for pointers via malloc(); resizes dynamically, grows upward
  - **Static data:**
    - Variables declared outside main, does not grow or shrink
  - **Code:**
    - Loaded when program starts, does not change

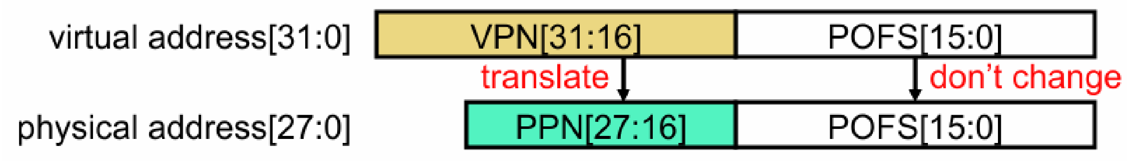
A process memory address layout



Virtual Memory (4/5)

- **Uses of virtual memory**
  - Isolation and multi-programming
  - Each app thinks it has 2^N B of memory, its stack starts 0xFFFFFFFF, …
  - Apps prevented from reading/writing each other’s memory
- **Protection**
  - Each page with a read/write/execute permission set by OS
- **Inter-process communication**
  - Map same physical pages into multiple virtual address spaces
  - Or share files via the UNIX mmap() call

Virtual Memory (5/5)



- **Address translation**
  - VA -> PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
  - Translate VPN into physical page number (PPN)
  - POFS is not translated
  - VA -> PA = [VPN, POFS] -> [PPN, POFS]
- Example above
  - 64KB pages -> 16-bit POFS
  - 32-bit machine -> 32-bit VA -> 16-bit VPN
  - Maximum 256 MB memory -> 28-bit PA -> 12-bit PPN

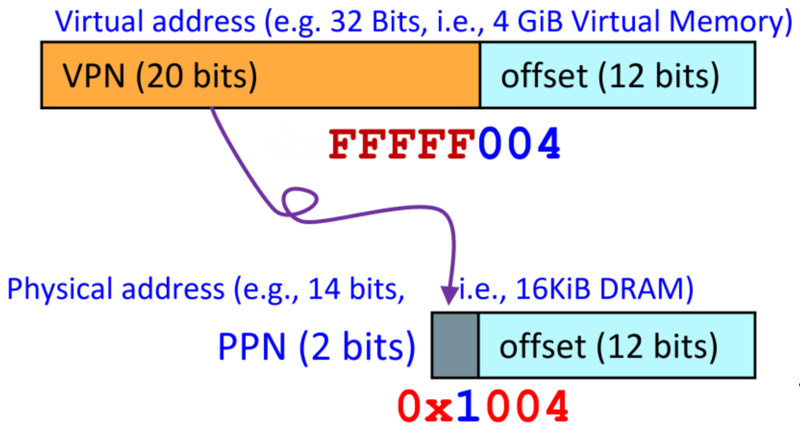
Outline

- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

Paged Memory (1/5)

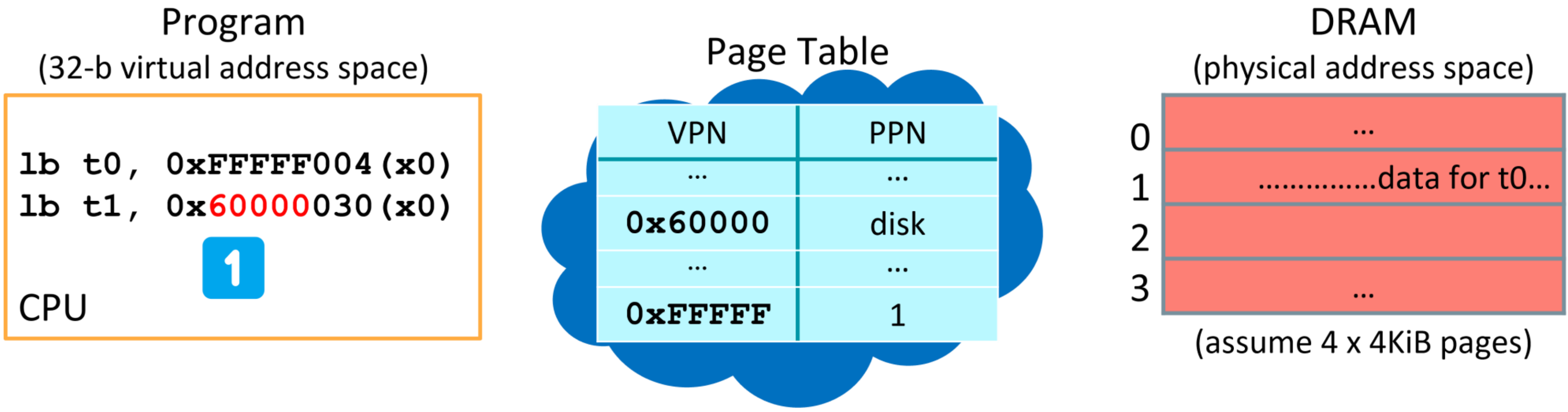
- The concept of "paged memory" dominates
  - Physical memory (DRAM) is broken into pages
  - A disk access loads an entire page into memory
  - Typical page size: 4KiB+ (on modern OSs)
    - Need 12 bits of page offset to address all 4KiB

Memory translation maps Virtual Page Number (VPN) to a Physical Page Number (PPN)


Virtual address (e.g. 32 Bits, i.e., 4 GiB Virtual Memory)
VPN (20 bits)
offset (12 bits)
0xFFFFF004
Physical address (e.g., 14 bits, i.e., 16KiB DRAM)
PPN (2 bits)
offset (12 bits)
0x1004

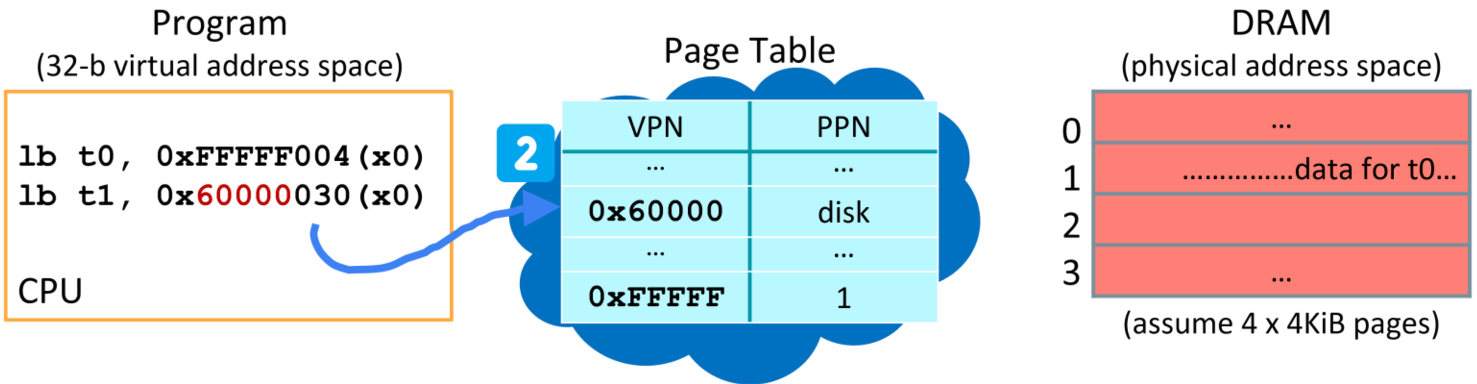
Paged Memory (2/5)

- How a program accesses memory?
  - Program executes a load specifying a virtual address (VA)



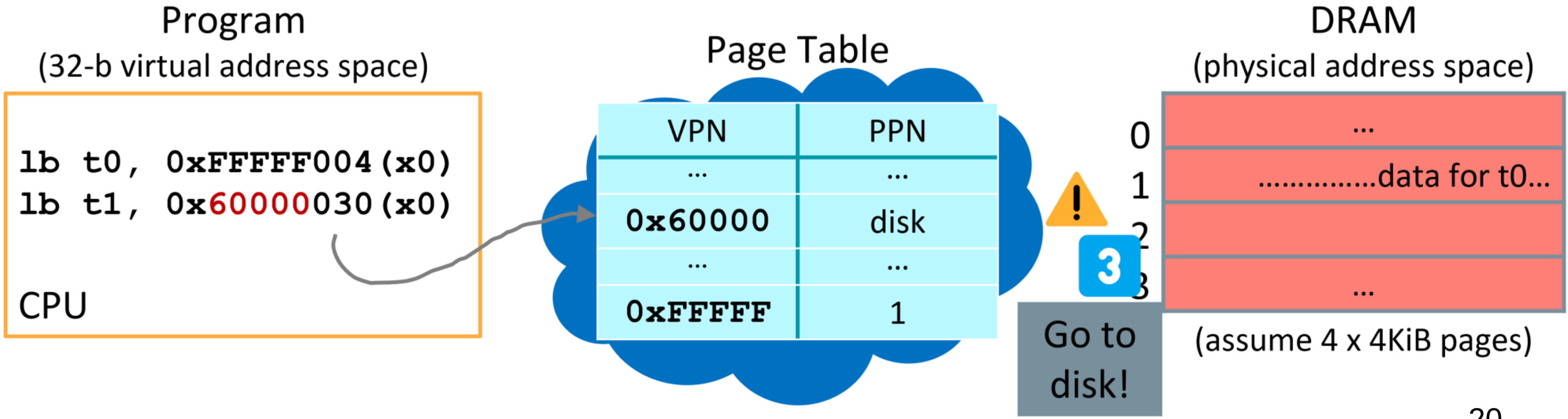
Paged Memory (3/5)

- How a program accesses memory?
  - Computer translates VA to the physical address (PA) in memory
    - Extract virtual page number (VPN) from VA, e.g. top 20 bits if page size 4KiB = 2^{12} B
    - Look up physical page number (PPN) in page table
    - Construct PA: physical page number + offset (from virtual address)



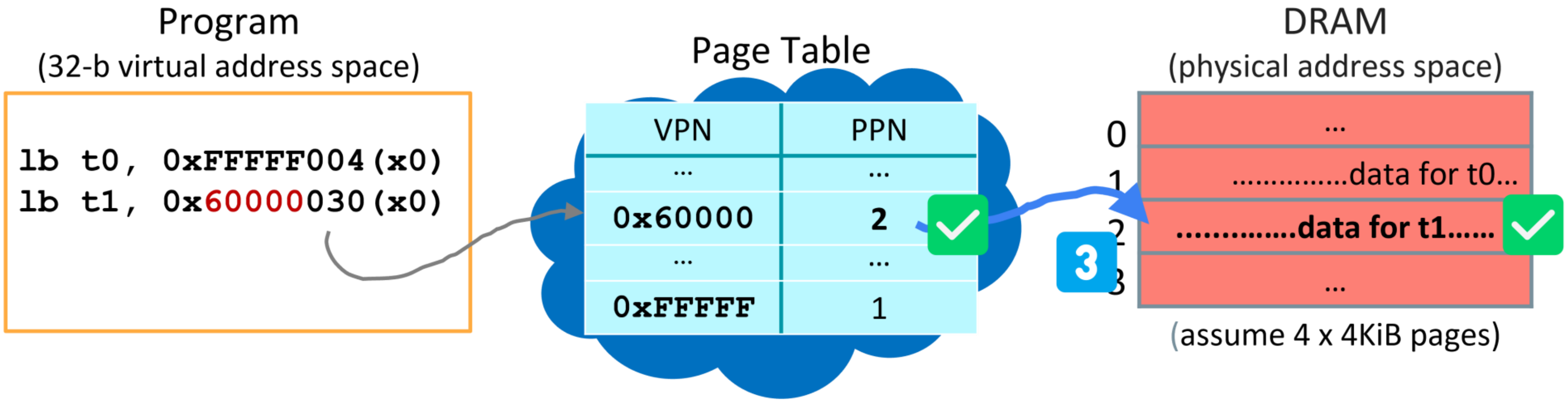
Paged Memory (4/5)

- How a program accesses memory?
  - If the physical page is not in memory, then OS loads it in from disk



Paged Memory (5/5)

- How a program accesses memory?
  - The OS reads memory at the PA and returns the data to the program

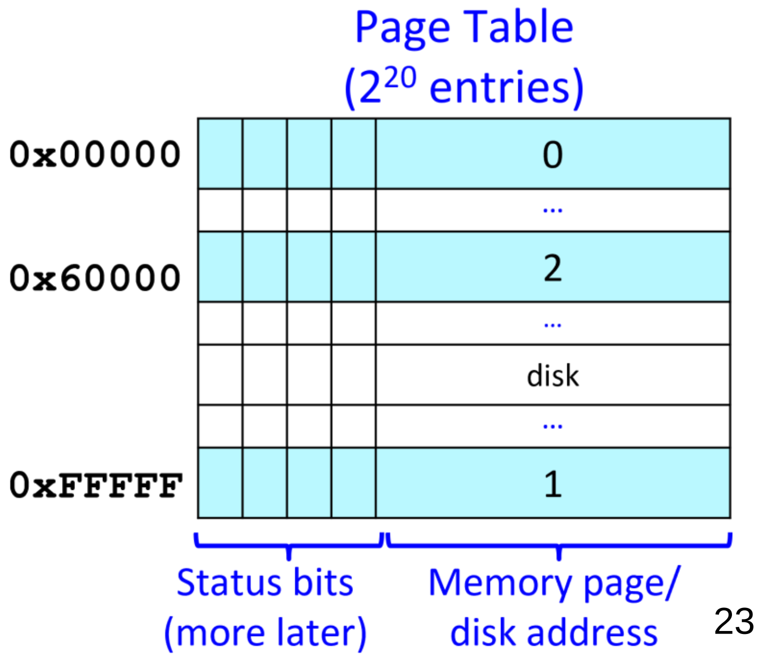


Outline

- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

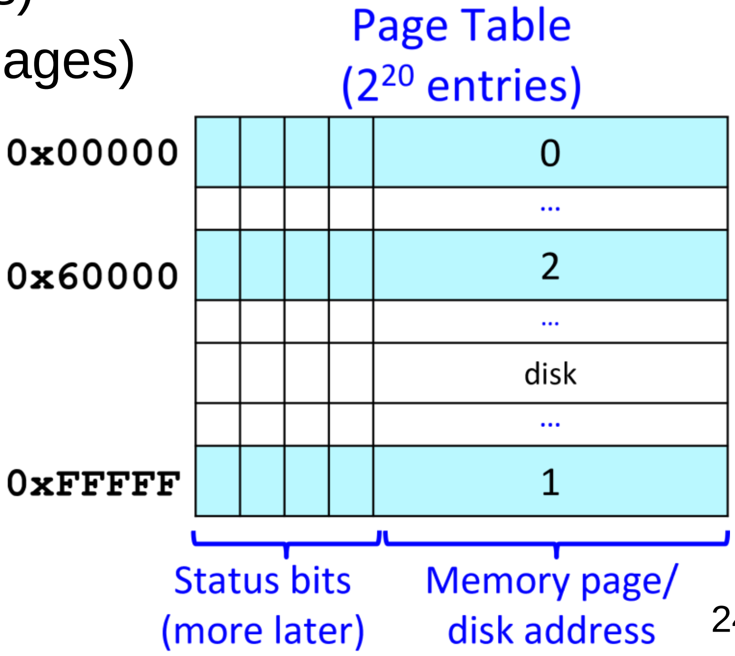
Paged Table (1/11)

- A page table is NOT a cache
  - A page table does not have data
  - It is a lookup table
  - All VPNs have a valid entry
  - Page tables are stored in the main memory



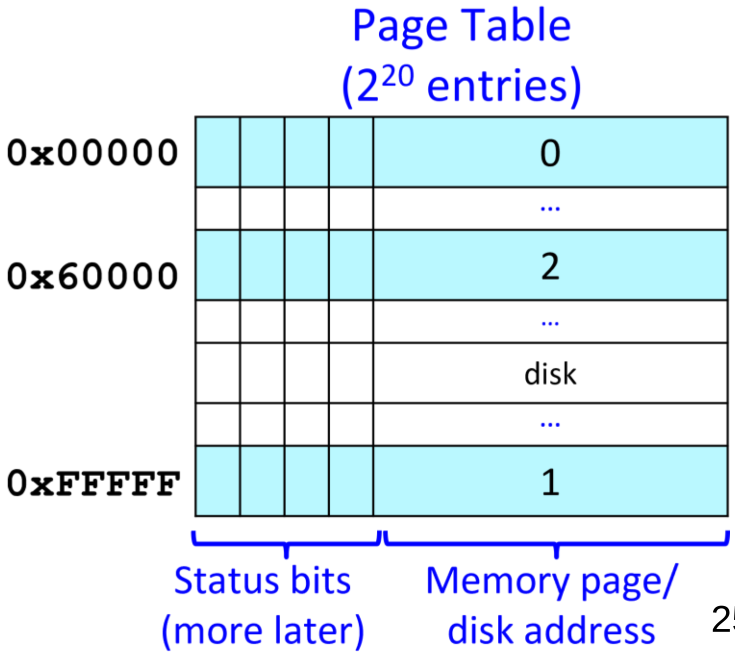
Paged Table (2/11)

- 32-bit virtual address space, 4-KiB pages
  - 2^{32} virtual addresses / (2^{12} B/pages)
  - = 2^{20} virtual page numbers (1MB pages)
- One page table per process
  - One entry per virtual page number
  - Entry has physical page number



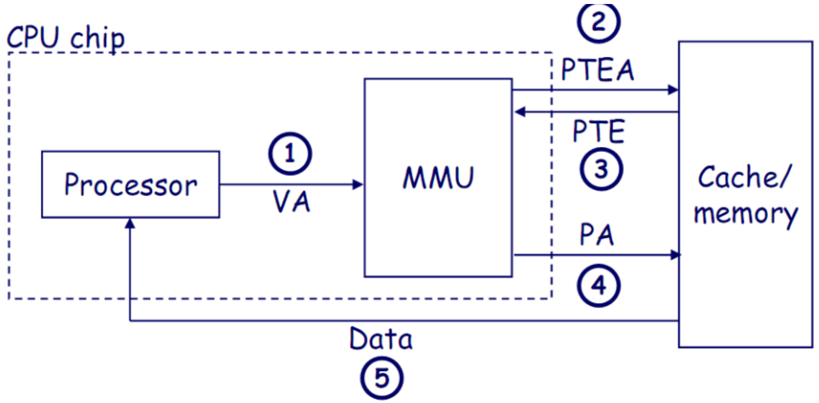
Paged Table (3/11)

- **Status Bits**
  - Write protection bit
    - On: If process writes to page trigger exception
  - Valid bit
    - On: Page is in RAM
  - Dirty bit
    - On: page on RAM is more up-to-date than page on dis



Paged Table (4/11)

- **Page hit**
  - 1) Processor sends virtual address to MMU
  - 2 – 3) MMU fetches PTE from page table in memory
  - 4) MMU sends physical address to L1 cache
  - 5) L1 cache sends data word to processor

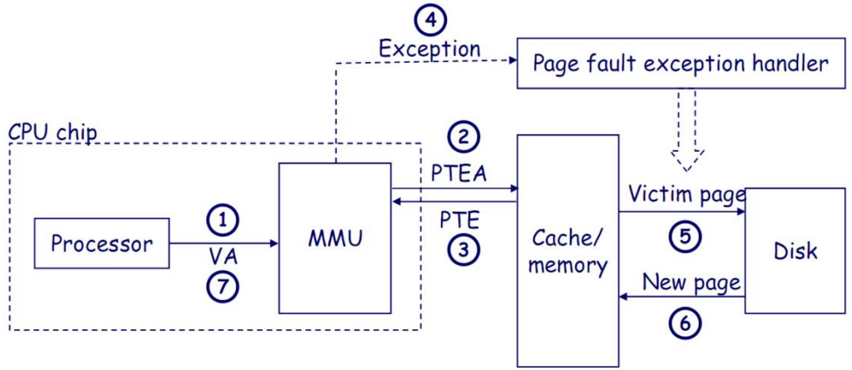


Paged Table (5/11)

- **Page faults**
  - Page table entries store status to indicate if the page is in memory (DRAM) or only on disk
    - One each memory access, check the page table entry “valid” status bit
  - Valid -> in DRAM
    - Read/write data in DRAM
  - Not valid -> on disk
    - Trigger a page fault; OS intervenes to allocate the page into DRAM
    - If out of memory, first evict a page from DRAM (LRU/FIFO/random)
    - Read request page from disk into DRAM
    - Finally, read/write data in DRAM

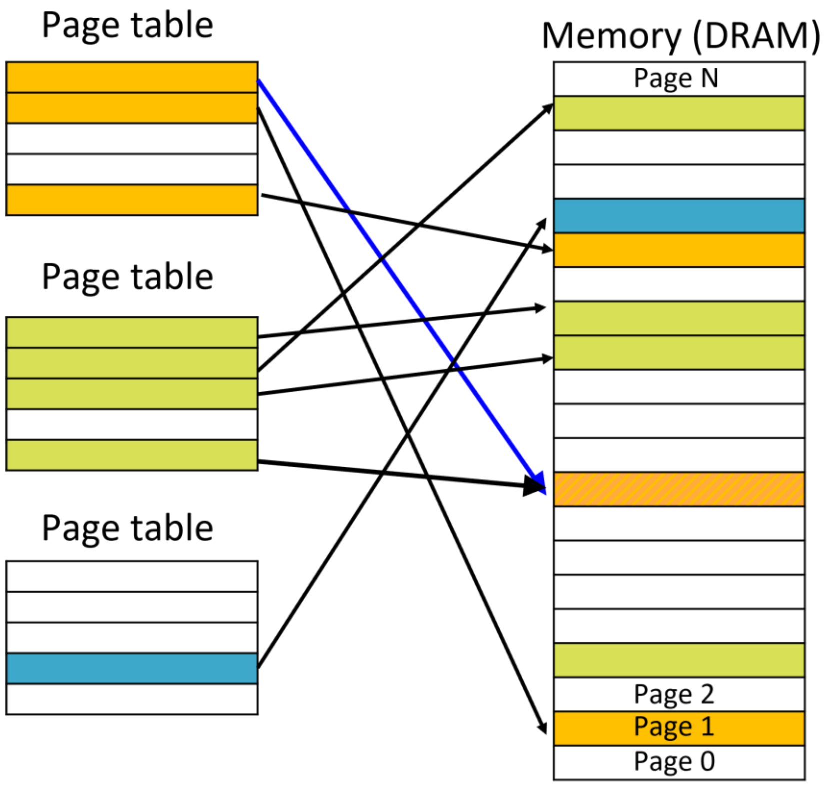
Paged Table (6/11)

- **Page faults**
  - 1) Process sends virtual address to MMU
  - 2-3) MMU fetches PTE from page table in memory
  - 4) Valid bit is 0, so MMU triggers page fault exception
  - 5) Handler identifies victim, and if dirty pages it out to disk
  - 6) Handler pages in new page and updates PTE in memory
  - 7) Handler returns to original process, restarting faulting instruction



Paged Table (7/11)

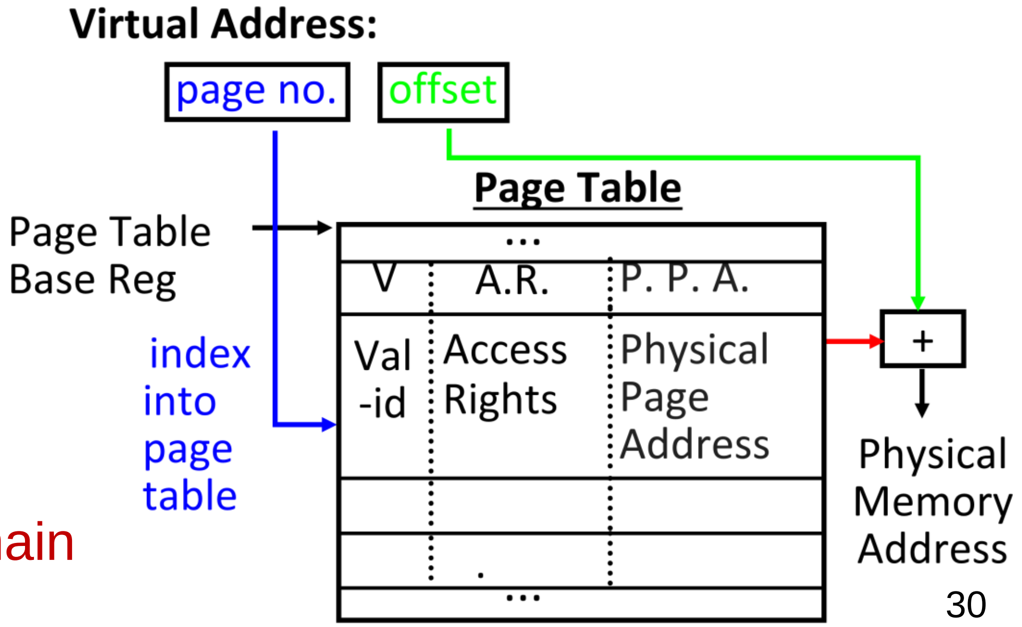
- Each process has a dedicated page table
  - OS keeps track of which process is active
- **Isolation**: Assign processes different pages in DRAM
  - Prevent accessing other processes' memory
  - OS may assign some physical page to several processes (e.g. system data), sharing is also possible



Paged Table (8/11)

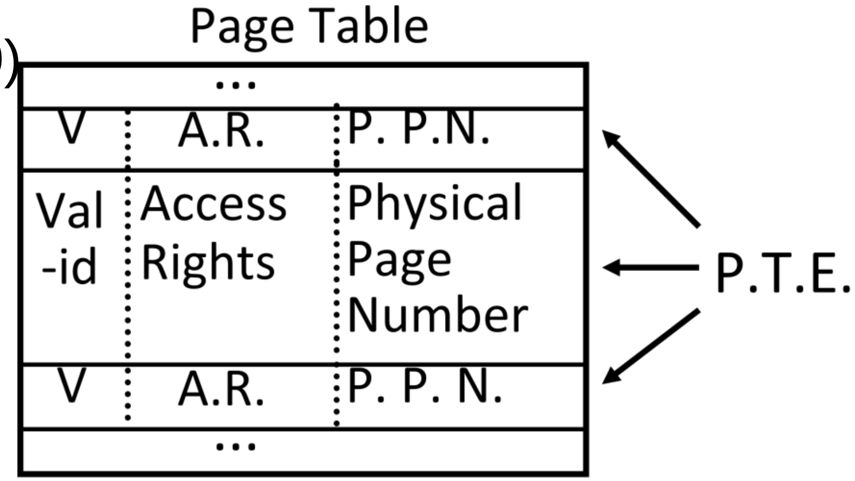
- A page table contains the mapping of virtual address to physical locations
    - Each process has its own page table
    - OS changes page tables by changing contents of Page Table Base Register

Page tables are stored in main memory



Paged Table (9/11)

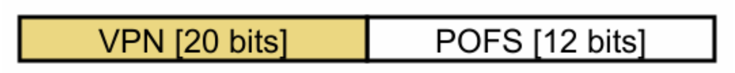
- **Page Table Entry (PTE) format**
  - Contains either physical page number or indication not in main memory
  - OS maps to disk if Not Valid (V=0)
  - If valid, also check if have permission to use page
    - Access Rights (A.R.) may be Read Only, Read/Write, Executable



Paged Table (10/11)

Cache version	Virtual Memory vers.
Block or Line	Page
Miss	Page Fault
Block Size: 32-64B	Page Size: 4K-8KB
Placement:	Fully Associative
Direct Mapped,	
N-way Set Associative	
Replacement:	Least Recently Used
LRU or Random	(LRU)
Write Thru or Back	Write Back

Paged Table (11/11)



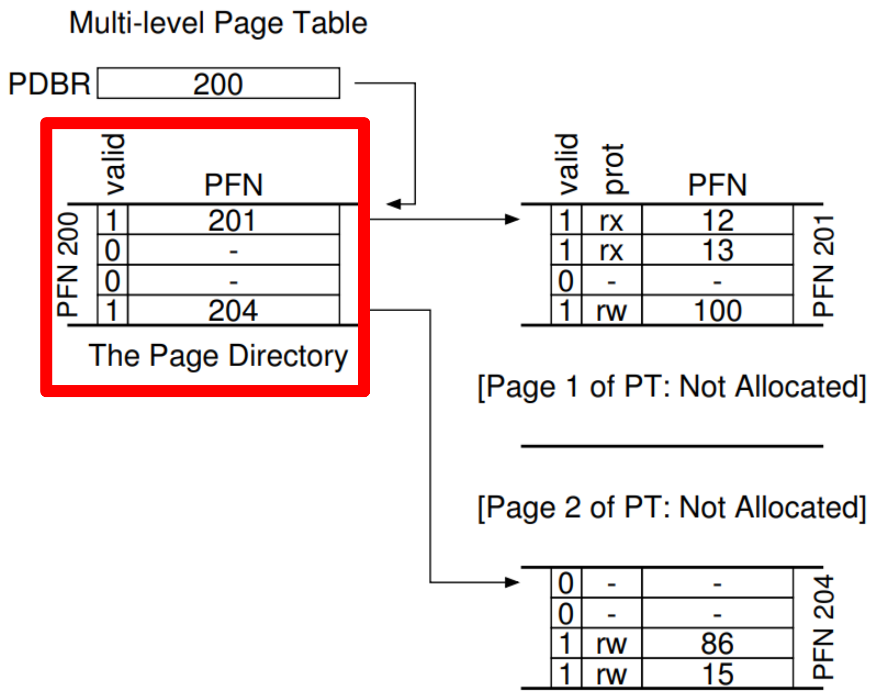
- How big is a page table on the following machine?
  - 32-bit machine->32-bit VA -> 2^{32} = 4GB virtual memory
  - 4B page table entries (PTEs)
  - 4KB pages
  - 4GB virtual memory / 4KB page size -> 1M VPs
  - 1M VPs * 4 bytes per PTE -> 4MB
- What is the problem when increasing page size from 4 KB to 16 KB?
  - **Internal fragmentation** (big pages lead to waste within each page)
- Page tables can get big
  - There are ways of making them smaller

Outline

- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

Multi-Level Page Table (1/8)

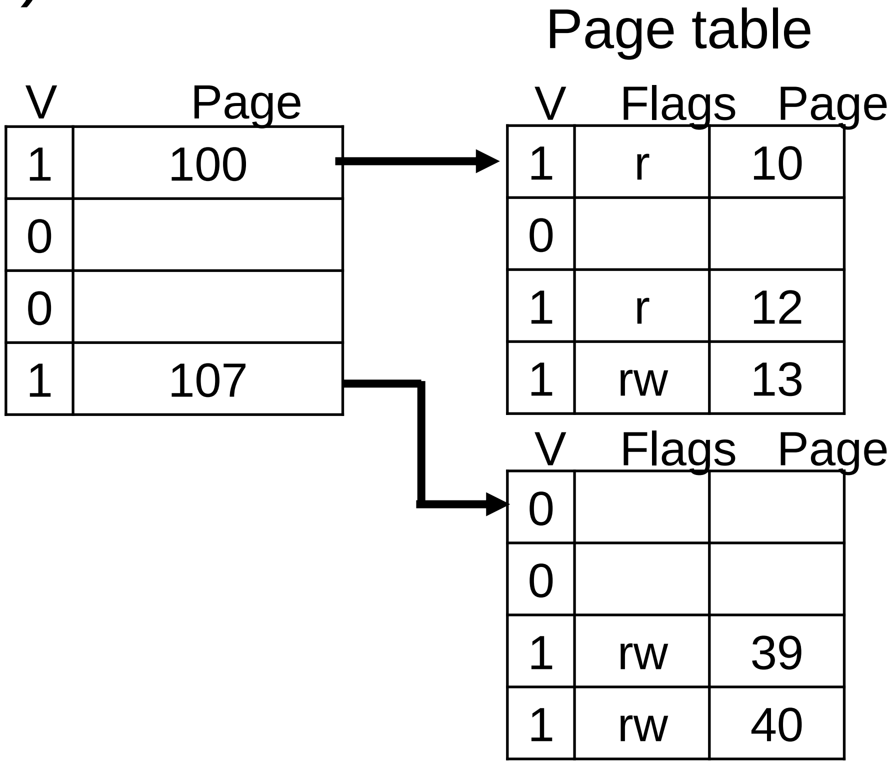
- **Multi-level page table**
  - Chop up the page table into page-sized units
  - **Page directory** tells where a page of the page table is
    - A number of **page directory entries (PDE)**
    - A **page frame number (PFN),** and a valid bit



https://pages.cs.wisc.edu/~remzi/OSTEP/vm-smalltables.pdf

Multi-Level Page Table (2/8)

- What are the advantages of multi-level page table?
  - Only allocate “using” page-table space
  - Compact and supports **sparse** address space



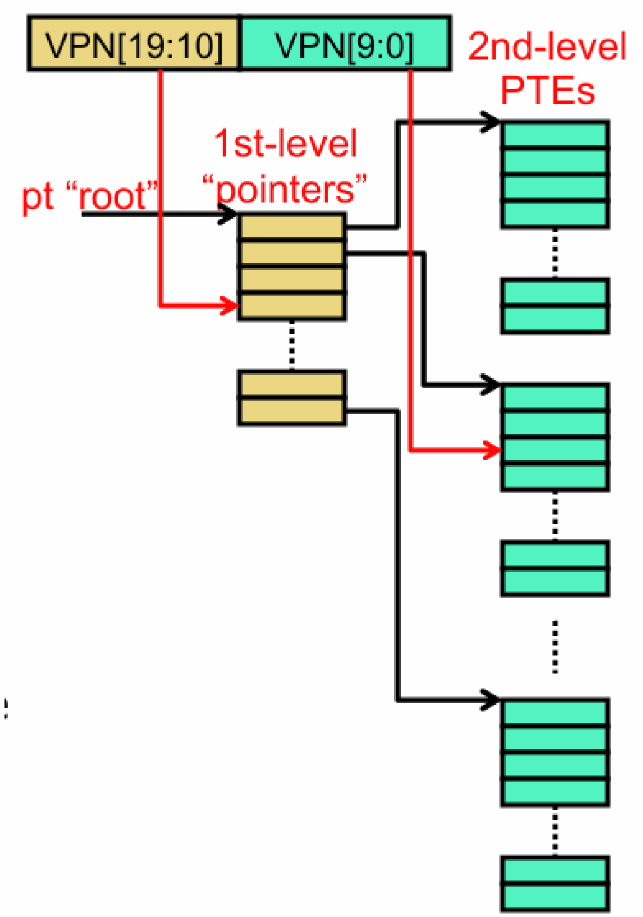
V	Page
1	100
0	
0	
1	107

V	Flags	Page
1	r	10
0		
1	r	12
1	rw	13

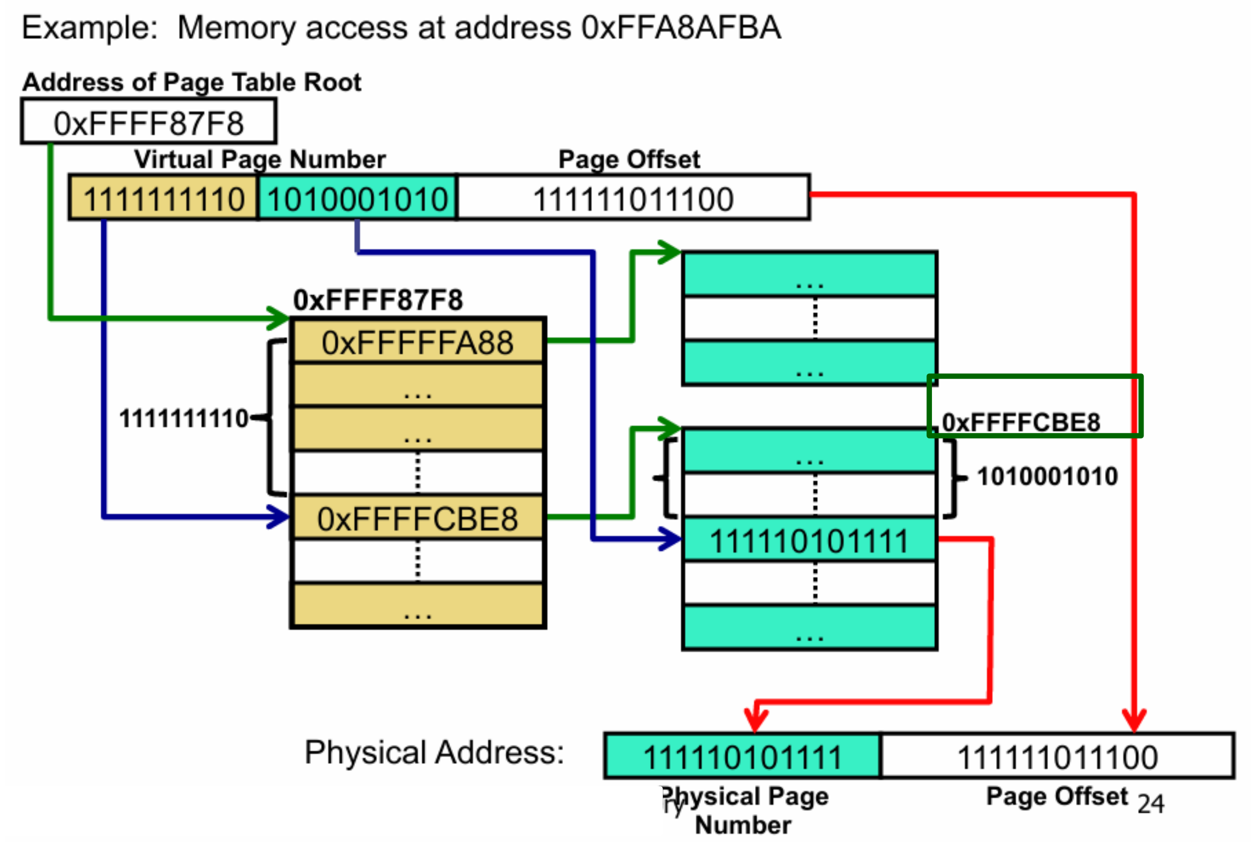
V	Flags	Page
0		
0		
1	rw	39
1	rw	40

Multi-Level Page Table (3/8)

- **Multi-level page tables**
  - Tree of page tables ("trie")
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels
- 20-bit VPN
  - Upper 10 bits index 1st –level table
  - Lower 10 bits index 2nd-level table



Multi-Level Page Table (4/8)



Multi-Level Page Table (5/8)

- **Have we saved any space?**
    - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
    - Yes, but
- **Large virtual address regions unused**
    - Corresponding 2nd-level tables need not exist
    - Corresponding 1st-level pointers are null
- **How large for contiguous layout of 256 MB?**
    - Each 2nd-level table maps 4MB of virtual addresses
    - One 1st-level + 64 2nd-level pages
    - 64 total pages = 260 KB (much less than 4MB)

Multi-Level Page Table (6/8)

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Initially
  - Page size = 8 KB = 2^{13} B
  - Virtual address space size = 2^{46} B
  - PTE = 4 B = 2^{2} B
  - Number of pages or number of entries in page table = 2^{46} B / 2^{13} B = 2^{33}
  - Size of page table = 2^{33} x 2^{2} B = 2^{35} B

Multi-Level Page Table (7/8)

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Now, size of page table > page size (2^{35} B > 2^{13} B)
  - Create one more level
  - Number of page tables in last level
    2^{35} B / 2^{13} B = 2^{22}
  - Size of page table [second last level]
  - 2^{22} x 2^{2} B = 2^{24} B

Multi-Level Page Table (8/8)

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Now, size of page table > page size (2^{24} B > 2^{13} B)
  - Create one more level [third last level]
  - Number of page tables in second last level
    = 2^{24} B / 2^{13} B = 2^{11}
  - Size of page table [third last level]=
    = 2^{11} x 2^{2} B = 2^{13} B = page size

Outline

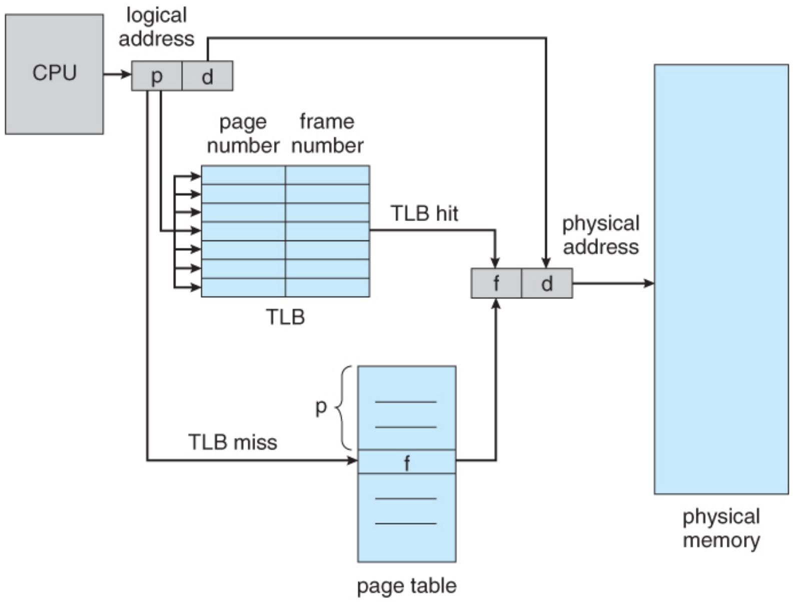
- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

Translation Lookaside Buffer (TLB) (1/4)

- Good virtual memory design should be fast (~1 clock cycle) and space efficient
    - Every instruction/data access needs address translation
- But if page tables are in memory
    - we must perform a page table walk per instruction/data access
        - Single-level page table: 2 memory accesses
        - Two-level page table: 3 memory accesses
    - Solutions: Cache some translations in **Translation Lookaside Buffer**

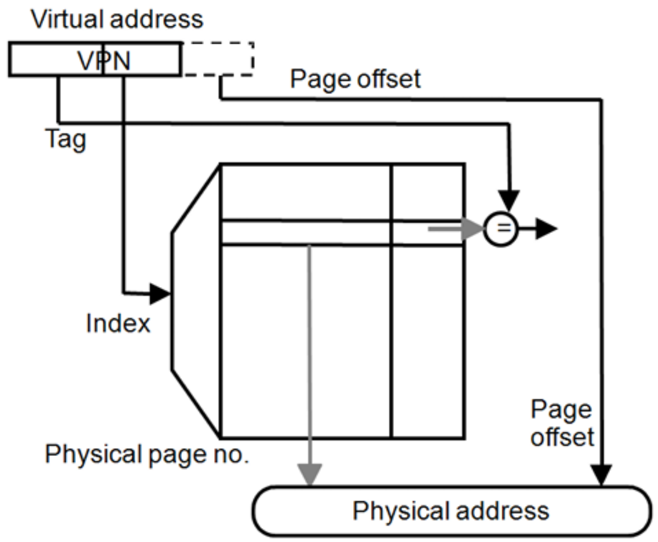
Translation Lookaside Buffer (TLB) (2/4)

- Translation lookaside buffer (TLB)
  - **Small cache**: 16-64 entries
  - **Associative** (4+ way or fully associative common)
  - Exploit <u>temporal locality </u>in page table
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler, walk page table

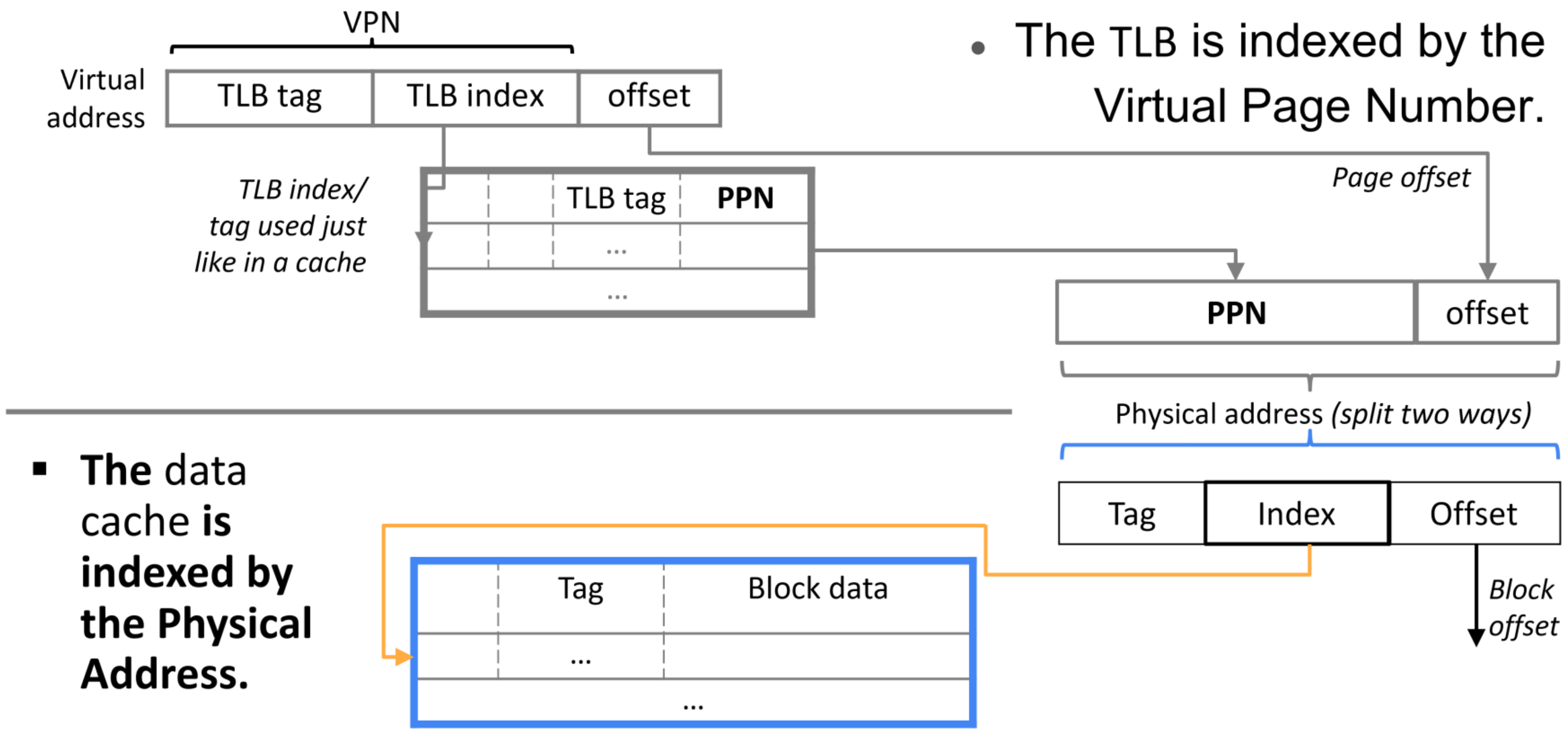


Translation Lookaside Buffer (TLB) (3/4)

- Translation lookaside buffer (TLB)
  - A cache of address translations
  - Avoid accessing the page table on every memory access
  - **Index** = lower bits of VPN (virtual page #)
  - **Tag** = unused bits of VPN + process ID
  - **Data** = a page-table entry
  - **Status** = valid, dirty



Translation Lookaside Buffer (TLB) (4/4)



- The TLB is indexed by the Virtual Page Number.

- **The** data cache **is indexed by the Physical Address.**

Outline

- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

Handling TLB Misses (1/4)

- The TLB is small; it cannot hold **all** PTEs
  - Some translation requests will inevitably miss in the TLB
  - Must access memory to find the required PTE
    - Called walking the page table
    - Large performance penalty

TLB hit: Single-cycle translation
TLB miss: Page table walk to refill

Handling TLB Misses (2/4)

- **Approach #1: Hardware managed (e.g., x86)**
  - The hardware does the page walk
  - The hardware fetches the PTE and inserts it into the TLB
    - If the TLB is full, the entry **replaces** another entry
  - Done transparently to system software
  - Can employ specialized structures and caches
    - E.g., page walkers and page walk caches

Handling TLB Misses (3/4)

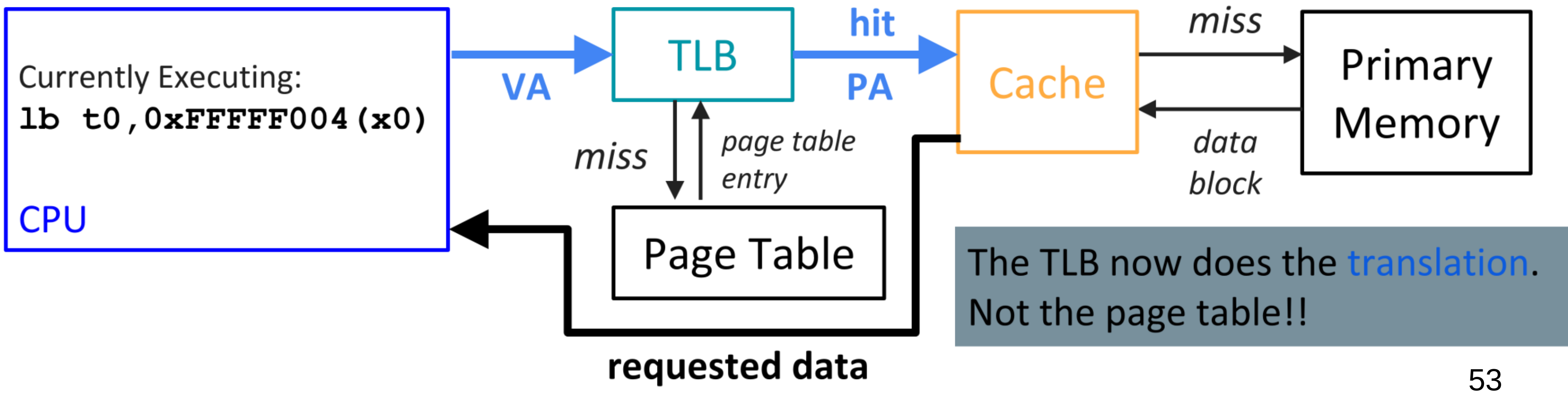
- **Approach #2: Software managed (e.g., MIPS)**
  - The hardware raises an exception
  - The operating system does the page walk
  - The operating system fetches the PTE
  - The operating system inserts/evicts entries in the TLB

Handling TLB Misses (4/4)

- **Hardware managed TLB**
  - + No exception on TLB miss. Instruction just stalls
  - + Independent instructions may execute and help tolerate latency
  - + No extra instructions/data brought into caches
- **Software-managed TLB**
  - + The OS can define the page table organization
  - + More sophisticated TLB replacement policies are possible
  - - Need to generate an exception -> performance overhead due to pipeline flush

Memory Access (1/2)

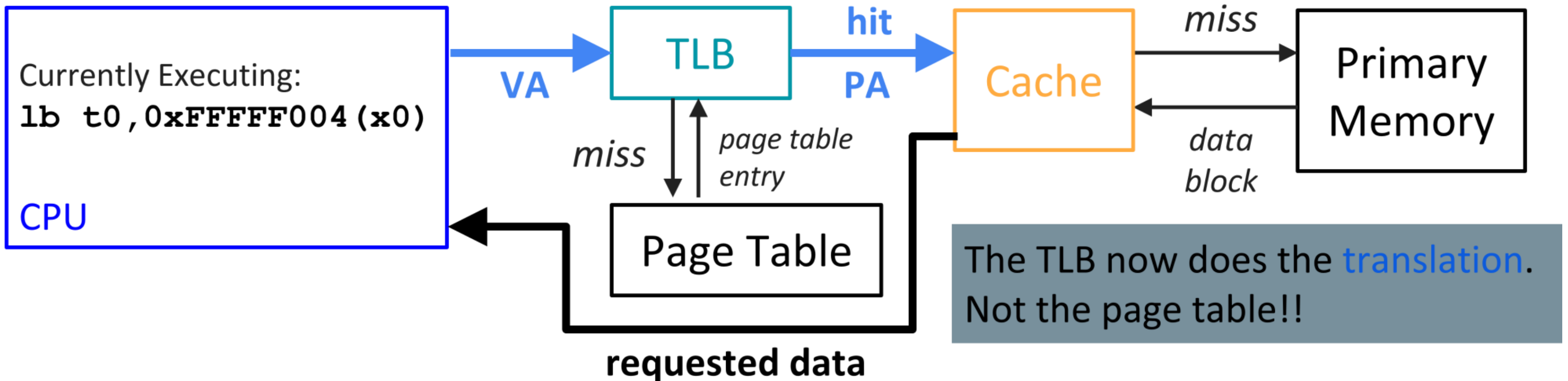
- Can a cache hold the requested data if the corresponding page is not in main memory?
  - NO !



The TLB now does the translation. Not the page table!!

Memory Access (2/2)

- On a memory reference, which block should we access first? When should we translate virtual addresses?
    - We will assume physically indexed, physically tagged caches (other design exist)
    - This means TLB first, then cache



The TLB now does the translation. Not the page table!!

Conclusion

- Virtual memory gives the illustration of “infinite” capacity
- A subset of virtual pages are located in physical memory
- A page table maps virtual pages to physical pages
  - Address translation
- A TLB speeds up address translation
- Multi-level page tables keep the page table size in check